Formalisation and execution of Linear Algebra: theorems and algorithms

Jose Divasón Mallagaray

Dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Supervisors: Dr. D. Jesús María Aransay Azofra
Dr. D. Julio Jesús Rubio García

UNIVERSIDAD DE LA RIOJA

Departamento de Matemáticas y Computación
Logroño, June 2016

Examining Committee
Dr. Francis Sergeraert *(Université Grenoble Alpes)*
Prof. Dr. Lawrence Charles Paulson *(University of Cambridge)*
Dr. Laureano Lambán *(Universidad de La Rioja)*

External Reviewers
Dr. Johannes Hölzl *(Technische Universität München)*
Ass. Prof. Dr. René Thiemann *(Universität Innsbruck)*

This work has been partially supported by the research grants FPI-UR-12, ATUR13/25, ATUR14/09, ATUR15/09 from Universidad de La Rioja, and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

Abstract

This thesis studies the formalisation and execution of Linear Algebra algorithms in Isabelle/HOL, an interactive theorem prover. The work is based on the HOL Multivariate Analysis library, whose matrix representation has been refined to datatypes that admit a representation in functional programming languages. This enables the generation of programs from such verified algorithms. In particular, several well-known Linear Algebra algorithms have been formalised involving both the computation of matrix canonical forms and decompositions (such as the Gauss-Jordan algorithm, echelon form, Hermite normal form, and QR decomposition). The formalisation of these algorithms is also accompanied by the formal proofs of their particular applications such as calculation of the rank of a matrix, solution of systems of linear equations, orthogonal matrices, least squares approximations of systems of linear equations, and computation of determinants of matrices over Bézout domains. Some benchmarks of the generated programs are presented as well where matrices of remarkable dimensions are involved, illustrating the fact that they are usable in real-world cases. The formalisation has also given place to side-products that constitute themselves standalone reusable developments: serialisations to SML and Haskell, an implementation of algebraic structures in Isabelle/HOL, and generalisations of well-established Isabelle/HOL libraries. In addition, an experiment involving Isabelle, its logics, and the formalisation of some underlying mathematical concepts presented in Voevodsky's simplicial model for Homotopy Type Theory is presented.

Contents

Chapter 1

Introduction

1.1 Motivation

9th May 2015. A military Airbus A400M crashed near Seville (Spain), after a failed emergency landing during its first flight. The four crew members on board were killed in the accident. Investigators found evidence the crash had been caused by software problems [26].

A software bug will usually cause a program to crash, to have an unexpected behaviour or to return erroneous computations. Nothing will *normally* explode and the bug will only cause an inconvenience. However, software and hardware faults can also have a huge economic impact, considerably damage enterprises' reputation, and the worst of all is that it can also cause loss of human lives.

Likely, the most world-renowned instances happened in the nineties. It took the European Space Agency 10 years and €6000 million to produce the Ariane-5 rocket. It exploded 36.7 seconds after the launch due to a data-conversion from a 64-bit floating-point number to a 16-bit signed integer value [7]. The Intel's Pentium II processor could return incorrect decimal results during mathematical calculations [116]. It caused a loss of about $475 million to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer.

Both of them are just two examples, but unfortunately the truth is that these are only the tip of a very large iceberg. A software flaw in the control part of the radiation therapy machine Therac-25 caused the death of six cancer patients in the late eighties, since they were exposed to an overdose of radiation (up to 100 times the intended dose) [105]. A software failure caused Mars Pathfinder to reset itself several times after it was landed on Mars in 1997 [132]. NASA had to remotely fix the code on the spacecraft. In 1999, the $125 million satellite Mars Climate Orbiter produced some results which were not converted from the United States customary unit into metric unit: the software calculated the force the thrusters needed to exert in pounds of force. A separate piece of software took the data assuming it was in the metric unit newtons. The satellite burned up in the Martian atmosphere during the orbital insertion [136].

Software development is error prone and examples of bugs with disastrous consequences make up a never-ending list. Thus, it is necessary to verify software somehow, in order to minimise possible faults. Software testing is one of the

major software verification techniques used in practice and it is a significant part, in both time and resources, of any software engineering project.

Mainly, software testing involves the execution of a program or application with the intent of finding software bugs, that is, running the programs and comparing the actual output of the software with the expected outputs. As exhaustive testing of all execution paths is infeasible (there are usually infinite many inputs), testing can never be complete. Subtle bugs often escape detection until it is too late. Quoting the famous theoretical computer scientist Edsger Dijkstra:

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> – Edsger W. Dijkstra, *Notes On Structured Programming*

Formal methods refer to "mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems" [75]. The value is that they provide a means to establish a correctness or safety property that is true for all possible inputs. One should never expect a hundred percent of safety, but the use of formal methods significantly increases the reliability and confidence in a computer program. Thus, formal methods are one of the *highly recommended* verification techniques for software development of safety-critical systems [23].

Formalisation of Mathematics is a related topic. Why formalise? The main answer is to improve the rigour, precision, and objectivity of mathematical proofs. There are plenty of examples of mathematical proofs that have been reviewed and then published containing errors. Sometimes these errors can be fixed, but other times they cannot and indeed published theorems were false. A book by Lecat [103] gives over 100 pages of errors made by major mathematicians up to the nineteenth century. Maybe today we would need many volumes.

A mathematical proof is rigorous when it has been formalised, that is, it has been written out as a sequence of inferences from the axioms (the foundations), each inference made according to one of the stated rules. However, to carry it out from scratch is tedious and requires much effort. In 1910, Whitehead and Russell formally proved $1 + 1 = 2$ after 379 pages of preliminary and related results in the book Principia Mathematica [153], the first sustained and successful actual formalisation of Mathematics. Russell finished exhausted.

> *My intellect never quite recovered [from the strain of writing Principia Mathematica]. I have been ever since definitely less capable of dealing with difficult abstractions than I was before.*
>
> – Bertrand Russell, *The Autobiography of Bertrand Russell*

Fortunately, nowadays we have computers and interactive theorem provers where formal proofs can be carried out and automatically checked by the computer. Interactive theorem provers are usually based on a small trusted kernel. A proof is only accepted by the theorem prover if it is a consequence of the few primitive inferences belonging to the kernel. Interactive theorem provers (such as Coq [45] and Isabelle [118]) are growing day by day and they have shown to

be successful in large projects, but to develop a complex mathematical proof in an interactive theorem prover is far from straightforward and it demands human-effort, resources and dedication to decompose it in little affordable milestones. For instance, the four colour theorem was formalised by Gonthier [70] and it took 5 years. The Kepler conjecture was formalised by Hales, after it was stated during the review process that the nature of the proof made it hard for humans to check every step reliably [81]. The formal proof took 11 years.

Computer Algebra systems (CAS) are used nowadays in various environments (such as education, diverse fields of research, and industry) and, after years of continuous improvement, with an ever increasing level of confidence. Despite this, these systems focus intensively on performance, and their algorithms are subject to continuous refinements and modifications, which can unexpectedly lead to losses of accuracy or correctness. On the other hand, theorem provers are specifically designed to prove the correctness of program specifications and mathematical results. This task is far from trivial, except for standard introductory examples, and it has a significant cost in terms of performance, paying off exclusively in terms of the simplicity and the insight of the programs one is trying to formalise.

In fact, in standard mathematical practice, formalisation of results and execution of algorithms are usually (and unfortunately) rather separate concerns. Computer Algebra systems are commonly seen as *black boxes* in which one has to trust, despite some well-known major errors in their computations [59], and mathematical proofs are more commonly carried out by mathematicians with *pencil & paper*, and sometimes *formalised* with the help of a proving assistant. Nevertheless, some of the features of each of these tasks (formalisation and computation) are considered as a burden for the other one; computation demands optimised versions of algorithms, and very usually *ad hoc* representations of mathematical structures, and formalisation demands more intricate concepts and definitions in which proofs have to rely on.

Fortunately, after years of continuous work, theorem proving tools have reduced this well-known gap, and the technology they offer is being used to implement and formalise state of the art algorithms and generate programs to, usually, functional languages, with a reasonable performance (see for instance [62,137]). Code generation is a well-known and admitted technique in the field of formal methods. Avigad and Harrison in a recent survey about formally verified Mathematics [20], enumerate three different strategies to verify "mathematical results obtained through extensive computations"; the third one is presented as "to describe the algorithm within the language of the proof checker, then extract code and run it independently".

In this thesis, we mainly present both formalisation of pure Mathematics (the Fundamental Theorem of Linear Algebra, algebraic structures, coordinates, and so on) and verification of Linear Algebra algorithms (Gauss-Jordan, Gram-Schmidt process, QR decomposition, echelon form and Hermite normal form). The connection between both fields has tried to be preserved, which is not usually carried out: most of the times algorithms are verified in the sense that they are formally proven to return, for a suitable input, an output which satisfies some properties. However, few times the *real* pure mathematical meaning of the algorithm is taken into account: for example, to triangularise a matrix by means of elementary operations is equivalent to apply a change of basis to a linear map.

As the title of this thesis points out, we seek *formalisation and execution*

at the same time, so our formalisations give room to verified algorithms which are later code-generated to the functional languages SML [112] and Haskell [86] following the third strategy quoted above. These algorithms are also refined to efficient structures in order to try to get a reasonable performance and make our verified programs usable in practice. They are also formally proven to be applicable to solve some of the central problems in Linear Algebra, such as computing the rank of matrices, computing determinants, inverses and characteristic polynomials, solving systems of linear equations, normal forms and decompositions, orthogonalisation of vectors, bases of the fundamental subspaces, and so on.

Besides, an *extra* chapter on foundations of Mathematics is presented in this thesis. Foundations of Mathematics are the basic pillars (axioms) from which all mathematical theorems are formulated and deduced. From the late nineteenth century, the study of the foundations of Mathematics has had a noteworthy interest. The *naive set theory* was one of its first attempts [35]. However, the celebrated Russell's paradox arose in 1901 spoiling it and showed that such a naive set theory was inconsistent [135]. Then, other foundations of Mathematics which avoid the paradox were proposed, such as the Zermelo-Fraenkel set theory. Soon after, a very important result was proven by Gödel in 1931: the consistency of any sufficiently strong formal mathematical theory cannot be proven in the theory itself [69]. This is widely accepted as to find a complete and consistent foundations for all Mathematics is impossible. The result was also formalised in Isabelle by Paulson in 2013 [128]. Finally, the Zermelo-Fraenkel set theory is nowadays accepted as the most common foundation of Mathematics.

Nevertheless, in the last few years a new question is gaining ground. *Will computers redefine the roots of Maths?* (See for instance the article of the same title in [92].) It all comes from Voevodsky's work. He has proposed a new foundations of Mathematics: the univalent foundations based on Homotopy Type Theory which try to bring the languages of Mathematics and computer programming closer together. This is thought to be a revolution [133].

> *Voevodsky told mathematicians that their lives are about to change. Soon enough, they're going to find themselves doing Mathematics at the computer, with the aid of computer proof assistants. Soon, they won't consider a theorem proven until a computer has verified it. Soon, they'll be able to collaborate freely, even with mathematicians whose skills they don't have confidence in. And soon, they'll understand the foundations of Mathematics very differently.*
>
> – Julie Rehmeyer, *Voevodsky's Mathematical Revolution*

In broad terms, Homotopy Type Theory is an attempt to formally redefine the whole mathematical behaviour somehow much closer to how informal Mathematics are actually done and to how Mathematics should be implemented to be checkable by a computer in an easy way. It is well-known that mathematical proofs are, in principle, already computationally checkable by means of a formalisation in an interactive theorem prover. In fact, we have already cited concrete examples of formal developments. Nevertheless, we have also shown that some of these instances, in which complex mathematical proofs are involved, have needed several years to be formalised using interactive theorem provers. These new foundations are expected to imply that the relationship between writing

a mathematical proof and checking it with a computer would be more natural and direct.

To sum up, a formal proof checked step by step manually is like to cover a very long distance on foot. Thanks to the current theorem provers, this way can be done like riding a bicycle. The new univalence foundations might make things go faster, like driving a car. The informal proof is more like a guide map where the steps are proposed, but they are not formally given.

Then, this thesis also includes something different from the formalisation of Linear Algebra: a more tentative chapter with a naive experiment where we have tried to formalise a small piece of Voevodsky's simplicial model for Homotopy Type Theory.

1.2 Contributions and Structure

The main topic of this thesis is the formalisation and execution of Linear Algebra algorithms. In more detail, the central contributions are listed below together with the chapter where each one of them is presented.

- First executable operations over matrices in the HOL Multivariate Analysis library, both using functions and immutable arrays. This provides a framework where algorithms over matrices can be formalised, executed, refined and coupled with their mathematical meaning (Chapter 3).
- The first formalisation in Isabelle/HOL of the Rank-Nullity theorem and the Fundamental Theorem of Linear Algebra (Chapter 4).
- A formalisation of the Gauss-Jordan algorithm as well as its applications, which allow computing ranks, determinants, inverses, dimensions and bases of the four fundamental subspaces of a matrix, and solutions of systems of linear equations (Chapter 4).
- A formalisation of the Gram-Schmidt process, the QR decomposition, and its application to the least squares problem (Chapter 4).
- Generalisation of some parts of the HOL Multivariate Analysis library of Isabelle/HOL (Chapter 4).
- A formalisation of the echelon form algorithm and its application to the computation of determinants and inverses of matrices over *Bézout domains* (Chapter 5).
- Enhancements of the HOL library about rings: implementation of *principal ideal domains*, *Bézout domains*, and other algebraic structures as well as their properties and relationships (Chapter 5).
- As far as we know, the first formalisation of the Hermite normal form of a matrix over *Bézout domains* and its uniqueness in any theorem prover (Chapter 5).
- The first formalisation about simplicial sets in Isabelle/HOL as well as some experiments in Isabelle/HOLZF about Voevodsky's simplicial model for Homotopy Type Theory (Chapter 6).

Most of the formalisations enumerated above have been published in the Archive of Formal Proofs (it is an online library, also known as AFP, of developments carried out in Isabelle). The only exception is the experiment related to Voevodsky's simplicial model, which has been published in [48, 49]. The developments sum up *ca.* 35000 Isabelle code lines. This number of lines includes the formalisations, examples of execution as well as documentation about the code. Although each one of such 35000 Isabelle code lines has been written by me, I feel it appropriate to value my Ph.D. supervisors' advice. Thus, this thesis is written in plural, that is, using *we* instead of *I*.

This thesis is structured as follows:

Chapter 1: Introduction.

Chapter 2: Preliminaries.

Chapter 3: Framework to Formalise, Execute, and Refine Linear Algebra Algorithms.

Chapter 4: Algorithms involving Matrices over Fields.

Chapter 5: Algorithms involving Matrices over Rings.

Chapter 6: Formalising in Isabelle/HOL a Simplicial Model for Homotopy Type Theory: a Naive Approach.

Chapter 7: Conclusions and Future Work.

Appendix A: Detailed List of Files and Benchmarks.

Chapter 2 presents both the mathematical and the interactive-proof machinery which have been necessary for this work. In Chapter 3 one of the main contributions of this thesis, at least in the sense that all the algorithms we have formalised are based on it, is presented: the framework that we have developed to formalise, execute, refine and link algorithms with their mathematical meaning. Following such an infrastructure, four Linear Algebra algorithms have been formalised. They are presented in two different chapters, depending on the algebraic structure of the elements of the matrices that are involved. The first kind of matrices we deal with are matrices over fields. We have formalised two algorithms, the Gauss-Jordan algorithm (over an arbitrary field) and the QR decomposition (for real matrices) which are presented in Chapter 4. Algorithms involving matrices over more general rings are presented in Chapter 5, concretely algorithms to compute the echelon form and Hermite normal form of a matrix. It is worth noting that each algorithm comes together with its own conclusions and related work, leaving to Chapter 7 the general conclusions and future work of this thesis. Chapter 6 shows an experiment on formalising the first definitions of Voevodsky's simplicial model for Homotopy Type Theory in Isabelle/HOL. The chapters are intended to be read in order, except for Chapter 6 which is independent from the rest of the thesis. A detailed enumeration of the Isabelle files that were developed for this work can be found in Appendix A as well as some benchmarks of the Gauss-Jordan algorithm and the QR decomposition. All of the benchmarks and execution tests presented throughout this thesis have been carried out in a laptop with an Intel® Core™ i5-3360M processor with 4GB of RAM, and Ubuntu GNU/Linux 14.04.

In addition, in each algorithm we will also show the formalisation of its corresponding applications, such as the computation of solutions of systems of linear equations and the least squares problem. We also provide some examples of real-world applications of the verified code obtained, such as the computation of the number of connected components of digital images (which is of interest in the study of the number of neurons' synapses) and the computation of determinants that some commercial software computes erroneously.

1.3 Publications

The formalisations which this work consists of have been published in the Archive of Formal Proofs (AFP). They are listed below. The chronological order of those AFP entries (in which they are given) corresponds closely to the section order in this thesis.

[52] Jose Divasón and Jesús Aransay. Rank-Nullity theorem in Linear Algebra. Archive of Formal Proofs, January 2013. `http://afp.sf.net/entries/Rank_Nullity_Theorem.shtml`, Formal proof development.

[54] Jose Divasón and Jesús Aransay. Gauss-Jordan Algorithm and Its Applications. Archive of Formal Proofs, September 2014. `http://afp.sf.net/entries/Gauss_Jordan.shtml`, Formal proof development.

[51] Jose Divasón and Jesús Aransay. QR Decomposition. Archive of Formal Proofs, February 2015. `http://afp.sf.net/entries/QR_Decomposition.shtml`, Formal proof development.

[50] Jose Divasón and Jesús Aransay. Echelon Form. Archive of Formal Proofs, February 2015. `http://afp.sf.net/entries/Echelon_Form.shtml`, Formal proof development.

[57] Jose Divasón and Jesús Aransay. Hermite Normal Form. Archive of Formal Proofs, July 2015. `http://afp.sf.net/entries/Hermite.shtml`, Formal proof development.

As we have already pointed out, the formalisation explained in Chapter 6 has not been published in the AFP yet. Nevertheless, all the Isabelle code written for such an experiment is accessible through [48, 49].

This thesis builds upon the following referred papers (ordered chronologically):

[11] Jesús Aransay and Jose Divasón. Performance Analysis of a Verified Linear Algebra Program in SML. In L. Fredlund and L. M. Castro, editors, *V Taller de Programación Funcional: TPF 2013*, pages 28 – 35, 2013.

[10] Jesús Aransay and Jose Divasón. Formalization and Execution of Linear Algebra: from Theorems to Algorithms. In G. Gupta and R. Peña, editors, *Preproceedings of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013*, pages 49 – 66. 2013.

[12] Jesús Aransay and Jose Divasón. Formalization and Execution of Linear Algebra: from Theorems to Algorithms. In G. Gupta and R. Peña, editors, *Postproceedings (Revised Selected Papers) of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013*, volume 8901 of LNCS, pages 01 – 19. Springer, 2014.

[14] Jesús Aransay and Jose Divasón. Generalizing a Mathematical Analysis Library in Isabelle/HOL. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 9058 of LNCS, pages 415–421. Springer, 2015.

[13] Jesús Aransay and Jose Divasón. Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. *Journal of Functional Programming*, 25, 2015.

[16] Jesús Aransay and Jose Divasón. Verified Computer Linear Algebra. Accepted for presentation at *the XV Spanish Meeting on Computer Algebra and Applications (EACA 2016)*, 2016.

[17] Jesús Aransay and Jose Divasón. Formalisation of the Computation of the Echelon Form of a matrix in Isabelle/HOL. Accepted for publication in *Formal Aspects of Computing*, 2016.

In addition, the following two draft papers are under revision process:

[15] Jesús Aransay and Jose Divasón. Proof Pearl - A formalisation in HOL of the Fundamental Theorem of Linear Algebra and its application to the solution of the least squares problem. Draft paper, 2015.

[18] Jesús Aransay, Jose Divasón, and Julio Rubio. Formalising in Isabelle/HOL a simplicial model for Homotopy Type Theory: a naive approach. Draft paper, 2016.

This thesis is not presented officially as a *compendium of publications*, however most of its chapters are built upon the articles presented above. Material from these publications has been reused with my supervisors' permission. More concretely, I have reused some parts of [12, 13] in order to develop Chapter 3. Chapter 4 is divided into four related parts: the Rank-Nullity theorem (based on [12, Sect. 2]), the Gauss-Jordan algorithm (based again on the articles [12, 13]), generalisations of the HOL Multivariate Analysis library (built upon [14]) and the QR decomposition (based on [15]). The echelon form algorithm has been described in Chapter 5 following our paper [17]. Furthermore, the Hermite normal form presented in such a chapter had never been published before. Chapter 6 is essentially based on the work presented in [18].

In addition, there exist another published papers which are related to this thesis, but they have not been presented as part of it:

[9] Jesús Aransay and Jose Divasón. Formalizing an Abstract Algebra Textbook in Isabelle/HOL. In J. R. Sendra and C. Villarino, editors, *Proceedings of the XIII Spanish Meeting on Computer Algebra and Applications (EACA 2012)*, pages 47–50. 2012.

[19] Jesús Aransay, Jose Divasón, Jónathan Heras, Laureano Lambán, María Vico Pascual, Ángel Luis Rubio, and Julio Rubio. Obtaining an ACL2 specification from an Isabelle/HOL theory. In G. A. Aranda-Corral, J. Calmet, and F. J. Martín-Mateos, editors, *Artificial Intelligence and Symbolic Computation - 12th International Conference, AISC 2014*. Proceedings, volume 8884 of Lecture Notes in Computer Science, pages 49–63, 2014.

1.4 Related Work

Although more detailed related work will be presented for each algorithm in the corresponding chapters, together with a comparison to ours, let us give here some broad strokes of it.

- **Coq Effective Algebra Library:** It is a set of libraries and commodities (also known as CoqEAL) developed by Cohen, Dénès, and Mörtberg [44] for the Coq proof assistant, where dependent types are allowed. The work presents an infrastructure over which algorithms involving matrices can be implemented, proved correct, refined, and finally executed. More concretely, they designed a methodology based on refinements which allows to prove the correctness of Linear Algebra algorithms and then refine them to efficient computation-oriented versions. Refinements can be performed both on the algorithms and on the data structures [39], data type refinements in CoqEAL are made in a *parametricity* fashion. In practice, the data type refinements in CoqEAL are reduced to using either functions or lists for representing vectors (and then matrices iteratively). In the CoqEAL framework computations are usually carried out over the Coq virtual machine (and thus "inside" the Coq system). It is a rich library, containing many formalisations of Linear Algebra algorithms, such as the Strassen's fast matrix product [143], Karatsuba's fast polynomial product [97], the Sasaki-Murao algorithm for efficiently computing the characteristic polynomial of a matrix [41], and an algorithm for computing the Smith normal form [34].

- **Jordan normal forms in Isabelle/HOL:** During the last year while developing this thesis, a new development about matrices in Isabelle/HOL was published: *Matrices, Jordan Normal Forms, and Spectral Radius Theory* by Thiemann and Yamada [148]. They have studied the growth rates of matrices in Jordan normal form. Their representation of matrices is slightly different (but related) from the one present in the HOL Multivariate Analysis library (which we will make use of), since they use a generic type to represent matrices of any dimension, whereas the one we use has a hardwired representation of the size in the matrices types. Their choice enables them to apply the algorithm to input matrices whose dimension is unknown in advance, one of their prerequisites. In fact, this new library for matrices admits to conveniently work with block matrices and it is also executable by a suitable setup of the Isabelle code generator. They refine algorithms to immutable arrays and reuse some of our serialisations. Their work is devoted to be applied to improve CeTA [146], their certifier to validate termination and complexity proof certificates.

Chapter 2

Preliminaries

Let us to lay the cards on the table to show the context which this thesis is based on, both the mathematical and the interactive-proof machinery. The chapter is organised as follows: Section 2.1 gives a brief introduction to the main mathematical theorems and notions which will play a central role in this thesis. Above all, we present concepts related to the manipulation of matrices and normal forms. In Section 2.2 we show the computer programs we have been working with, mainly Isabelle as well as some tools and facilities around such a theorem prover. In fact, all chapters of this thesis are concerned with formalising or implementing mathematical results and algorithms in Isabelle.

2.1 Mathematical Definitions and Theorems

Let us here introduce the main mathematical concepts which this thesis deals with. They will make up a mathematical basis for Chapters 3, 4, and 5. We let the introduction of some concrete concepts to their corresponding sections and chapters, due to they are quite specific to some parts of the thesis and they do not form a core to the whole work (specially in Chapter 6). Anyway, some of the following definitions and theorems will be revisited in their corresponding chapters, in order to see easily the relationship between the mathematical statements and the corresponding formalised results.

We suppose the reader to be familiar with Linear Algebra and algebraic structures. We have followed the references [22, 68, 115, 134, 142], where further details about the definitions and theorems can be found.

First of all, we should define some notation. By PIR (principal ideal ring) we mean a commutative ring with identity in which every ideal is principal (see Definition 17). We use PID (principal ideal domain) to mean a PIR which has no zero divisors. It is worth noting that some authors use PIR to refer to what we call PID, such as Newman [115]. Nevertheless, we consider that it is important to make the difference: for instance, the Hermite normal form, which will be presented later, is not a canonical form for left equivalence of matrices over a PIR, but it is over PIDs (see [140]). In the sequel, we assume that F is a field and R a commutative ring with a unit. We also focus our work on finite-dimensional vector spaces.

2.1.1 Introduction to Linear Maps

Let us revisit the relationship between linear maps and matrices, since this link plays a fundamental role in this thesis. We omit the proofs, but they can be found in [134].

Definition 1 (Linear map)**.** *Let V and W be vector spaces over a field F. A function $\tau : V \to W$ is a linear map if*

$$\tau(ru + sv) = r\tau(u) + s\tau(v)$$

for all scalars $r, s \in F$ and vectors $u, v \in V$. The set of all linear maps from V to W is denoted by $L(V, W)$.

Throughout this thesis, the application of a linear map τ on a vector v is denoted by $\tau(v)$ or by τv, parentheses being used when necessary or to improve readability.

Let $\{e_1, \ldots, e_n\}$ be the standard basis for F^n, that is, the ith standard vector has 0's in all coordinate positions except the ith, where it has a 1. Given any $m \times n$ matrix A over F the multiplication map $\tau_A(v) = Av$ is a linear map. In fact, any linear map $\tau \in L(F^n, F^m)$ has this form, that is, τ is just multiplication by a matrix, for we have

$$(\tau e_1 \mid \cdots \mid \tau e_n)e_i = (\tau e_1 \mid \cdots \mid \tau e_n)^{(i)} = \tau e_i$$

and so $\tau = \tau_A$ where $A = (\tau e_1 \mid \cdots \mid \tau e_n)$

Then, we have the following theorem, which corresponds to Theorem 2.10 in [134]. It states the existing link between linear maps and matrices.

Theorem 1 (Matrix of a linear map)**.**

1. *If A is an $m \times n$ matrix over F then $\tau_A \in L(F^n, F^m)$.*
2. *If $\tau \in L(F^n, F^m)$ then $\tau = \tau_A$, where*

$$A = (\tau e_1 \mid \cdots \mid \tau e_n)$$

The matrix A is called the matrix of the linear map τ.

Suppose that $B = (b_1, \ldots, b_n)$ and $C = (c_1, \ldots, c_n)$ are ordered bases for a finite-dimensional vector space V. Let $[v]_B$ be the coordinates of $v \in V$ for the basis B and $[v]_C$ the coordinates of $v \in V$ for the basis C. The coordinate vectors $[v]_B$ and $[v]_C$ are related by means of the following theorem.

Theorem 2 (Change of basis matrix)**.** *The change of basis matrix, also known as matrix of change of basis, from B to C is denoted as $M_{B,C}$ and it is*

$$M_{B,C} = ([b_1]_C \mid \cdots \mid [b_n]_C)$$

Hence

$$[v]_C = M_{B,C}[v]_B$$

and

$$M_{B,C}^{-1} = M_{C,B}$$

The theorem presented below states that any invertible matrix is indeed a matrix of change of basis.

Theorem 3. *If we are given any two of the following:*

1. *an invertible $n \times n$ matrix A;*
2. *an ordered basis B for F^n;*
3. *an ordered basis C for F^n;*

then the third is uniquely determined by the equation $A = M_{B,C}$.

Theorem 1 states that any linear map $\tau \in L(F^n, F^m)$ can be represented as a matrix. The following theorem states that we can indeed represent any linear map $\tau \in L(V, W)$ with respect to two ordered bases for V and W by means of a matrix (whenever V and W are finite-dimensional vector spaces).

Theorem 4. *Let $\tau \in L(V, W)$ and let $B = (b_1, \ldots, b_n)$ and $C = (c_1, \ldots, c_m)$ be ordered bases for V and W respectively. Then τ can be represented with respect to B and C as a matrix multiplication, that is,*

$$[\tau v]_C = [\tau]_{B,C}[v]_B$$

where $[\tau]_{B,C} = ([\tau b_1]_C \mid \cdots \mid [\tau b_n]_C)$ is called the matrix of τ with respect to the bases B and C.

Let us show now another two important theorems, which relate coordinates of a vector and change of basis matrices:

Theorem 5. *Let $\tau \in L(V, W)$ and let (B, C) and (B', C') be pairs of ordered bases of V and W respectively. Then,*

$$[\tau]_{B',C'} = M_{C,C'}[\tau]_{B,C}M_{B',B}$$

Theorem 6. *Let $\tau \in L(V, V)$ and let B and C be ordered bases for V. Then the matrix of τ with respect to C can be expressed in terms of the matrix of τ with respect to B as follows:*

$$[\tau]_C = M_{B,C}[\tau]_B M_{B,C}^{-1}$$

2.1.2 The Fundamental Theorem of Linear Algebra

Let us start introducing here the four fundamental subspaces of a matrix. From here on, by the notation $M_{n\times m}(F)$ we mean the set of all $n \times m$ matrices over a field F (and analogously, over $\mathbb{R}$, a ring R, and so on).

Definition 2 (The four fundamental subspaces)**.** *Given a matrix $A \in M_{n\times m}(F)$,*

- *The column space of A is $\{A\,y \mid y \in F^m\}$.*
- *The row space of A is $\{A^T y \mid y \in F^n\}$.*
- *The null space of A is $\{x \mid A\,x = 0\}$.*

- *The left null space of A is* $\{x \mid A^T x = 0\}$.

These four subspaces (usually named *four fundamental subspaces*) together share interesting properties about their dimensions and bases, that tightly connect them. These connections also provide valuable insight to study systems of linear equations $A\,x = b$, as we will show in Section 4.5.

Another interesting concept is the *inner product* of vectors, which indeed introduces a *geometrical* interpretation in $\mathbb{R}^n$ for the aforementioned subspaces and results. It is an algebraic operation ($\langle\cdot,\cdot\rangle : V \times V \to F$, for a vector space V over a field F, where F is either $\mathbb{R}$ or $\mathbb{C}$), which satisfies the following properties:

- $\langle x, y\rangle = \overline{\langle y, x\rangle}$, where $\overline{\langle\cdot,\cdot\rangle}$ denotes the conjugate;
- $\langle ax, y\rangle = a\langle x, y\rangle$, $\langle x + y, z\rangle = \langle x, z\rangle + \langle y, z\rangle$;
- $\langle x, x\rangle \geq 0$, $\langle x, x\rangle = 0 \Rightarrow x = 0$.

Note that in the particular case of the finite-dimensional vector space $\mathbb{R}^n$ over $\mathbb{R}$, the inner or *dot product* of two vectors $u, v \in \mathbb{R}^n$ is defined as $u \cdot v = \sum_{i=1}^{n} u_i v_i$. When $F = \mathbb{R}$, the *conjugate* is simply the identity.

Then, two vectors are said to be *orthogonal* when their inner product is 0 (which geometrically means that they are perpendicular). The *row space* and the *null space* of a given matrix are *orthogonal complements*, and so are the *column space* and the *left null space*. These results are brought together in an important result of Linear Algebra. In fact, some textbooks name it the *Fundamental Theorem of Linear Algebra*, see [142]. We present here its statement:

Theorem 7 (Fundamental Theorem of Linear Algebra)**.** *Let $A \in M_{n\times m}(F)$ be a matrix and $r = \operatorname{rank} A$; then, the following equalities hold:*

1. *The dimensions of the column space and the null space of A are equal to r and $m - r$ respectively.*
2. *The dimensions of the row space and the left null space of A are equal to r and $n - r$ respectively.*
3. *The row space and the null space are orthogonal complements.*
4. *The column space and the left null space are orthogonal complements.*

A complete formalisation of this theorem will be presented in Section 4.5. Let us stress that items 1 and 2 hold for $A \in M_{n\times m}(F)$, with F an arbitrary field, whereas items 3 and 4 hold for inner product spaces where either $F = \mathbb{R}$ or $F = \mathbb{C}$.

In addition, Item 1 in Theorem 7 is usually labelled as the *Rank-Nullity Theorem* and it is normally stated as follows:

Theorem 8 (The Rank-Nullity Theorem)**.** *Let $\tau \in L(V, W)$.*

$$\dim(\ker(\tau)) + \dim(\mathit{im}\ (\tau)) = \dim(V)$$

or, in other notation,

$$\mathit{rk}\ (\tau) + \mathit{null}\ (\tau) = \dim(V)$$

The statement presented above has been obtained from [134]. This part of the Fundamental Theorem of Linear Algebra will be crucial to compute, among other things, the dimension of the image of a linear map by means of the corresponding matrix associated to such a linear map. A formalisation of it will be presented in Section 4.2.

Furthermore, let f be $f\colon F^n \to F^m$ and $A \in M_{m\times n}(F)$ the matrix representing f with respect to suitable bases of F^n and F^m. Thanks to the existing link between matrices and linear maps, which has been presented in the previous subsection, the properties of A provide relevant information about f. For instance, computing the dimension of the *range* of f (or the *rank* or dimension of the *column space* of A), or the dimension of its *kernel* (or the *null space* of A) we can detect if f is either *injective* or *surjective*.

2.1.3 Matrix Transformations

This thesis presents the formalisation of some Linear Algebra algorithms, which transform matrices into different canonical forms (echelon form, reduced row echelon form, Hermite normal form) and decompositions (QR decomposition). Most of these transformations can be carried out by elementary row (column) operations. There are three types of elementary row (column) operations over a matrix $A \in M_{m\times n}(R)$. Let us remark that in this case we generalise the definition in order to work with matrices over a ring R (we do not restrict ourselves to a field F).

Definition 3 (Elementary operations)**.**

- Type 1. *Interchange of two rows (columns) of A.*
- Type 2. *Multiplication of a row (column) of A by a unit of R.*
- Type 3. *Addition of a scalar multiple of one row (column) of A to another row (column) of A.*

Definition 4 (Elementary matrix)**.** *If we perform an elementary operation of type k ($k \in \{1, 2, 3\}$) to an identity matrix, the result is called an elementary matrix of type k.*

Theorem 9. *All elementary matrices are invertible.*

It is worth noting that, in order to perform an elementary row operation of type k to a matrix $A \in M_{m\times n}(R)$, we can perform such an operation on the identity matrix I_m to obtain an elementary matrix P and then take the product PA. A similar multiplication on the right (starting from I_n) has the same effect of performing elementary column operations.

Definition 5 (Equivalent matrices)**.**

- *Two matrices A and B are equivalent if there exist invertible matrices P and Q for which*
$$B = PAQ^{-1}$$
- *Two matrices A and B are row equivalent if there exists an invertible matrix P for which*
$$B = PA$$

- *Two matrices A and B are column equivalent if there exists an invertible matrix Q for which*

$$B = AQ$$

Given a matrix, it can be transformed into another row (column) equivalent matrix by means of elementary operations. These transformations are very useful when they are applied properly, since they allow obtaining equivalent matrices which simplify the computation of, for instance, the inverse, determinant, decompositions such as LU and QR, and so on, of the original matrix.

Let us introduce the relationship between equivalent matrices, linear maps and change of basis matrices.

Theorem 10. *Let V and W be vector spaces with $\dim V = n$ and $\dim W = m$. Then two $m \times n$ matrices A and B are equivalent if and only if they represent the same linear map $\tau \in L(V, W)$, but possibly with respect to different ordered bases.*

In a straightforward way, we can define the concept of similar matrices:

Definition 6 (Similar matrices)**.** *Two matrices A and B are similar if there exists an invertible matrix P for which*

$$B = PAP^{-1}$$

Finally, we get the analogous version of Theorem 10 for square matrices:

Theorem 11. *Let V be a vector space of dimension n. Then two $n \times n$ matrices A and B are similar if and only if they represent the same linear map $\tau \in L(V, V)$, but possibly with respect to different ordered bases.*

Essentially, Theorems 10 and 11 represent the link between elementary transformations over matrices and their corresponding change of basis of linear maps. Thanks to this, as we have already said, elementary transformations allow us to obtain equivalent matrices which can simplify the computation of interesting properties of linear maps, such as the rank.

The most basic matrix canonical form (in the sense that many other canonical forms are based on it) that can be obtained using elementary operations is the echelon form.

Definition 7. *The* leading entry *of a nonzero row is its first nonzero element.*

Definition 8 (Echelon form)**.** *A matrix $A \in M_{m\times n}(R)$ is said to be in* echelon form *if:*

1. *All rows consisting only of 0's appear at the bottom of the matrix.*
2. *For any two consecutive nonzero rows, the leading entry of the lower row is to the right of the leading entry of the upper row.*

Note that a matrix in echelon form has many advantages from the manipulation point of view. For instance, it is upper triangular so it is straightforward to compute its determinant. An algorithm to compute the echelon form of a matrix is presented in Section 5.2.

Furthermore, the *reduced row echelon form* is another useful matrix canonical form, since it is the output of the Gauss-Jordan algorithm which is presented in Section 4.3.

Definition 9 (Reduced row echelon form)**.** *A matrix $A \in M_{m\times n}(F)$ is said to be in* reduced row echelon form *(or shorter, in* rref*) if:*

1. *A is in echelon form.*
2. *In any nonzero row, the leading entry is a 1.*
3. *Any column that contains a leading entry has 0's in all other positions.*

By means of elementary operations, any matrix over a PID can be transformed into an echelon form and any matrix over a field can be transformed into its reduced row echelon form, which is unique (see [140]). It is also a well-known result that over more general rings than fields it could be impossible to get the reduced row echelon form of a given matrix (leading entries different from 1 could appear).

There are many other kinds of canonical matrices which are based on the echelon form and present useful properties, such as the Hermite normal form. The Hermite normal form is the natural generalisation of the reduced row echelon form for PIDs, although it is normally studied just in the case of integer matrices. A primary use of such a normal form is to solve systems of linear diophantine equations over a PID [32]. Another example is the Smith normal form, which is useful in topology for computing the homology of a simplicial complex. The minimal echelon form and the Howell form are also examples of other canonical matrix forms (see [140] for detailed definitions).

It is worth noting that there is not a single definition of Hermite normal form in the literature. For instance, some authors [115] restrict their definitions to the case of square nonsingular matrices (that is, invertible matrices). Other authors [40] just work with integer matrices. Furthermore, given a matrix A its Hermite normal form H can be defined to be upper triangular [140] or lower triangular [115]. In addition, the transformation from A to H can be made by means of elementary row operations [115] or elementary column operations [40].

In our case, we have decided to work as general as possible, so we do not impose restrictions in the input matrix (thus, the case of nonsquare matrix is included and the coefficients can belong to a generic PID). We have decided to carry out the transformation from A to H by means of elementary row operations, obtaining H upper triangular. In fact, any algorithm or theorem using an alternative definition of Hermite normal form (for example, in terms of column operations and/or lower triangularity) can be easily moulded into the form of Definition 12.

Firstly, we have to define the concepts of *complete set of nonassociates* and *complete set of residues modulo μ*.

Definition 10 (Complete set of nonassociates)**.** *An element $a \in R$ is said to be an* associate *of an element $b \in R$, if there exists an invertible element $u \in R$ such that $a = ub$. This is an equivalence relationship over R. A set of elements of R, one from each equivalence class, is said to be a* complete set of nonassociates*.*

Definition 11 (Complete set of residues)**.** *Let μ be any nonzero element of R. Let a and b be elements in R. It is said that a is* congruent *to b modulo μ if μ divides $a - b$. This is an equivalence relationship over R. A set of elements of R, one from each equivalence class, is said to be a* complete set of residues modulo μ *(or a* complete set of residues of μ*).*

Definition 12 (Hermite normal form)**.** *Given a complete set of nonassociates and a complete set of residues, a matrix $H \in M_{m\times n}(R)$ is said to be in Hermite normal form if:*

1. *H is in echelon form.*
2. *The leading entry of a nonzero row belongs to the complete set of nonassociates.*
3. *Let h be the leading entry of a nonzero row. Then each element above h belongs to the corresponding complete set of residues of h.*

Definition 13 (Hermite normal form of a matrix)**.** *A matrix $H \in M_{m\times n}(R)$ is the Hermite normal form of a matrix $A \in M_{m\times n}(R)$ if:*

1. *There exists an invertible matrix P such that $A = PH$.*
2. *H is in Hermite normal form.*

Any matrix whose elements belong to a PID can be transformed by means of elementary operations to a matrix in Hermite normal form.

The complete sets of nonassociates and residues appear to define the Hermite normal form as general as possible. As we have already there is no one single definition of it in the literature, so some authors impose different conditions. In the particular case of integer matrices, leading coefficients (the first nonzero element of a nonzero row) are usually required to be positive, but it is also possible to impose them to be negative since we would only have to multiply by -1, since -1 is a unit in $\mathbb{Z}$.

In the case of the elements h_{ik} above a leading coefficient h_{ij} (they have to be residues modulo h_{ij}), some authors demand $0 \leq h_{ik} < h_{ij}$ (see [40]), other ones impose the conditions $h_{ik} \leq 0$ and $\mid h_{ik} \mid < h_{ij}$ (see [32]), and other ones $-\frac{h_{ij}}{2} < h_{ik} \leq \frac{h_{ij}}{2}$ (see [5]). More different options are also possible. All the possibilities can be represented selecting a complete set of nonassociates and a complete set of residues.

The following theorem states the uniqueness of the Hermite normal form of a nonsingular matrix, which corresponds to Theorem II.3 in [115].

Theorem 12. *If $A \in M_{n\times n}(R)$ is a nonsingular matrix, then its Hermite normal form is unique.*

We will show a formalisation of an algorithm to obtain the Hermite normal form of a matrix and its uniqueness in Section 5.3.

2.2 Isabelle

The main software that we have used in the development of our work is the Isabelle theorem prover. In addition, we have also taken advantage of some other well-known tools such as existing libraries and code generation facilities. Let us show a brief toolkit overview:

- Isabelle (Lawrence Paulson [124])
- Isabelle/Isar (Makarius Wenzel [151])

- Type Classes (Florian Haftmann [77])
- Locales (Clemens Ballarin [24])
- HOL Multivariate Analysis library (John Harrison [85])
- Code Generation (Florian Haftmann [78])

All of them will be superficially explained throughout this section, although we let the reader explore the references presented above for further details. Let us start explaining what Isabelle is.

2.2.1 Isabelle/HOL

Isabelle [124] is a generic theorem prover which has been instantiated to support different object-logics. It was originally created by Paulson and nowadays it is mainly developed at University of Cambridge by Paulson's group, at Technische Universität München by Nipkow's group, and by Wenzel, as well as it also includes numerous contributions from other institutions and individuals worldwide.

Its main application is the "formalisation of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols", see [6]. It is an LCF-style theorem prover (written in Standard ML [127]), so it is based on a small logical core to ease logical correctness.

The most widespread object-logic supported by Isabelle is higher-order logic (or briefly, HOL [118]). Isabelle's version of HOL (usually called Isabelle/HOL) corresponds to Church's simple type theory [38] extended with polymorphism, Haskell-style type classes, and type definitions. HOL allows nested function types and quantification over functions. HOL is a logic of total functions and its predicates are simply functions to the Boolean type (*bool*). HOL conventions are a mixture of mathematics and functional programming and it is usually introduced following the equation *HOL = Functional Programming + Logic*. It is by far the logic where the greatest number of tools (code generation, automatic proof procedures) are available and the one which most of developments are based on. These two reasons encourage us to carry out our development in Isabelle/HOL. However, it is worth noting that there exist other logics that have been implemented in Isabelle, such as Zermelo-Fraenkel set theory (whose Isabelle's version is known as Isabelle/ZF) and higher-order logic extended with ZF axioms (denoted as Isabelle/HOLZF). These logics will specially take on importance in Chapter 6.

Isabelle/HOL also includes powerful specification tools, e.g. for (co)datatypes, (co)inductive definitions and recursive functions with complex pattern matching. More concretely, the HOL type system is based on non-empty types, function types ($\Rightarrow$) and type constructors of different arities ($\_\ list$, $\_ \times \_$) that can be applied to already existing types (*nat, bool*) and type variables (α, β). The notation $t :: \tau$ means that the term t has type τ. Types can be also introduced by enumeration (*bool*) or by induction, as lists (by means of the **datatype** command). Additionally, new types can be also defined as non-empty subsets of already existing types (α) by means of the **typedef** command; the

command takes a set defined by comprehension over a given type $\{x :: \alpha \mid P\, x\}$, and defines a new type σ.

Isabelle incorporates some automatic methods and algebraic decision procedures which are used to simplify proofs and to automatically discard goals and *trivial* facts. For instance, the Isabelle's classical reasoner, which simulates a sequent calculus, can perform chains of reasoning steps to prove formulas and the *simplifier* can reason about equations. Let us note that, if it does not cause confusion, we usually write Isabelle when we mean Isabelle/HOL.

Isabelle also introduces type classes in a similar fashion to Haskell [77]; a type class is defined by a collection of operators (over a single type variable) and premises over them. For instance, the HOL library has a type class `field` representing the algebraic structure. Concrete types (`real`, `rat`) can be proven to be an **instance** of a given type class (`field` in our example). Type classes are also used to impose additional restrictions over type variables; for instance, the expression (`x :: 'a :: field`) imposes the constraint that the type variable `'a` possesses the structure and properties stated in the `field` type class, and can be later replaced exclusively by types which are instances of that type class.

Another interesting Isabelle's feature is *locales* [24], which are an approach for dealing with parametric theories. They are specially suitable to represent the complex inter-dependencies between structures found in Abstract Algebra, since they allow us to talk about carriers, sub-structures and existence of structures. However, they have proven fruitful also in other applications, such as software verification [99]. Locales enable to prove theorems abstractly, relative to sets of assumptions. Such theorems can then be used in other contexts where the assumptions themselves, or instances of the assumptions, are theorems. This form of theorem reuse is called **interpretation**. For instance, any theorem proven over vector spaces (within the **locale** `vector_space`) can be reused in real vector spaces (class `real_vector`), since `real_vector` is an interpretation of `vector_space`. The idea is similar to that of **instance** of a type class.

One of the most famous Isabelle's facilities is the *Intelligible semi-automated reasoning*, denoted as *Isar* [152]. Isar is an approach to get readable formal proof theories and it sets out to bridge the semantic gap between internal notions of proof given by Isabelle and an appropriate level of abstraction for user-level work. Isar is intended as a generic framework for developing formal mathematical documents with full proof checking and it works for all of the usual Isabelle object-logics.

Isabelle/HOL has been successfully used, for instance, in the proof of the Kepler conjecture [81] (the largest formal proof completed to date), in the formal verification of seL4 [99] (an operating-system kernel), and in the first machine-assisted formalisation of Gödel's second incompleteness theorem [128].

2.2.2 HOL Multivariate Analysis

The HOL Multivariate Analysis (or *HMA* for short) library [88] is a set of Isabelle/HOL theories which contains theoretical results in mathematical fields such as Analysis, Topology and Linear Algebra. It is based on the impressive work of Harrison in HOL Light [85], which includes proofs of intricate theorems (such as the Stone-Weierstrass theorem) and has been used as a basis for appealing projects such as the formalisation of the proof of the Kepler conjecture by Hales [82]. The translation of this library from HOL Light to Isabelle/HOL

is far from complete. It is mainly being done *by hand* and, apparently, translating HOL Light tactics and proofs to Isabelle is quite intricate.[1] Among others, Paulson, Hölzl, Eberl, and Immler are actively contributing to this translation, and also to extend the HMA library in other directions. The HMA library intensively uses the implementation of *type classes* to represent mathematical structures (such as semigroups, rings, fields and so on). We recommend the work by Hölzl, Immler, and Huffman [90] for a thorough description of the type classes appearing in the HMA library.

Among the fundamentals of the library, one of the keys is the representation of n-dimensional vectors over a given type `'a`. The idea (first presented by Harrison in [84]) is to represent n-dimensional vectors (type `vec`) over `'a` by means of *functions* from a finite type variable `'b :: finite` to `'a`, where `card ('b) = n` (the cardinal of a type can be interpreted as an abuse of notation; it really stands for the cardinal of the universe set of such a type). For proving purposes, this type definition is usually sufficient to support the generic structure R^n, where R is a ring. Note that the HOL family of provers, such as HOL Light and Isabelle/HOL, excludes dependent types, and consequently the possibility of defining n-dimensional vectors depending directly on a natural number, n.

The Isabelle `vec` type definition is as follows; vectors in finite dimensions are represented by means of *functions* from an underlying finite type to the type of the vector elements. The Isabelle constant `UNIV` denotes the set of every such a function. Indeed, **typedef** builds a new type as a subset of an already existing type (in this particular case, the set includes every function whose source type is finite). Elements of the newly introduced type and the original one can be *converted* by means of the introduced **morphisms**, in this case the functions `vec_nth` and `vec_lambda` are the morphisms between the abstract data type `vec` and the underlying concrete data type, functions with finite domain. The **notation** clause introduces an infix notation (`$`) for converting elements of type `vec` to functions and a binder χ that converts functions to elements of type `vec`.

```
typedef ('a ,'b) vec = "UNIV :: (('b::finite) ⇒ 'a) set"
 morphisms vec_nth vec_lambda ..
```

The previous type also admits in Isabelle the shorter notation `'a^'b`. Additional restrictions over `'a` and `'b` are added only when required for formalisation purposes. The idea of using underlying finite types for vectors indices has great advantages from the formalisation point of view, as already pointed out by Harrison. For instance, the type system can be used to guarantee that operations on vectors (such as addition) are only performed over vectors of equal dimension, *i.e.*, vectors whose indexing types are exactly the same (this would not be the case if we were to use, for instance, lists as vectors). Moreover, the functional flavour of operations and properties over vectors is kept (for instance, vector addition can be defined in a pointwise manner).

The representation of matrices is then derived in a natural way based on the representation of vectors by iterating the previous construction (matrices over a type `'a` will be terms of type `'a^'m^'n`, where `'m` and `'n` stand for finite type variables).

[1] See the messages in this email thread for some subjective estimations: `https://www.mail-archive.com/isabelle-dev@mailbroy.informatik.tu-muenchen.de/msg06184.html`.

A subject that has not been explored either in the Isabelle HMA library, or in HOL Light, is the possibility of executing the previous data types and operations. Another aspect that has not been explored in the HMA library is algorithmic Linear Algebra. One of the novelties of our work is to establish a link between the HMA library and a framework where algorithms can be represented and also executed (see Chapter 3).

Furthermore, the HMA library is focused on concrete types such as $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{R}^n$ and on algebraic structures such as real vector spaces and Euclidean spaces, represented by means of type classes. This limitation had been pointed out in some previous developments over this library (see, for instance [21]). The generalisation of the HMA library to more abstract algebraic structures (such as vector spaces in general and finite-dimensional vector spaces) is something desirable but it has not been tackled yet. In Section 4.4 we show how we have generalised part of the library in order to be able to execute some of our formalised algorithms involving matrices over arbitrary fields (for instance, $\mathbb{F}_p$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$).

2.2.3 Code Generation

Another interesting feature of Isabelle/HOL is its code generation facility [78]. Its starting point are specifications (in the form of the different kinds of definitions supported by the system) whose properties can be stated and proved, and (formalised) rewriting rules that express properties from the original specifications. From the previous *code equations*, a *shallow embedding* from Isabelle/HOL to an abstract intermediate functional language (Mini-Haskell) is performed. Finally, straightforward transformations to the functional languages SML, Haskell, Scala and OCaml are performed. Then, the code can be exported to such functional languages, obtaining programs and computations from verified algorithms. The expressiveness of HOL (such as for instance universal and existential quantifiers and the Hilbert's ϵ operator) is greater than that of functional programming languages, and therefore one must restrict herself to use Isabelle "executable" specifications, if she aims at generating code from them (or she must prove *code equations* that refine non-executable specifications to executable ones).

One weakness of this methodology is the different semantics among the source Isabelle constructs and their functional languages counterparts; this gap can be narrowed to a minimum, since the tool is based on a *partial correctness* principle. This means that whenever an expression v is evaluated to some term t, $t = v$ is derivable in the equational semantics of the intermediate language, see [80] for further details. Then, from the intermediate language, the code generation process can proceed to the functional languages by means of the aforementioned straightforward transformations, or, in a broadly accepted way of working [20, 62, 79], *ad-hoc* serialisations to types and operations in the functional languages library can be performed. These serialisations need to be *trusted*, and, therefore, they are kept as simple as possible (in Section 3.4 we explicitly introduce these transformations).

In this thesis, the approach to get verified code is to describe algorithms within the language of Isabelle/HOL, then extract code and run it independently. Furthermore, the existing code-generator infrastructure provides three different evaluation techniques *within* Isabelle, each one comprising different

aspects: expressiveness, efficiency and trustability. We summarise them here, further details can be obtained in [78]:

- **The simplifier (simp)**: The use of the simplifier together with the original code equations of the underlying program is the simplest way for evaluation. This allows fully symbolic evaluation as well as the highest trustablity, but with the cost of the usual (low) performance of the simplifier.
- **Normalization by evaluation (nbe):** it provides a comparably fast partially symbolic evaluation which permits also normalization of functions and uninterpreted symbols. The stack of code to be trusted is considerable.
- **Evaluation in ML (code):** The highest performance can be achieved by evaluation in ML, at the cost of being restricted to ground results and a layered stack of code to be trusted, including code generator configurations by the user (serialisations). Evaluation is carried out in a target language *Eval* which inherits from SML but for convenience uses parts of the Isabelle runtime environment. The performance of this evaluation is essentially the same as if code is exported to SML and run it independently. The soundness of the computations carried out depends crucially on the correctness of the code generator setup, this is why serialisations must be introduced carefully and kept as simple as possible.

2.2.4 Archive of Formal Proofs

The Archive of Formal Proofs, also known as AFP, is an online library of formalisations carried out in Isabelle and contributed by its users. It is organised like a scientific journal (in fact, each contribution is called an article) and submissions are refereed. Its aim is to be the main place for authors to publish their developments, being a resource of knowledge and formal proofs for users. The AFP was started in 2004 and nowadays it contains over 200 articles, including very different areas such as Graph Theory [119] and Rewriting [138]. Articles presented in the AFP slightly evolve throughout time to be up to date with the latest Isabelle version. One important point is that despite reusing libraries is something desirable, unfortunately it is not done as often as expected (see [30]). As we have already said in the previous chapter, the formalisation of the theorems and algorithms which are presented throughout this thesis have been published in the AFP. Thus, any other user of Isabelle can make use of them. Moreover, we have tried to reuse as many existing libraries as possible.

Chapter 3

Framework to Formalise, Execute, and Refine Linear Algebra Algorithms

3.1 Introduction

HMA [88] is a huge library which contains many theoretical results in mathematical fields such as Analysis, Topology and Linear Algebra. However, a subject that had not been explored either in the HMA library or in HOL Light is the possibility of obtaining an executable representation from its data types that represent vectors and matrices as well as the corresponding operations over them. For instance, matrix multiplication is defined in the HMA library, but it was not possible to compute it. Furthermore, the formalisation of Linear Algebra algorithms had not been explored in the HMA library: there is no implementation of common algorithms such as Gaussian elimination or diagonalisation.

In this chapter, we aim to show that we can provide a framework where algorithms over matrices can be formalised, executed, refined, and coupled with their mathematical meaning.

As we will show, the formalisation of Linear Algebra results, implementation of algorithms, and code generation to functional programs is achieved with an affordable effort within this framework, by using well-established tools. It also shows that the performance of the generated code is enough (even if it is not comparable to specialised programs in Computer Algebra) to obtain interesting computations (such as determinants with big integers that disclosed a bug in Mathematica® [59] and relevant properties of digital images, see Section 4.3). In addition, this part of the thesis shows that the ties between matrix algorithmics and Linear Algebra can be established thanks to the HMA library infrastructure (a property that is not possible in Computer Algebra systems).

The idea is to make executable the matrix representation presented in the HMA library, define algorithms over it, formalise their correctness, refine them to more efficient matrix representations, export verified code to functional languages (SML and Haskell), and connect algorithms with their mathematical

meaning. For the latter purpose, we provide a large library (see the file *Linear_Maps.thy* in [54]) about the existing relations between matrices and linear maps: for instance, the rank of a matrix is the dimension of the image of the linear map associated to such matrix, and it is preserved by elementary transformations since they correspond to a change of basis. In fact, an invertible matrix corresponds to a change of basis, which has also been proven. In Section 4.3 we will present some examples of this link between Linear Algebra and algorithmics. In this context we have also formalised the definitions of the four fundamental subspaces together with the properties among them.

Let us start to explain the approach. It is worth noting that most of Linear Algebra algorithms are based on the three types of elementary row/column transformations (see Definition 3). We have defined them in Isabelle as follows (we present the row version, the column operations are analogous):

```
definition interchange_rows :: "'a^'n^'m⇒'m⇒'m⇒'a^'n^'m"
  where "interchange_rows A a b = (χ i j. if i=a then A $ b $ j else if
i=b then A $ a $ j else A $ i $ j)"

definition mult_row :: "('a::times)^'n^'m⇒'m⇒'a⇒'a^'n^'m"
  where "mult_row A a q = (χ i j. if i=a then q*(A $ a $ j) else A $ i
$ j)"

definition row_add :: "('a::{plus,times})^'n^'m⇒'m⇒'m⇒'a⇒'a^'n^'m"
  where "row_add A a b q = (χ i j. if i=a then (A $ a $ j) + q*(A $ b $
j) else A $ i $ j)"
```

Apart from proving the expected properties of each operation, we have demonstrated that there exist invertible matrices which apply such elementary transformations (Theorem 9). For example, in the case of interchanging two rows:

```
lemma interchange_rows_mat_1:
  shows "interchange_rows (mat 1) a b ** A = interchange_rows A a b"

lemma invertible_interchange_rows:
 shows "invertible (interchange_rows (mat 1) a b)"
```

Let us note that `mat 1` is the implementation of the identity matrix in the HMA library. Thanks to the previous definitions, an algorithm based on them can be defined using the `vec` representation for matrices (see Subsection 2.2.2), proven its correctness inside the HMA library, and connected with the mathematical meaning thanks to the proven correspondence between linear maps and matrices. However, we would also like to execute the algorithm and here is where refinements come into play.

Data refinement [79] offers the possibility of replacing an abstract data type in an algorithm by a concrete type, Figure 3.1 shows how it works in general. The correctness of an algorithm should be proven using an abstract structure, that is, using the one where it is easier to formalise the specified properties. This abstract representation usually makes the formalisation easier, but also makes it too slow or even prevents the execution. Then, we want to get efficient

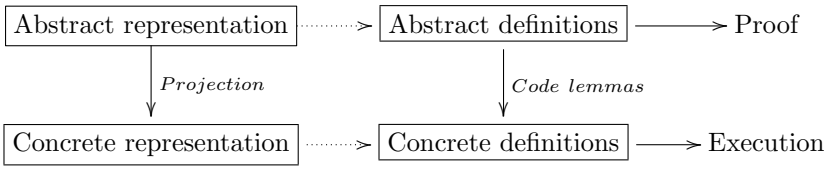


Figure 3.1: How a refinement works

computations. From such an abstract type, in our case the `vec` data type presented in the HMA library, a projection is defined and proven to another data type that admits a representation in a programming language. After that, the operations of the algorithm must be defined in the concrete representation as well, and these operations must be connected with the corresponding ones in the abstract representation by means of code lemmas. These code lemmas will *translate* the abstract (and possibly non-computable) operations to the concrete (and computable) ones. Then, execution can be carried out inside Isabelle or extracting code to functional programming languages such as SML and Haskell (see Subsection 2.2.3).

As we explained in Subsection 2.2.2, the `vec` type is an *abstract* type, produced as a subset of the concrete type of functions from a finite type to a variable type; this type cannot be directly mapped into an SML type, since its definition, a priori, could involve HOL logical operators unavailable in SML. In the code generation process, a data type refinement from the abstract to a concrete type must be defined; the concrete type is then the one chosen to appear in the target programming language. A similar refinement is carried out over the operations of the *abstract* type; definitions over the concrete data type (functions, in our case) have to be produced, and proved equivalent (*modulo* type morphisms) to the ones over the abstract type. The general idea is that formalisations have to be carried out over the abstract representation, whereas the concrete representations are exclusively used during the code generation process. The methodology admits iterative refinements, as long as their equivalence is proved. A detailed explanation of the methodology by Haftmann and Nipkow is found in [80]; an interesting case study by Esparza *et al.* in [62].

Let us focus on our framework; in our case `vec` is itself an *abstract* type which is non-executable and also has to be *refined* to *concrete* data types that can be code generated. We present here two such refinements. The first one consists in refining the abstract type `vec` to its underlying concrete type *functions (with finite domain)*. We expected the performance to be unimpressive, but the close gap between both types greatly simplifies the refinement; interestingly, at a low cost, executable versions of algorithms can be achieved, capable of being computed over matrices of small sizes. The second data type refinement is more informative; we refine the `vec` data type to the Isabelle type `iarray`, representing *immutable arrays* (which are generated in SML to the *Vector* structure [67] and to *IArray.array* in Haskell [4], as it is explained in Section 3.4).

In order to do that, we define both the basic operations involving matrices (addition, multiplication, manipulation of rows/columns, ...) and the elementary transformations using functions and also immutable arrays. Then, we will prove the equivalence between these new executable definitions and the corresponding non-executable ones of the `vec` representation.

This framework will be strongly reused in the formalisation of the Gauss-Jordan algorithm (Section 4.3), the QR decomposition (Section 4.5), the echelon form algorithm (Section 5.2) and the Hermite normal form (Section 5.3).

3.2 The Natural Refinement: from *vec* to Functions over Finite Types

In this section we show how we have carried out the refinement from the abstract type `vec` to its underlying concrete type, functions with finite domain. We present it in two steps:

1. Code generation from finite types which represent the indexes of the rows/columns of a matrix (Subsection 3.2.1).
2. Data type refinement from `vec` to functions over finite types (Subsection 3.2.2).

The second one depends on the first one: first, we need to be able to execute and work with the indexes, and after that, achieve the execution of the matrix representation which makes use of them.

3.2.1 Code Generation from Finite Types

As we have already said, our work is based on the HMA library and thus, we have used in our development an *abstract* data type `vec` (and its iteration for representing matrices), for which the underlying *concrete* types are functions with an indexing type.

The indexing type is instance of the `finite`, `enum` and `mod_type` type classes. These classes demand the universe of the underlying type to be finite, to have an explicit enumeration of the universe, and some arithmetical properties. Let us explain why we have demanded the finite types to be instances of such classes.

The `finite` type class is enough to generate code from some abstract data structures, such as *finite sets*, which are later mapped into the target programming language (for instance, SML) to data structures such as lists or red black trees (see the work by Lochbihler [108] for details and benchmarks). Our case study (Linear Algebra algorithms) is a bit more demanding, since the indexing types of vectors and matrices usually have to be also enumerable. The `enum` type class allows us to *execute* operations such as matrix multiplication, $A * B$ (as long as the type of columns in A is the same as the type of rows in B), algorithms traversing the universe of the rows or columns indexing types (such as operations that involve the logical operators $\forall$ or $\exists$ or the Hilbert's ϵ operator), enabling operators like "every element in a row is equal to zero" or "select the least position in a row whose element is not zero".

In many Linear Algebra algorithms, the proof of correctness is performed by induction over column or row indices, so they must be inductive. Thus, we make use of an additional type class `mod_type`, which resembles the structure $\mathbb{Z}/n\mathbb{Z}$, together with some required arithmetic operations and conversion functions from it to the integers (`to_nat` and `from_nat`). We have implemented them in Isabelle as follows:

```
class mod_type = times + wellorder + neg_numeral +
fixes Rep :: "'a ⇒ int"
  and Abs :: "int ⇒ 'a"
  assumes type: "type_definition Rep Abs {0..<int CARD ('a)}"
  and size1: "1 < int CARD ('a)"
  and zero_def: "0 = Abs 0"
  and one_def:  "1 = Abs 1"
  and add_def:  "x + y = Abs ((Rep x + Rep y) mod (int CARD ('a)))"
  and mult_def: "x * y = Abs ((Rep x * Rep y) mod (int CARD ('a)))"
  and diff_def: "x - y = Abs ((Rep x - Rep y) mod (int CARD ('a)))"
  and minus_def: "- x = Abs ((- Rep x) mod (int CARD ('a)))"
  and strict_mono_Rep: "strict_mono Rep"
```

Therefore, if one pursues execution the underlying types used for representing the rows and columns of the input matrices must be instances of the type classes `finite`, `enum` and `mod_type`. However, to state an algorithm just the type class `mod_type` is necessary, as it can be noted in the following Isabelle definition of the Gauss-Jordan algorithm:

```
definition Gauss_Jordan::"'a::{inverse, uminus,
  semiring_1}^columns::{mod_type}^rows::{mod_type} ⇒ 'a^columns^rows"
 where ...
```

In the previous algorithm definition we exclusively included the type classes required to *specify* the algorithm; in the later proof of the algorithm, we have to restrict `'a` to be an instance of the type class `field`; additionally, if we try to *execute* the algorithm (or generate code from it), the rows and columns types need to be instances of `enum`. The `finite` type class is implicit in the rows and columns types, since `mod_type` is a subclass of it.

The standard setup of the Isabelle code generator for (finite) sets is designed to work with sets of generic types (for instance, sets of natural numbers), mapping them to *lists* on the target programming language. This poses some restrictions, since operations such as *coset* $\emptyset$ cannot be computed over arbitrary finite types, whereas in an *enumerable* type *coset* $\emptyset$ is equal to a set containing every element of the enumerable type (and therefore, in the target programming language, the result of the previous operation will produce a list containing every element in the corresponding type). The particular setup enabling this kind of calculations (only for enumerable types), which is *ad-hoc* for our framework, can be found in the file *Code_Set.thy* of our development [54].

Another different but related issue is the election of a concrete type to be used as index of vectors and matrices; we already know that the type has to be an instance of the type classes `finite`, `enum` and `mod_type`. The Isabelle library contains an implementation of *numeral types* used to represent finite types of any cardinality. It is based on the binary representation of natural numbers (by means of the two type constructors, `bit0` and `bit1`, applied to underlying finite types, and of a singleton type constructor `num1`).

```
typedef 'a bit0 = "{0...<2 * CARD('a::finite)}"
typedef 'a bit1 = "{0...<1 + 2 * CARD('a::finite)}"
```

From the previous constructors, an Isabelle type representing $\mathbb{Z}/5\mathbb{Z}$ (or 5 in Isabelle notation) can be used, which is internally represented as `bit1 (bit0 (num1))`. The representation of the (abstract) type 5 is the set $\{0, 1, 2, 3, 4 :: 5\}$; its concrete representation is the subset $\{0, 1, 2, 3, 4 :: int\}$. The integers as underlying type allow users to reuse (with adequate modifications) integer operations (subtraction and unary minus) in the resulting finite types. As part of our development, we prove that the `num1`, `bit0` and `bit1` type constructors are instances of the `enum` and `mod_type` type classes.

```
instantiation bit0 :: enum
begin
definition "(enum::'a bit0 list) =
     map (Abs_bit0'∘ int) (upt 0 (CARD 'a bit0))"
definition "enum_all P = (∀ b ∈ enum. P b)"
definition "enum_ex P = (∃ b ∈ enum. P b)"
instance proof (intro_classes) ...
```

The Isabelle library already provides basic arithmetic functions for numeral types, with definitions of addition, subtraction, multiplication and division. Note that, for these operations to be defined generally for arbitrary cardinalities, the cardinality of the finite type must be *computed* on demand (adding 3 and 4 in type 5 must return 2). To this aim, the Isabelle library has a type class (`card_UNIV`) for types whose cardinality is *computable*; we prove that the previous numeral types are instances of such a **class**, enabling the computation of their cardinals. These proofs have been included as part of the official Isabelle library.

3.2.2 From *vec* to Functions over Finite Types

Up to now, we have been able to obtain executable representations (that can be implemented in programming languages) for finite types which represent the indexes of rows and columns of matrices. Now, we want to obtain a representation for the `vec` data type that can be implemented in a programming language: functions over finite types. In order to achieve it, the type morphisms between the type `vec` and its counterpart (functions) have to be labelled precisely in the code generator setup.

```
lemma [code_abstype]: "vec_lambda (vec_nth v) = (v::'a^'b::finite)"
```

Additionally, every operation over the abstract data type has to be *mapped* into an operation over the concrete data type (and their behavioural equivalence proved). It can be noted that because of the iterative construction of matrices (as elements of type `vec` over `vec`) each operation over matrices (as multiplication below) usually demands two lemmas to translate it to its computable version. The reason for this is that the code generator does not support nested abstract datatypes directly, thus a trick to get code execution is needed: redefine the operations in terms of the behaviour in each row. It is also remarkable that `setsum` is *computable* as long as there is an explicit version of the `UNIV` set, and this holds since we have restricted ourselves to `enum` types.

```
definition "mat_mult_row m m' f =
```

```
 vec_lambda(λj. setsum (λi.(m$f$i * m'$i$j)) UNIV)"
lemma [code abstract]: "vec_nth (mat_mult_row m m' f) =
 vec_lambda (λj. setsum (λi.(m$f$i * m'$i$j)) UNIV)"
lemma [code abstract]: "vec_nth (m ** m') = mat_mult_row m m'"
```

We have developed a refinement where the main operations involving vectors and matrices as well as the elementary row/column transformations can be transformed from the abstract type `vec` to computable versions using functions over finite types. This formalisation is presented in the file *Code_Matrix.thy* of [54]. As long as our algorithms are based on (abstract) operations which are mapped into corresponding concrete operations, the later ones will be correctly code generated.

Since dealing with matrices as functions can become rather cumbersome, we also define additional functions (such as `list_of_list_to_matrix` below) for conversion between lists of lists and functions (so that the input and output of the algorithm are presented to the user as lists of lists).

One subtlety appears at this step; from a given list of elements, a vector of a certain dimension is to be produced. In a logic such as HOL, where dependent types are not available, the user must add a type annotation declaring which dimension the generated vector has to be (in other words, the size of the list needs to be known in advance).

Below we present examples of the evaluation (by means of SML generated code) of a couple of basic operations (matrix multiplication and interchange rows). The evaluation can be also performed *in* Isabelle (and therefore the code generator would not intervene):

```
value[code] "let A = list_of_list_to_matrix
  [[0,0,0],[0,0,1],[2,3,4]]::rat^3^3
  in matrix_to_list_of_list (A**A)"

value[code] "let A = list_of_list_to_matrix
  [[0,0,0],[0,0,1],[2,3,4]]::rat^3^3
  in matrix_to_list_of_list (interchange_rows A 0 1)"
```

Their corresponding outputs are respectively:

```
[[0, 0, 0], [2, 3, 4], [8, 12, 19]] ::  rat list list
[[0, 0, 1], [0, 0, 0], [2, 3, 4]] ::  rat list list
```

We would also like to show how execution involving the indexes of the rows and columns of a matrix works. Many Linear Algebra algorithms require operators based on indexes (positions) of the matrices, which are represented by finite types within this matrix representation. For instance, it is quite common to "select the least position in a row whose element is not zero". The fact of being able to use such kind of operators in an algorithm is a great advantage from the formalisation point of view. Indeed now these statements are also executable in our approach since `num1`, `bit0` and `bit1` are proven to be instances of `enum`.

As an example, we show how we can obtain the least (the first) nonzero row of a matrix:

```
value[code] "let A = list_of_list_to_matrix
  [[0,0,0],
   [0,0,1],
   [2,3,4]]::rat^3^3
  in (LEAST n. row n A ≠ 0)"
```

The result is `1`, since row and column indexes start in 0.

The previous computations have been carried out with matrices represented as functions. The main advantage of this refinement is that it is quite straightforward to carry out (proofs are almost immediate) and thus it is easy to get execution from the abstract type `vec`. Unfortunately, the performance obtained makes algorithms based on this representation unusable in practice, except for testing purposes like the examples presented above. For instante, the computation of the Gauss-Jordan algorithm, which will be presented later, over matrices of size 15×15 is already very slow (several minutes).

More concretely, there are two sources of inefficiency in the results obtained. First, Isabelle is not designed as a programming language (it is an interactive theorem prover), and execution inside of the system offers poor performance. In Section 3.4 we present a solution to *translate* our specifications to functional programming languages. Second, the data structures (functions) are optimal for formalisation, but not for execution. Section 3.3 describes a *verified refinement* between the type used for representing matrices in our formalisation (`vec` and its iterated construction) and *immutable arrays*, a common data structure in functional programming.

3.3 Looking for a Better Performance: from *vec* to Immutable Arrays

Some data types present better properties for specification and formalisation purposes. For instance, specifying an algorithm over sets could be easier than doing so over lists. However, the latter data type is better suited for execution tests. Following this idea, the poor performance presented by functions representing matrices can be solved by means of a *data refinement* to a better performing data structure. Our intention is to replace the `vec` type representing vectors before applying code generation. In our development, we have used the Isabelle type `iarray` as the target type of our refinement. As it will be presented in Section 3.4, the Isabelle `iarray` data type will be serialised to *Vector.vector* in SML and *IArray.array* in Haskell. Both *Vector.vector* and *IArray.array* are the implementations in the target languages for immutable arrays, which are immutable sequences with constant-time access. The latter feature is the one that will allow us to achieve a better performance, improving an implementation with lists.

One of the first things that we have to do is to define the addition of immutable arrays, that is, to prove `iarray` to be an instance of the `plus` class. We present two different possibilities:

- **First possibility:**

```
instantiation iarray :: (plus) plus
begin
definition plus_iarray :: "'a iarray ⇒ 'a iarray ⇒ 'a iarray"
  where "plus_iarray A B =  IArray.of_fun (λn. A!!n + B !! n)
(IArray.length A)"
instance
proof
qed

end
```

- **Second possibility:**

```
instantiation iarray :: ("{plus,zero}") plus
begin

definition plus_iarray :: "'a iarray ⇒ 'a iarray ⇒ 'a iarray"
  where "plus_iarray A B =
  (let length_A = (IArray.length A);
  length_B= (IArray.length B);
  n=max length_A length_B ;
  A'= IArray.of_fun (λa. if a < length_A then A!!a else 0) n;
  B'=IArray.of_fun (λa. if a < length_B then B!!a else 0) n
  in
  IArray.of_fun (λa. A' !! a + B' !! a) n)"

instance
proof
qed

end
```

The first option just adds two *iarrays* componentwise without taking care of their length (which indeed is not necessary in our refinement, since when coming from `vec`, we will just want to add *iarrays* of the same length). This definition will be used in the Gauss-Jordan algorithm (Section 4.3), the echelon form algorithm (Section 5.2) and the Hermite normal form (Section 5.3).

In the second one, the addition of *iarrays* is done up to the length of the shortest vector and it is completed with zeros up to the length of the longest vector. This is less efficient, but on the other hand it allows us to prove the **datatype** constructor `iarray` to be an instance of `comm_monoid_add` (to be a commutative monoid), which is quite useful for some algorithms. For instance, we make use of this definition in the QR decomposition (Section 4.5), where we are able to directly execute *setsums* involving immutable arrays (which is not possible using the first option, since *setsums* are only defined over types which are instance of the `comm_monoid_add` class).

Both options are just alternative definitions of addition of immutable arrays. The user should decide which definition to use depending on the algorithm to formalise.

On another note, we have to define functions *vec_to_iarray* that convert elements of type *vec* to elements of type *iarray* (an operation *matrix_to_iarray* will convert iterated vectors to iterated iarrays, the natural representation for matrices in this setting).

```
definition vec_to_iarray::"'a^'col::{mod_type} ⇒ 'a iarray"
where "vec_to_iarray v = IArray.of_fun (λi.v$(from_nat i)) (CARD('col))"
```

Each function over elements of type *vec* needs to be replaced by a new function over type *iarray*. This requires first specifying a function over the type *iarray*, and then proving that it behaves as the one over type *vec*. The following result is labelled as a *code equation*, that will be later used in the code generation process.

```
lemma [code-unfold]:
fixes A::"'a::{semiring-1}^'col::{mod_type}^'row::{mod_type}"
shows "matrix_to_iarray (interchange_rows A i j) =
  interchange_rows_iarray (matrix_to_iarray A) (to_nat i)(to_nat j)"
```

The code equation certifies that it is correct to replace the function *interchange_rows* (defined over abstract matrices, or elements of type *vec*) by the function *interchange_rows_iarrays*. As it can be observed, the code equation does not include premises; this is a requirement from the Isabelle code generator [78]. The label *code_unfold* instructs the code generation tool to record the lemma as a rewriting rule, replacing occurrences of the left-hand side in the execution and code generation processes by the right-hand side. From a broader perspective, the function *matrix_to_iarray* has to be proved to be a morphism between the original and the refined type.

Again, we have defined and proved a framework to operations that can be used in most of Linear Algebra algorithms. In this framework there have been included both basic operations over vectors/matrices and the well-known elementary row/column operations over matrices. Among others, the following ones have been defined over the *iarray* structure and proved the equivalence with their corresponding definitions involving *vec*:

- Vector addition
- Vector subtraction
- Pointwise vector multiplication
- Multiply a vector by a scalar
- Scalar product of vectors (inner product)
- Obtain a row/column of a matrix
- Obtain the set of all rows/columns of a matrix
- Matrix addition
- Matrix subtraction

- Matrix multiplication
- Multiply a matrix by a vector on the left/right
- Identity matrix
- Interchange two rows/columns of a matrix
- Multiply a row/column of a matrix by a constant
- Add to a row/column of a matrix another row/column multiplied by a constant
- ...

The equivalence proofs are almost straightforward, since the `iarray` and `vec` representations share a *functional* flavour (in the way of accessing elements) that can be exploited in proofs. This effort pays off in terms of reusability, since the lemmas proving the equivalence between abstract and concrete operations for such operations can be reused in implementations of different algorithms over matrices.

Mapping some other operations on the type `vec` to the type `iarray` may involve programming decisions. For instance, the `LEAST` operator, which in its Isabelle original definition merely has a logical specification (a description of the properties that the least element satisfying a predicate possesses), needs to be proved equivalent to a user provided operation on the data type `iarray`. The definition below shows an operation on the type `iarray`, which is itself based on the operation *find* of lists. The Isabelle corollary presented below proves that this operation successfully finds the *least* (with respect to the indexing order) nonzero element in a matrix column j over a row i (if such an element exists, as expressed in the `assumes` clause).

```
definition "least_nonzero_position_index v i =
  the (List.find (λx. v!!x ≠ 0) [i..<(IArray.length v)])"

corollary
 fixes A::"'a::zero^'col::mod_type^'row::mod_type" and "j::'col"
 defines "colj ≡ vec_to_iarray (column j A)"
 assumes "¬(vector_allzero_from_index (to_nat i, colj))"
 shows "least_nonzero_position_index (colj) (to_nat i) =
  to_nat (LEAST n. A$n$j ≠ 0 ∧ i ≤ n)"
```

The formalisation of the infrastructure presented in this section is available from the file *Matrix_To_IArray.thy* in [54].

3.4 Serialisations to SML and Haskell Native Structures

Since the Isabelle code is not suitable for computing purposes, the original Isabelle specifications are *translated* to a programming (functional) language, as introduced in Subsection 2.2.3. Our choices (from the available languages in the standard Isabelle code generation setup) were SML (since the SML Standard

Library includes a *Vector* type representing immutable arrays) and Haskell (for a similar reason, with the Haskell *IArray* class type and its corresponding instance *IArray.Array*, and also because there is a built-in *Rational* type representing arbitrary precision rational numbers).

Additionally, we make use of *serialisations*, a process to map Isabelle types and operations to the corresponding ones in the target languages. Serialisations are a common practice in the code generation process (see the tutorial by Haftmann [78] for some introductory examples); otherwise, the Isabelle types and operations would be *generated* from scratch in the target languages, and the obtained code would be less readable and efficient (for instance, the `nat` type would be generated to an *ad-hoc* type with 0 and *Suc* as constructors, instead of using the built-in representation of integers in the target language). As we show below, and for the sake of correctness, serialisations are usually kept to a minimum, serialising Isabelle constants and operations (`0::nat`, `op +`, access operations) to the same ones in the target languages. The advantage of applying serialisations to the target languages suitably is stressed by an empirical result; profiling the computations carried out over matrices of rational numbers in SML, we detected that the greatest amount of time was spent in normalising fractions (operations *gcd* and *divmod*). Serialising these Isabelle operations to the corresponding built-in Poly/ML [130] and MLton [113] functions (which are not part of the SML Standard Library, but particular to each compiler), decreased the running time by a factor of 20.

We have carried out serialisations from Isabelle/HOL to SML and Haskell of:

- Immutable arrays (the efficient type used to represent vectors and matrices)
- $\mathbb{Z}_2$, $\mathbb{Q}$ and $\mathbb{R}$ numbers (the types of the coefficients of the matrices)

The following Isabelle code snippet presents the serialisation that we produced from the Isabelle type `rat` representing rational numbers (which is indeed based on equivalence classes), to the Haskell type *Prelude.Rational*. As it can be observed, it merely identifies operations (including type constructors) from the source and the target languages.

```
code-printing
type-constructor rat ⇀ (Haskell) "Prelude.Rational"
 | class-instance rat :: "HOL.equal" ⇒ (Haskell) -
 | constant "0 :: rat" ⇀ (Haskell) "Prelude.toRational (0::Integer)"
 | constant "1 :: rat" ⇀ (Haskell) "Prelude.toRational (1::Integer)"
 | constant "Frct-integer" ⇀ (Haskell) "Rational.fract (_)"
 | constant "numerator-integer ⇀  (Haskell) "Rational.numerator(_)
 | constant "denominator-integer ⇀  (Haskell) "Rational.denominator(_)
 | constant "HOL.equal :: rat ⇒ rat ⇒ bool" ⇀ (Haskell) "(_) == (_)"
 | constant "op < :: rat ⇒ rat ⇒ bool" ⇀ (Haskell) "_ < _"
 | constant "op ≤ :: rat ⇒ rat ⇒ bool" ⇀ (Haskell) "_ <= _"
 | constant "op + :: rat ⇒ rat ⇒ rat" ⇀ (Haskell) "(_) + (_)"
 | constant "op - :: rat ⇒ rat ⇒ rat" ⇀ (Haskell) "(_) - (_)"
 | constant "op * :: rat ⇒ rat ⇒ rat" ⇀ (Haskell) "(_) * (_)"
 | constant "op / :: rat ⇒ rat ⇒ rat" ⇀ (Haskell) " (_) '/ (_)"
 | constant "uminus :: rat ⇒ rat" ⇀ (Haskell) "Prelude.negate"
```

The complete set of Isabelle serialisations that we used is shown in Table 3.1. The Isabelle types `rat`, `real` and `bit` represent respectively $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{Z}_2$. The SML type *IntInf.int* represents arbitrarily large integers. It is worth noting that the Isabelle type `real`, internally constructed as equivalence classes of Cauchy sequences over the rational numbers, can be also serialised to the types used for `rat` in SML and Haskell, preserving arbitrary precision and avoiding numerical stability issues. Types presented in bold face identify serialisations that were introduced by us as part of this thesis.

The *Vector* structure in SML defines polymorphic vectors, immutable sequences with constant-time access. The serialisation of the Isabelle datatype `iarray` to *Vector.vector* in SML was already part of the Isabelle library. We also contributed some improvements to such an Isabelle library in the serialisation to the SML type *Vector.vector*, such as the serialisation of the Isabelle functions `IArray.all` and `IArray.exists` (given a function `f :: 'a ⇒ bool` and an immutable array `A :: 'a iarray`, they check if $\forall a \in A. fa = True$ and if $\exists a \in A. fa = True$ respectively) to the corresponding *Vector.exists* and *Vector.all* presented in the SML library.

In the case of Haskell, we have serialised the `iarray` Isabelle datatype to the *Data.Array.IArray.array* (or shorter, *IArray.array*) constructor presented in the standard Haskell's library. It is worth noting that there exist other implementations of immutable arrays in Haskell, such as *UArrays* (*Data.Array.Unboxed.array*). According to the Haskell library, "a *UArray* will generally be more efficient (in terms of both time and space) than the equivalent *Array* with the same element type, but *UArray* is strict in its elements". However, in the case of the code generated from our Isabelle/HOL developments, we have empirically tested that *IArray.array* performs slightly better than if the *Unboxed.array* constructor is used. As an example, the computation of the determinant of a 1500×1500 $\mathbb{Z}_2$ matrix by means of the code generated to Haskell from the verified Gauss-Jordan algorithm that will be presented in Section 4.3 takes 6.09 *secs* using *Data.Array.IArray.array* and 6.37 *secs* using *Data.Array.Unboxed.array*. Nevertheless, if one specifically wants to make use of *UArrays*, the file which serialises the corresponding Isabelle structure to them is presented in [47].

Furthermore, a more specific Haskell module for immutable arrays is *Data.Array*. The *Data.Array.IArray* module, the one that we use, provides a more general interface to immutable arrays: it defines operations with the same names as those defined in *Data.Array* (where the *Data.Array.array* constructor is involved), but with more general types, and also defines instances of the relevant classes. We have also empirically checked if there is a substantial difference between using *Data.Array.IArray.array* and *Data.Array.array*. As in the case of unboxed immutable arrays, the use of *Data.Array.array* does not suppose an advantage in terms of performance.

On another note, the SML Standard Library lacks of a type representing arbitrary precision rational numbers, and thus the proposed serialisation for `rat` is quotients of arbitrarily large integers. In this particular case, Haskell takes advantage of its native *Rational* type to challenge SML performance, which otherwise tends to be poorer.

We also explored the serialisation of Isabelle type `real` to double-precision floating-point formats (*Double* in Haskell, *Real.real* in SML) in the target lan-

Isabelle/HOL	SML	Haskell
`iarray`	*Vector.vector*	***IArray.Array***
`rat`	*IntInf.int / IntInf.int*	***Rational***
`real`	*Real.real*	***Double***
`bit`	***Bool.bool***	***Bool***

Table 3.1: Type serialisations

guages, but the computations performed present the expected *round-off errors*. This means that, although exported code has been generated from verified algorithms, computations cannot be trusted since they make use of floating-point numbers.

The `bit` type, which will be used to represent $\mathbb{Z}_2$ admits multiple serialisations, ranging from boolean values to subsets of the integers (with either fixed or arbitrary size). Experimental results showed us that the best performing option was to serialise `bit` and its operations to the corresponding boolean types in the target languages.

The previous serialisations are carried out in order to improve the performance of the exported code in the target languages (SML and Haskell). They can be reused, indeed they have already been reused, in other non-related developments. For instance, our Haskell serialisation for immutable arrays was reused in a development by Thiemann and Yamada [148]. Their matrix representation in Isabelle is different to the one that we are using (the one presented in the HMA library): they define a new datatype for matrices which is also based on immutable arrays. Then, they reuse the serialisation presented in our work to connect the immutable arrays in Isabelle (`iarray`) to the corresponding structure in Haskell (*IArray.array*).

3.5 Functions *vs.* Immutable Arrays *vs.* Lists

Previous sections have shown two refinements that we have developed in order to get execution from the `vec` data type. However, there are some questions that remain: how good is the performance with each representation? Is it really worth using immutable arrays? How about using lists? In this section we try to answer such questions.

We have carried out some benchmarks in order to compare the performance of `vec` implemented as functions over finite domains, as immutable arrays and also as lists (using an existing AFP entry about an implementation of matrices as lists of lists [139]). To do that, we have defined a recursive function which takes as input a matrix A, and in each iteration interchanges the first two rows of $A + A$. Thus, just two basic operations are involved: addition of matrices and interchange rows. The following one is the version for `vec` (which will be executed as functions over finite domains), the definitions for immutable arrays and lists are analogous.[1]

[1] The AFP entry [139] bases its matrix implementation on a list of columns. Our `Benchmark` function in this representation will interchange columns instead of rows. If one uses interchange rows, times would be much slower.

```
primrec Benchmark ::
"nat ⇒ 'a::{semiring_1}^'cols::{mod_type}^'rows::{mod_type}
 ⇒ 'a^'cols::{mod_type}^'rows::{mod_type}
  where
    "Benchmark 0 A = A"
    |
    "Benchmark (Suc n) A = Benchmark n (interchange_rows (A+A) 0 1)"
```

We execute the previous function in two cases:

1. Applied to the 50×50 rational identity matrix with $n = 5$.
2. Applied to the 100×100 rational identity matrix with $n = 20$.

As it was said in Subsection 2.2.3, there are three ways to execute code from Isabelle's definitions: using *simp*, *nbe* and *code*. Table 3.2 shows the performance obtained in each case (*simp, nbe, code*) involving the three representations (*function over finite domains*, *immutable arrays*, *lists*). It is worth noting that inside Isabelle (but not when code is exported), `iarray` is just a wrapper of *list*.

Results show that applying the function `Benchmark` to the 50×50 identity matrix (over rational numbers) with $n = 5$ is usable in practice with any of the three representations. However, when using bigger matrices, functions over finite domains become too slow. Immutable arrays outperform functions and lists in any case, as expected. In the sequel, unless otherwise stated, we will use `iarray` to perform execution tests and to produce refinements from the algorithms over the abstract `vec` representation to the concrete one based on `iarray`.

These benchmarks have been carried out in a laptop with an Intel® Core™ i5-3360M processor with 4GB of RAM and Ubuntu GNU/Linux 14.04. The Isabelle code developed to carry out the benchmarks can be obtained from [46].

		vec	**IArray**	**List**
50×50 $n = 5$	**simp**	250.496s	-	22.775s
	nbe	3.984s	0.486s	4.289s
	code	0.639s	0.159s	2.084s
100×100 $n = 20$	**code**	813.539s	1.137s	3.982s

Table 3.2: Time to execute the `Benchmark` function using different matrix representations.

Chapter 4

Algorithms involving Matrices over Fields

4.1 Introduction

In Chapter 3 a framework where algorithms over matrices can be formalised, executed, refined, and coupled with their mathematical meaning was presented. In this chapter, we show two case studies in which we aim at developing a formalisation in Linear Algebra in which computations are still possible, based on the previous framework. This shows that formalisation and computation can be brought together.

In particular, we have formalised two algorithms involving matrices over fields: the Gauss-Jordan algorithm (Section 4.3) and the QR decomposition (Section 4.5).

Firstly, we will show how we have formalised a mathematical result, known as the "Rank-Nullity theorem" (Theorem 8). The result is of interest by itself in Linear Algebra (some textbooks name it the first part of the *Fundamental Theorem of Linear Algebra*, see [142]) but it is even more interesting if we consider that each linear map between *finite-dimensional* vector spaces can be represented by means of a *matrix* with respect to some provided bases. Every matrix over a field can be turned into a matrix in *reduced row echelon form* (rref, from here on) by means of operations that preserve the behaviour of the linear map, but change the underlying bases; the number of *nonzero rows* of such a matrix is equal to the rank of the (original) linear map; the number of zero rows is the dimension of its *kernel*. There exist specific algorithms to obtain other forms from which the rank of a matrix can be computed, but the rref pays off since it can also be applied to solve other well-known problems in Linear Algebra, as we will see later.

The best-known algorithm for the computation of the rref of a matrix is the Gauss-Jordan elimination method, which will be formalised in this chapter. We also link the original statement of the Rank-Nullity theorem together with the Gauss-Jordan elimination algorithm, and we use both tools to produce *certified* computations of the rank and kernel of linear maps, as well as other certified computations involving some other well-known applications of the algorithm: computation of inverses of matrices, determinants, bases and dimensions of the

four fundamental subspaces, and solutions of systems of linear equations.

As we have said, we also formalise another algorithm involving matrices over fields (in this case, concretely over $\mathbb{R}$): the QR decomposition, which is specially interesting for the computation of the least squares approximation of a system of linear equations with no solution.

It is worth noting that the HMA library of Isabelle/HOL is focused on concrete types such as $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{R}^n$ and on algebraic structures such as real vector spaces and Euclidean spaces, represented by means of type classes. This means that the most important lemma which links linear maps and matrices (and hence, algorithmics with the mathematical meaning) is only proven for real matrices:

```
theorem matrix_works:
assumes "linear f"
shows "matrix f *v x = f (x::real ^ 'n)"
```

Thus, if we exclusively base our work on the HMA library we will just be able to prove the Rank-Nullity theorem over real vector spaces, as we show in Section 4.2. Furthermore, the applications of the Gauss-Jordan algorithm could only be proven for real matrices. However, our interest lies on generating verified code for matrices over arbitrary fields, which is useful in many applications, for instance, in Bioinformatics [87]. This forces us to generalise part of the HMA library, as it is explained in Section 4.4. Once such a task is carried out, the Rank-Nullity theorem will be proven involving finite-dimensional vector spaces over an arbitrary field. Even more, the applications of the Gauss-Jordan algorithm will also be proven in the general case of matrices over fields.

This chapter is divided as follows: firstly, the formalisation of the Rank-Nullity theorem based on the HMA library is presented in Section 4.2. In Section 4.3 we show the formalisation of the Gauss-Jordan algorithm as well as its applications for matrices over fields, explaining briefly where we had to carry out generalisations. Then, the process and difficulties we found while generalising are explained in Section 4.4. Finally, the formalisation of the QR decomposition together with its application to the computation of the least squares approximation of a system of linear equations is shown in Section 4.5.

4.2 The Rank-Nullity Theorem of Linear Algebra

In this section we present how we did the formalisation of the Rank-Nullity theorem, based on the infrastructure presented in the HMA library. This allows us to prove the theorem for real vector spaces, but not involving finite-dimensional vector spaces in general. However, thanks to the generalisation which will be presented in Section 4.4, finally we were able to prove the theorem in its generalised statement.

The Rank-Nullity theorem is a well-known result in Linear Algebra. Its formulation was presented as Theorem 8. It states that the dimension of the null space plus the dimension of the range of a linear map equals the dimension of the domain. The following formalisation is part of the Isabelle AFP [53]; thanks to the infrastructure in the HMA library, the complete formalisation

(involving real matrices) comprises a total of 380 lines of Isabelle code (the proof of the theorem itself is 165 lines). The Isabelle statement of the result is as follows:

```
theorem rank_nullity_theorem:
  assumes "linear (f::('a::{euclidean_space}) ⇒ ('b::{real_vector}))"
  shows "DIM ('a) = dim {x. f x = 0} + dim (range f)"
```

Following the ideas in the HMA library, the vector spaces are represented by means of types belonging to particular type classes; the finite-dimensional premise on the source vector space is part of the definition of the type class `euclidean_space` (in the hierarchy of algebraic structures of the HMA library [90], this is the first type class to include the requisite of being finite-dimensional). Accordingly, `real_vector` is the type class representing vector spaces over $\mathbb{R}$. The operator `dim` represents the dimension of a set of a type, whereas `DIM` is equivalent to `dim`, but refers to the carrier set of that type.

There is one remarkable result that we did not find in textbooks, but that proved crucial in the formalisation. It is somehow specific to how formalising in HOL (HOL Light and Isabelle/HOL) works. Its Isabelle statement reads as follows:

```
lemma inj_on_extended:
  assumes "linear f" and "finite C"
  and "independent C" and "C = B ∪ W"
  and "B ∩ W = {}" and "{x. f x = 0} ⊆ span B"
  shows "inj_on f W"
```

The result claims that any linear map f is *injective* over any collection (W) of linearly independent elements whose images are a *basis* of the *range*; this is required to prove that, given $\{e_1 \dots e_m\}$ a basis of $\ker(f)$, when we complete this basis up to a basis $\{e_1 \dots e_n\}$ of the vector space V, the linear map f is injective over the elements $W = \{e_{m+1} \dots e_n\}$, and therefore its cardinality is the same than the one of $\{fe_{m+1} \dots fe_n\}$ (and equal to the dimension of the *range* of f).

The Isabelle statement of the Rank-Nullity theorem over matrices turns out to be straightforward; we make use of a result in the HMA library (labelled as `matrix_works`) which states that, given any linear map `f`, `f (x::real^'n)` is equal to the (matrix by vector) product of the matrix associated to `f` and `x`. The picture has slightly changed with respect to the Isabelle statement of the Rank-Nullity theorem; where the source and target vector spaces were, respectively, an Euclidean space and a real vector space (of any dimension), they are now replaced by a `real^'n^'m` matrix, *i.e.*, the vector spaces `real^'n` and `real^'m`.

```
lemma fixes A::"real^'a^'b"
  shows "DIM (real^'a) = dim (null_space A) + dim (col_space A)"
```

This statement is used to compute the dimensions of the rank and kernel of linear maps by means of their associated matrices. It exploits the fact that the *rank* of a matrix is defined to be the dimension of its *column space*, also known as column rank, which is the vector space generated by its columns; this dimension is also equal to the ones of the *row space* and the range.

Finally, let us remark that we have presented here the theorem involving real vector spaces (and real matrices in its matrix version). However, as we have said at the beginning of this chapter, the Rank-Nullity theorem can be stated and proven involving finite-dimensional vector spaces over arbitrary fields. We explain the generalisations that we carried out to accomplish such a task in Section 4.4. The generalised version is also part of the AFP and it can be obtained from [52]. The formalisation of the Gauss-Jordan algorithm and its applications will rely on it.

4.3 Gauss-Jordan Algorithm

In this section we present a formalisation of the well-known Gauss-Jordan algorithm together with its applications. In the previous section we have shown how we formalised the Rank-Nullity theorem. As it was said, if we exclusively base the formalisation on the HMA library, the theorem can only be proven involving real matrices. Same occurs to the Gauss-Jordan algorithm: if we base its formalisation on the results presented in the HMA library, then its applications can only be stated and proven involving real matrices. The first version was developed that way and it was published in [55]. However, that was not completely satisfactory, since the algorithm can be executed over matrices over arbitrary fields, which increases the range of applications. To formalise a generalised version of the algorithm, firstly we had to generalise part of the HMA library (see Section 4.4 for further details) and the Rank-Nullity theorem (see [52]). Finally, we were able to accomplish such a task, so the formalised version of the Gauss-Jordan algorithm and its applications involving matrices over arbitrary fields was published as part of the AFP entry [54].

This section will be divided as follows: in Subsection 4.3.1, we present a formalised version of the Gauss-Jordan algorithm over fields, as well as the different applications of it that we have formalised in Isabelle/HOL. In Subsection 4.3.2, we present the code generation process from the formalised Isabelle algorithm to the running versions in SML and Haskell. In Subsection 4.3.3 we introduce some case studies in which the generated algorithms show their usefulness, and some relevant related work. The website [56] includes the SML and Haskell code generated from the Isabelle specifications, an implementation of the algorithm in C++, whose performance is compared to ours, and some input matrices that have been used for profiling and benchmarking. Finally, in Subsection 4.3.4, we draw some conclusions and possible research lines that follow from our work.

4.3.1 The Gauss-Jordan Algorithm and its Applications

In the previous section we have shown how we formalised the Rank-Nullity theorem of Linear Algebra. In our formalisation, it is established that, given V a finite-dimensional vector space over $\mathbb{R}$, W a vector space over $\mathbb{R}$, and $\tau \in L(V, W)$ (a linear map between V and W), $\dim(\ker(\tau)) + \dim(\text{im } (\tau)) = \dim(V)$ or, in a different notation, null (τ) + rk $(\tau) = \dim(V)$. As it has already been said, in our development [52] the previous result was generalised replacing $\mathbb{R}$ by a generic field F of any characteristic. Unfortunately, having formalised the previous result does not provide us with an algorithm that computes the dimension of the image and kernel sets

of a given linear map.

As it has been formalised in [52], every linear map between finite-dimensional vector spaces over a field F is equivalent to a matrix $F^{m\times n}$ (once a pair of bases have been fixed in both F^n and F^m), and, therefore, we can reduce the computation of the dimensions of the range (or rank) and the kernel (or nullity) of a linear map to the computation of the *reduced row echelon form* [134] (or *rref*) of a matrix; the number of nonzero rows of such a matrix provides its rank, and the number of zero rows its nullity. The Gauss-Jordan algorithm computes the *rref* of a matrix.

Algorithm 1 describes the Gauss-Jordan elimination process that inspired our Isabelle formalisation. We must note that our Isabelle implementation of the algorithm differs from Algorithm 1 since we have replaced side effects and imperative statements (such as *for* loops) by standard functional constructs that are detailed below.

Algorithm 1 Gauss-Jordan elimination algorithm

Input: A, a matrix in $F^{m\times n}$;
Output: The *rref* of the matrix A
$l \leftarrow 0$; ▷ l is the index where the pivot is placed
for $k \leftarrow 0$ **to** $(\text{ncols}\, A) - 1$ **do**
  if $\text{nonzero}\, l\, (\text{col}\, k\, A)$ **then** ▷ Check that col. k has a pivot over pos. l
    $i \leftarrow \text{index-nonzero}\, l\, (\text{col}\, k\, A)$ ▷ Let i be the such entry
    $A \leftarrow \text{interchange-rows}\, A\, i\, l$ ▷ Rows i and l are interchanged
    $A\, l \leftarrow \text{mult-row}\, A\, l\, (1/A\, l\, k)$ ▷ Row l is multiplied by $(1/A\, l\, k)$
    for $t \leftarrow 0$ **to** $(\text{nrows}\, A) - 1$ **do**
      if $t \neq l$ **then** ▷ Row t is added row l times $(-A\, t\, k)$
        $A\, t \leftarrow \text{row-add}\, A\, t\, l\, (-A\, t\, k)$
      end if
    end for
    $l \leftarrow l + 1$
  end if
end for

Algorithm 1 traverses the columns of the input matrix, finding in each column k an element in the i-th row (the first nonzero element in a row greater than or equal to the index l); if such an element (the *pivot*) exists, rows i and l are interchanged (if the matrix has maximum rank, l will be equal to the column index, otherwise it will be smaller), and the l-th row is multiplied by the inverse of the pivoted element; this row is then used to perform row operations to reduce all remaining coefficients in column k to 0. If a column does not contain a pivot, the algorithm processes the next column. The algorithm performs exclusively *elementary row operations*.

In our Isabelle specification, rows and columns are assigned finite enumerable types (in the sense that they admit an extensional definition), over which matrices are represented as functions (see Section 3.2). We have replaced the operations in the previous algorithm that produce side effects (for instance, *interchange-rows*, *mult-row*, *row-add*) by Isabelle *functions* whose outputs are matrices (see the definitions in Section 3.1). We have replaced the first (and outer) *for* loop ranging along the columns indexes in the original algorithm by

means of a *fold* operation of a list with the desired indexes. The second (and inner) *for* loop, whose range is the rows of a given column, is replaced by means of a *lambda expression* over the rows type (indeed, each row is itself a function over this finite type).

Note that we have represented matrices by means of functions over the columns type of a function over the rows type (see Section 2.2.2). The following Isabelle code snippets are presented to provide a better understanding of the gap between Algorithm 1 and our Isabelle implementation; additionally, they provide the complete Isabelle definition of the *Gauss-Jordan* algorithm, in terms of elementary row operations.

The Isabelle definition `Gauss_Jordan_in_pos` shown below selects an index i greater than or equal to l in the k-th column (line 6 in Algorithm 1), interchanges rows i and l (line 7), multiplies the row l by the multiplicative inverse of the element in position (l, k) (line 8) and reduces the rest of the rows of the matrix, by means of a lambda expression, which represents the new created matrix (lines 9 to 13).

```
definition "Gauss_Jordan_in_pos A l k =
 (let
   i = (LEAST n. A $ n $ k ≠ 0 ∧ n ≥ l);
   interchange_A = (interchange_rows A i l);
   A' = mult_row interchange_A l (1/interchange_A $ l $ k)
  in
    vec_lambda (λt. if t = l then A' $ l
     else (row_add A' t l (-(interchange_A $ t $ k)))$ t))"
```

The definition `Gauss_Jordan_column` checks if there is a pivot on column k of a matrix A, starting from position i and then applies the previous operation `Gauss_Jordan_in_pos` (and corresponds with line 5 in Algorithm 1).

```
definition "Gauss_Jordan_column (i', A) k' =
 (let
  i = from_nat i';
  k = from_nat k'
 in
  if (∀m≥i. A $ m $ k = 0) ∨ (i = nrows A) then (i, A)
  else (i+1, (Gauss_Jordan_in_pos A i k)))"
```

The traversing operation over columns (line 4) is performed by *folding* the operation `Gauss_Jordan_column` over the list containing the columns type universe, starting with a pair given by the index 0 (line 3) and the input matrix A.

```
definition "Gauss_Jordan_upt A k = snd (foldl Gauss_Jordan_column (0,A)
[0..<Suc k])"

definition "Gauss_Jordan A = Gauss_Jordan_upt A ((ncols A) - 1)"
```

The algorithm admits several variants, both to speed up its performance and also to avoid numerical stability issues with floating-point numbers [68, Chap. 9], but in order to reduce the complexity of its formalisation we chose the above presentation.

The *rref* of a matrix has indeed further applications than computing the rank. Since this version of Gauss-Jordan is based on elementary row operations, it can be also used for:

- Computation of the inverse of a matrix, by "storing" the elementary row operations over the identity matrix.
- Determinants, taking into account that some of the elementary row operations can introduce multiplicative constants.
- Computation of bases and dimensions of the null space, left null space, column space, and row space (also known as fundamental subspaces) of a matrix.
- Discussion of systems of linear equations ($\{x \in F^m \mid A \cdot x = b\}$), both consistent (with unique or multiple solutions) and inconsistent ones.

The formalisation of the Gauss-Jordan algorithm, together with numerous previous results in Linear Algebra, and the different applications that are presented above, summed up *ca.* 10000 lines of code (see Table A.2); the proofs check that the defined objects (determinant, inverse matrix, solution of the linear system, fundamental subspaces) are preserved (or modified in a certain way) after each algorithm step (and more concretely, after each elementary row operation). By using product types, we store an initial value for the desired object, and the input matrix. In the case of determinants, the initial pair is $(1, A)$.

```
definition "Gauss_Jordan_upt_k_det_P A k =
 (let step = foldl Gauss_Jordan_column_k_det_P (1,0,A) [0..<Suc k]
   in (fst step, snd (snd step))))"

definition "Gauss_Jordan_det_P A =
 Gauss_Jordan_upt_k_det_P A ((ncols A) - 1)"
```

After each algorithm step, a corresponding modification is applied to the first component. The previous function `Gauss_Jordan_upt_k_det_P` takes as inputs a matrix A and a column k, performs Gauss-Jordan in the first k columns of A, and returns a pair whose first component is the product of the coefficients originated when performing Gauss-Jordan in the first k columns, and the second component contains the matrix obtained after performing Gauss-Jordan in the first k columns.

In the computation of each of the previous pairs, there is a notion of *invariant* that is preserved through the Gauss-Jordan algorithm steps. For instance, in the case of determinants, given a matrix A, after n elementary operations the pair (b_n, A_n) is obtained, and it holds that $b_n \times (\det A) = \det A_n$, and $b_n \neq 0$. Since the algorithm terminates (the elements indexing the columns are an enumerable type), after a finite number, m, of operations, we obtain a pair $(b_m, \operatorname{rref} A)$ such that $b_m \times (\det A) = \det(\operatorname{rref} A)$; since we proved that the determinant of $\operatorname{rref} A$ is the product of its diagonal elements, the computation is completed.

```
lemma det_Gauss_Jordan_det_P:
 fixes A::"'a::field^n^n::mod_type"
 defines "comp == Gauss_Jordan_det_P A"
 shows "(fst comp) * det A = det (snd comp)"
```

In a similar way, we perform the proof of the computation of the inverse of a matrix (starting from an input square matrix A of dimension n, the pair (I_n, A) is built and after every row operation, (P', A') is such that $P' \cdot A = A'$), as long as A is invertible (in other words, $\operatorname{rref} A = I_n$). When the Gauss-Jordan algorithm reaches $\operatorname{rref} A$, the first component of the pair holds the matrix P (this matrix is indeed the product of every elementary operation performed). The computations of the bases of the fundamental subspaces of linear maps are also based on the computation of the matrix P generated from applying the Gauss-Jordan algorithm to A (or A^T) and the same operations to I_n (I_m). Their Isabelle definitions are given as follows:

```
definition "basis_null_space A =
  {row i (P_Gauss_Jordan (transpose A))  |i. to_nat i ≥ rank A}"
definition "basis_row_space A =
  {row i (Gauss_Jordan A)  |i. row i (Gauss_Jordan A) ≠ 0}"
definition "basis_col_space A = {row i (Gauss_Jordan (transpose A))
  |i. row i (Gauss_Jordan (transpose A)) ≠ 0}"
definition "basis_left_null_space A =
 {row i (P_Gauss_Jordan A) | i. to_nat i ≥ rank A}"
```

With respect to the solution of systems of linear equations, $A \cdot x = b$, we prove that, if a system is *consistent*, its set of solutions is equal to a single point plus any element which is a solution to the homogeneous system $A \cdot x = 0$ (or, in other words, the null space of A). In order to solve the system, we start from the pair (I_n, A) and after applying the Gauss-Jordan algorithm to A, and the same elementary operations to I_n, a new pair $(P, \operatorname{rref} A)$ is obtained. The vector b is then multiplied by P, and from its number of nonzero positions and the rank of A (or $\operatorname{rref} A$) the system is classified as *consistent* or *inconsistent*. In the first case, a single solution is computed by taking advantage of $\operatorname{rref} A$. The basis of the null space is computed applying Gauss-Jordan elimination to A^T in (I_m, A^T), and performing similar row operations to I_m.

In order to consider inconsistent systems suitably, we have represented the solutions as elements of the Isabelle *option* type (whose elements are of the form `(Some x)` and `None`); the set of solutions of a system will be presented as a singular point (whenever the system has solution), and the corresponding vectors forming a basis of the null space (or the empty set). As an additional result, we have formalised in Isabelle that every solution to a given system is of the previous form. Previous formalisations of the solution of systems of linear equations through the Gauss-Jordan algorithm (see, for instance, [117]) and most Computer Algebra systems compute exclusively single solutions (even more, for exclusively compatible systems with equal number of equations and unknowns).

In Chapter 3 we pointed out that the HMA library permitted us to keep the tie between Linear Algebra and algorithmics. The crucial result to this aim consists in establishing and formalising a link between *linear maps* and *matrices*.

The following result states that applying a linear map f to an element x of its source vector space is equal to multiplying the associated matrix to f by the vector x (the matrix represents the linear map with respect to some previously fixed bases). The generalisation of this result to fields was formalised as part of

this development, inspired by the result over *real vector spaces*, already available in the HMA library. This result is available in the file *Generalisations.thy* in [52] and explained in Section 4.4).

```
lemma matrix_works:
  assumes "linear (op *s) (op *s) f"
  shows "matrix f *v x = f (x::'a::field^n)"
```

Then, in file *Linear_Maps.thy*, several results have been proven based on this one, that relate linear maps and properties of their associated matrices. We have proven all the theorems presented in Subsection 2.1.1, as well as most of the theorems presented in [134, Chap. 2]. Many of them relate changes of bases with invertible matrices and coordinates. For instance, the following one corresponds to [134, Theorem 2.13]:

```
lemma invertible_matrix_is_change_of_basis:
  assumes invertible_P: "invertible P"
  and basis_X: "is_basis (set_of_vector X)"
  shows "∃!Y. matrix_change_of_basis Y X = P
         ∧ is_basis (set_of_vector Y)"
```

The previous theorem states that an invertible matrix corresponds to a matrix of change of basis. Moreover, Theorems 10 and 11 in Subsection 2.1.3 have also been formalised in the file *Linear_Maps.thy*. Their corresponding statements in Isabelle look as follow (`matrix' X Y f` represents the matrix of a linear map `f` with respect to the bases `X` and `Y`):

```
corollary equivalent_iff_exist_matrix':
  shows "equivalent_matrices A B ⟷ (∃X Y X' Y'
f::'a::{field}^'n⇒'a^'m.
  linear (op *s) (op *s) f ∧ matrix' X Y f = A ∧ matrix' X' Y' f = B
  ∧ is_basis (set_of_vector X) ∧ is_basis (set_of_vector Y)
  ∧ is_basis (set_of_vector X') ∧ is_basis (set_of_vector Y'))"


corollary similar_iff_exist_matrix':
  fixes A B::"'a::{field}^'n^'n"
  shows "similar_matrices A B ⟷ (∃X Y f. linear (op *s) (op *s) f ∧
matrix' X X f = A
    ∧ matrix' Y Y f = B ∧ is_basis (set_of_vector X) ∧ is_basis
(set_of_vector Y))"
```

Another instance of relationships between linear maps and matrices is the following result, which proves that a linear map is bijective if and only if its associated matrix has full column rank [134, Th. 2.11].

```
lemma linear_bij_rank_eq_ncols:
  fixes f::"'a::field^n::mod_type ⇒ 'a^n"
  assumes "linear (op *s) (op *s) f"
  shows "bij f ⟷ rank (matrix f) = ncols (matrix f)"
```

The crucial result in the formalisation of an algorithm preserving the rank

of matrices is that elementary operations (*i.e.*, invertible matrices) applied to a matrix preserve its rank:

```
lemma fixes A::"'a::field^'n^'m" and "P::'a^'m^'m"
 assumes "invertible P" and "B = P ** A"
 shows "rank B = rank A"
```

As a consequence of the previous result, we also proved that linear maps are preserved by elementary operations (only the underlying bases change; thus, algorithmics and mathematical meaning are connected). Note that the previous machinery is not particular to the Gauss-Jordan algorithm, but it could also be reused for different algorithms in Linear Algebra.

The *rank* of a matrix is now a computable property (by means of the Gauss-Jordan algorithm presented), as it is its number of columns of its rref.

4.3.2 The Refinement to Immutable Arrays

As it has been explained in Chapter 3, we have carried out two refinements in order to be able to execute Linear Algebra algorithms formalised in the HMA library.

Following the infrastructure presented in Section 3.2, we can execute the Gauss-Jordan algorithm and its applications by means of the refinement to functions over finite types in order to get verified computations. Below we present examples of the evaluation (by means of SML generated code) of the Gauss-Jordan algorithm to compute the dimension of the rank (which is also the one of the column space) and the one of the null space of given matrices of reals:

```
value[code] "rank (list_of_lists_to_matrix
    [[1,0,0,7,5],[1,0,4,8,-1],[1,0,0,9,8],[1,2,3,6,5]]::real^5^4)"
value[code] "vec.dim (null_space (list_of_list_to_matrix
    [[1/7,0,0,7/5,5],[1,0,4/3,8,-1],[1,0,0,9,8],[1,2,3/11,6,5]]))"
```

The previous computations have been carried out with matrices represented as functions and the results are respectively `4` and `1`. They are almost instantaneous, but the computation of the algorithm over matrices of size 15×15 is already very slow (several minutes). Thus, to make use of the refinement to immutable arrays presented in Section 3.3 is necessary in order to make the algorithm applicable to some real-world problems.

Again, each function over elements of type `vec` needs to be replaced by a new function over type `iarray`. This requires first specifying a function over the type `iarray`, and then proving that it behaves as the one over type `vec`. This statement will be labelled as a code equation and it will be unfolded in the code generation process. Let us show one example of code equation which transforms the Gauss-Jordan algorithm over `vec` to `iarray`:

```
lemma [code_unfold]:
 fixes A::"'a::{field}^col::{mod_type}^row::{mod_type}"
 shows "matrix_to_iarray (Gauss_Jordan A) =
  Gauss_Jordan_iarrays (matrix_to_iarray A)"
```

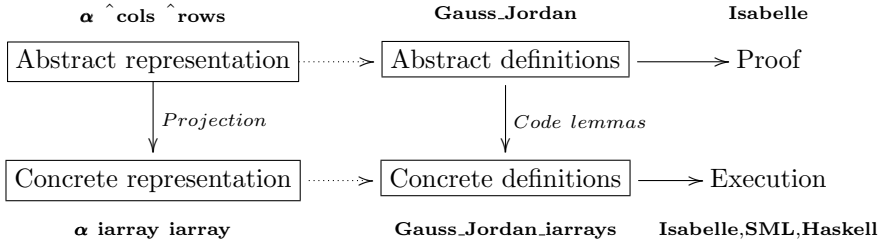


Figure 4.1: Refinement diagram for the formalisation of the Gauss-Jordan algorithm

The code equation certifies that it is correct to replace the function `Gauss_Jordan` (defined over abstract matrices, or elements of type `vec`), the one proven to compute the rref of a matrix, by the function `Gauss_Jordan_iarrays`. Figure 4.1 shows how the refinement works for such a function. The same approach will be applied in the formalisation of the QR decomposition, the echelon form, and the Hermite normal form. We will omit analogous figures for such algorithms. Correctness is proven using the `vec` representation, execution is carried out using `iarray`.

The proving effort (*ca.* 2800 code lines, see Table A.2) to prove the type refinement is significantly smaller than the formalisation itself (*ca.* 10000). It must be noticed that reproducing the formalisation over *iarrays* would presumably take more than the aforementioned 10000 lines, since the type `vec` has particular properties that ease and simplify the proofs. At least two features of this refinement from `vec` to `iarray` can be identified that have an influence in the modest amount of work devoted to it. First, the close relationship between the original data type `vec` and the new data type `iarray`, since both of them share a similar interface that allows us to easily *identify* operations between them, as we have already explained in Chapter 3. Second, the fact that we preserved the original algorithm design, replacing operations over the `vec` type by equivalent ones over the `iarray` type. Nevertheless, the Isabelle code generator leaves the door open to algorithmic refinements (obviously, the bigger the differences between the *abstract* and the *concrete* algorithms, the greater the proving effort that has to be invested to fill such a gap).

4.3.3 The Generated Programs and Related Work

4.3.3.1 The Generated Programs

The Isabelle code generation facility is now applied to generate the SML and Haskell code from the Gauss-Jordan algorithm specification. Note that in this process both the serialisations and data type refinements presented in Sections 3.4 and 3.3 respectively are used. The automatically generated code sums up 2300 lines in SML and 1300 in Haskell. The SML and Haskell sources can be automatically generated from the development and they are also available from [56]. They closely resemble the Isabelle definitions presented in Subsec-

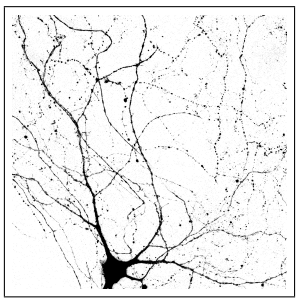


Figure 4.2: Image (2560×2560 px.) of a neuron acquired with a confocal microscope.

tion 4.3.1 of the Gauss-Jordan algorithm.

It shall be clear that the programs pose a serious drawback, since the input matrices are stored by means of immutable data types. Since we are using *iarrays* for representing both vectors and matrices, every operation with side effects requires additional memory consumption, whereas mutable data structures in imperative languages should avoid this matter. Even so, the obtained programs were useful in at least a couple of interesting case studies:

- The rank (computed, for instance, as the number of nonzero rows of the *rref*) of a $\mathbb{Z}_2$ matrix, permits the computation of the number of connected components of a digital image; in Neurobiology, this technique can be used to compute the number of synapses in a neuron, automating a cumbersome task previously made "by hand" by biologists (see [87] for details). With our programs, the computations can be carried out on images at least of 2560×2560 px. size (which are conventional ones in real life experiments, see Figure 4.2). The running time to obtain these computations was *ca.* 160 seconds.

- Varona *et al.* [59] detected that Mathematica®, at least in versions 8.0, 9.0 and 9.0.1, was erroneously computing determinants of matrices of big integers (about 10 thousand digits), even for matrices of small dimensions (in their work they present an example of a matrix of dimension 14×14). The same computation, performed twice, can produce different results. This kind of failure could be critical in cryptology. The bug was reported to the Mathematica® support service. The error might be originated in the use of some arithmetic operations modulus large primes (apparently, the prime numbers could be either not large enough or not as many as required). With our verified program, the computation takes *ca.* 0.64 seconds (in MLton; Haskell and Poly/ML performance are also practically the same over inputs of such sizes), and the result obtained is the same as in Sage (the running time is negligible) and Maple™. Our algorithm relies on the arbitrarily large integers as implemented in each functional language. The computing time in Mathematica® (of the wrong result) sums up 2.68 seconds.

In general, the applicability of the generated programs is widely possible. Linear Algebra is well-known for being a central tool for graph theory, coding theory, and cryptography, among many other fields.

4.3.3.2 Related Work

The formalisation of Abstract Algebra has been of recurrent interest for the theorem proving community, and it would be difficult to provide here a complete survey on the subject; even so, the formalisation of Linear Algebra algorithms has not received such an attention from the community. This is somehow surprising, since new algorithms are being continuously implemented in Computer Algebra systems in a search for better performance.

Nevertheless, it is worth mentioning the CoqEAL (standing for Coq Effective Algebra Library) effort [44]. This work is devoted to develop a set of libraries and commodities over which algorithms over matrices can be implemented, proved correct, refined, and finally executed. We try to compare our work with theirs in each of these terms.

The *implementation of algorithms* is rather similar, except for the peculiarities of the Isabelle and Coq systems; the algorithms in CoqEAL take advantage of the SSReflect library and facilities [73], which indeed represents vectors and matrices as functions, and therefore quite close to ours. The Coq type system is richer than the one of Isabelle; in particular, it allows the definition of dependent types, enabling the use of submatrices and some other natural constructions, that in our Isabelle implementation have been avoided.

In order to *prove the correctness* of algorithms, CoqEAL algorithms can take advantage of the use of dependent types to extensively apply induction over the dimension of matrices, whereas we chose to apply induction over the number of columns; there might be algorithms where this fact really poses a difference, but we did not detect any particular hindrance in the proof of the algorithm computing the *rref*.

Refinements in CoqEAL can be performed both on the algorithms and on the data structures [39] (as we also do in our development); data type refinements in CoqEAL are made in a *parametricity* fashion, since algorithms are implemented and proved over unspecified operations (by means of *type classes*) that are later replaced with computational or proof oriented types. Then, some parametricity lemmas state the equivalence between computational and proof oriented operations. These lemmas sometimes introduce additional parameters (such as the size of matrices) that could be unnecessary in some of the representations. In practice, the data type refinements in CoqEAL are reduced to using either functions or lists for representing vectors (and then matrices iteratively as lists of lists). Our approach is more flexible since it is based in proving the equivalence (modulus type morphisms) of the operations in the computational and proof oriented types [79]. In Isabelle, there is no special requirement over the proof type and the computational type, as long as a suitable morphism exists between both, and operations that are proven equivalent in both types.

Regarding *execution*, there is a crucial difference between CoqEAL's approach and ours. In the CoqEAL framework computations are carried out over the Coq virtual machine (and thus "inside" the Coq system, avoiding code generation), a bundle of Coq libraries to optimise running time. Our approach is thought for code generation, and executions are completed in Haskell or SML (they can be also performed "inside" of Isabelle, but execution times are considerably slower). This provides us with a remarkable edge in performance. For instance, the computation of the rank of a $\mathbb{Z}_2$ matrix of size 1000×1000 consumes 32 seconds in Poly/ML but 121 seconds in Coq (following the figures

"Imperative" version		Verified version	
Function	**Time perc.**	**Function**	**Time perc.**
nth.fn	29.8%	sub	33.4%
upd.fn.fn.fn	12.2%	of_fun	32.7%
IntInf.schckToInt64	12.1%	IntInf.extdFromWord64	9.3%
make.fn	8.1%	IntInf.schckToInt64	7.5%
plus_nat.fn	7.9%	row_add_iarray.fn	6.3%
...	...	...	...
Total			
9.42 seconds of CPU time		10.06 seconds of CPU time	
(0.04 seconds of GC)		(0.22 seconds of GC)	

Table 4.1: Profiling of the "imperative" and verified versions of Gauss-Jordan.

in [60]), even if their algorithm exclusively computes the rank of the matrix, without any additional information, whereas ours produces the *rref* of the matrix. One potential source of inefficiency in CoqEAL performance could be the use of *lists* to encode vectors and matrices.

A different tempting approach would be to verify an *imperative* version of the algorithm producing the *rref* of a matrix. The implementation of imperative programs and (mutable) arrays in Isabelle/HOL has been studied by Bulwahn *et al.* [33], through the use of suitable monads, giving place to a library called *Imperative HOL*. When applying code generation, mutable arrays are refined to the SML *Array.array* type, which is also mutable (whereas immutable arrays were refined in our work to the SML type *Vector.vector*). The authors present two different case studies; in one of them the performance gain is a 30% with respect to a purely functional implementation using balanced trees. The use of monads introduces inherent intricacies both in the definitions and also in the consequent formalisations.

As an experiment, we produced a (non-formalised) *imperative* implementation of the Gauss-Jordan algorithm, inspired by Algorithm 1 and our functional representation (see file *Gauss_Jordan_Imp.thy* in [56]). Algorithm 1 involves numerous matrix rows and column traversals that we have implemented by recursion (up to our knowledge, there is not a suitable array traverse operator in Imperative HOL).

We generated SML code from this imperative version, and carried out some performance tests between the *functional* verified version presented in Subsection 4.3.3.1 and the new *imperative* one. Rather surprisingly, the functional and imperative versions performed almost equally (with a little bias of a 10% for each version depending on the compiler used, MLton or Poly/ML). Profiling both algorithms (see Table 4.1) it seemed clear that the amount of time spent in the functional version of the algorithm in the construction of new immutable arrays (operation *of_fun*) was greater than the amount of time devoted in the imperative version (operation *make.fn*), but this amount of time was balanced with the time spent in the imperative version performing updating and the remaining recursive operations (operations *sub* and *nth.fn* represent both the array access operation).

Table 4.1 presents the result of profiling in MLton the computation of the *rref* of a 600×600 matrix with inputs in $\mathbb{Z}_2$ (profiling itself adds up an overhead of almost 20% in computing time).

Replacing some of the recursive operations used in the imperative version

Matrix sizes	**C++ version**	**Verified version**
600×600	01.33s.	06.16s.
1000×1000	05.94s.	32.08s.
1200×1200	10.28s.	62.33s.
1400×1400	16.62s.	97.16s.

Table 4.2: C++ and verified versions of the Gauss-Jordan algorithm.

by traversals may reduce computing time, but apparently by a low margin. An additional drawback of using Imperative HOL is that the underlying types (used in arrays) have to be proven instances of the type class `countable`, since the *heap* is modelled as an *injective* map from addresses to values. In our particular case, this limitation prevents us from using this representation for the type `real`.

As an additional test of the performance of the obtained verified code, we implemented the same algorithm as presented in Algorithm 1 in C++ (see file *Gauss_Jordan.cpp* in [56]), where vectors are represented by C++ arrays, and array traversals are replaced by *for* loops. We produced some benchmarking (involving $\mathbb{Z}_2$ matrices, where the coefficients are represented by means of booleans in both SML and C++) to compare our verified functional version, executed with the Poly/ML interpreter, against the C++ version, that is presented in Table 4.2.

The obtained times with both versions are of the same order of magnitude, and grow cubically with respect to the number of rows of the matrices (it is worth noting that the Gauss-Jordan algorithm is of cubic order). Apparently, the use of *immutable* arrays in our verified code does not pose a drawback, and both algorithms scale similarly. For more execution tests and benchmarks of the Gauss-Jordan algorithm applied to randomly generated matrices over the fields $\mathbb{Z}_2$, $\mathbb{Q}$, and $\mathbb{R}$ see Appendix A.

4.3.4 Conclusions and Future Work

In this section we have presented a formalisation of the Gauss-Jordan algorithm and its principal applications in Linear Algebra. The algorithm has been proved in an environment (the HMA library and the framework presented in Chapter 3) that allows us to keep the mathematical meaning of the results (for instance, the relation between matrices and linear maps) and over a representation of matrices which simplifies the proving effort. Additionally, the algorithm has been refined to a more suitable data type, and generated to SML and Haskell, languages in which we have been capable of executing the algorithm over matrices of remarkable dimensions (in our case study with digital images) and inputs (in matrices with *big* coefficients, which were erroneously handled by Mathematica®). The infrastructure created can be certainly reused; some of the serialisations introduced in Section 3.4 are already part of the Isabelle library. The mathematical results are proven over a well-established Isabelle library (the HMA library). Results on matrices include many basic operations. The data type refinement from vectors as matrices to immutable arrays is also easily reusable, not only for Linear Algebra purposes.

The number of lines of verified code generated in our work (*ca.* 2300 in SML, 1300 in Haskell) is considerable and covers a wide range of applications in Lin-

ear Algebra. Both its formalisation and the refinement, available in [54], took *ca.* 10000 lines of Isabelle code. Our development of the Rank-Nullity theorem [52] and the HMA library infrastructure were intensively reused, reducing significantly the amount of mathematical results to be formalised. Isabelle/HOL also provided powerful and efficient features to produce data type refinements. Table A.2 shows the collection of files and their sizes produced in our work.

The design decisions that we are subject to (specially, the use of *iarrays* in the generated code) does not seem to pose a real limitation, since the other options explored (use of monads and use of imperative languages) do not provide a remarkable edge over our verified version.

4.4 Generalisations

In the previous sections, we have presented a formalisation of the Rank-Nullity theorem and the Gauss-Jordan algorithm based on the HMA library and our framework explained in Chapter 3. In such developments, we also set up Isabelle to generate code from the matrix representation presented in the HMA library. A refinement to immutable arrays was carried out to improve performance. We also formalised some of the well-known applications of the Gauss-Jordan algorithm: computation of ranks, inverses, determinants, dimensions and bases of the four fundamental subspaces of a matrix, and discussion of systems of linear equations (in every case: unique solution, multiple solutions and no solution). Verified code of these computations is generated to both SML and Haskell.

However, while formalising the previous results we found a limitation in the HMA library: some important results that we needed were only proven for real matrices or for real vector spaces. Due to this fact, we were only able to prove the Rank-Nullity theorem for real vector spaces and generate verified code of the Gauss-Jordan algorithm for real matrices. But we were especially interested in matrices whose coeffients belong to some other fields. For instance, as we have already explained, the rank over $\mathbb{Z}_2$ matrices permits the computation of the number of connected components of a digital image. This limitation arises since the HMA library derives from earlier formalisations limited to concrete types, such as $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{R}^n$. Many results presented in the HMA library are ported from Harrison's work in HOL Light [85], where most theorems are proven only for $\mathbb{R}^n$.

Hölzl *et al.* [90] improved significantly the HMA library. They presented a new hierarchy of spaces based on type classes to represent the common structures of Multivariate Analysis, such as topological spaces, metric spaces and Euclidean spaces. This improvement showed the power of Isabelle's type system. Some limitations still remain; for instance, most properties about vector spaces are only demonstrated in the HMA library over real vector spaces, impeding us from working with matrices whose elements belong to other fields. Generalising the results in the HMA library is a known problem but has not been tackled. Harrison already pointed it out in his work [85]: "many proofs are *morally the same* and it would be appealing to be able to use similar technology to generalise the proofs". Avigad also found this limitation when working with the HMA library in his formalisation of the Central Limit Theorem [21]; he said that some concepts "admit a common generalisation, which would unify the library substantially".

In this section we present a work in progress which aims at being the foundation stone to get such a generalisation. The final aim would be to generalise the library as far as possible. As work done, we present the generalisations and the methodology that permitted us to prove the Gauss-Jordan algorithm over matrices whose elements belong to an arbitrary field.

4.4.1 Generalisation of the HMA library

Mathematical structures presented in the HMA library are defined by means of *type classes* [77]; type classes, as presented in Subsection 2.2.1, are provided by Isabelle and have great advantages: they allow us to organise polymorphic specifications, to create a hierarchy among different classes, to provide instances, to produce a simple syntax and to simplify proofs thanks to the Isabelle type inference mechanism. A type class C specifies assumptions $P_1, \ldots, P_k$ for constants $c_1, \ldots, c_m$ (that are to be overloaded) and may be based on other type classes $B_1, \ldots, B_n$. Only one type variable α is allowed to occur in the type class specification. Hence, if we want to prove properties of arbitrary vector spaces (where two type variables appear), we have to use locales instead.

As it was explained in Subsection 2.2.1, *locales* [24] are an Isabelle approach for dealing with parametric theories and they are specially suitable for Abstract Algebra. On the other hand, code generation within locales with assumptions essentially does not work.

We are on the borderline: we want to use abstract structures such as vector spaces or modules (we have to use locales) but preserving the executability (code generation). Our proposal is to work with a mix between locales and type classes: every possible lemma is generalised to newly introduced locales, but lemmas required in type classes are kept (because they belong there, or because they are obtained thanks to an **interpretation** of the corresponding abstract **locale**).

4.4.1.1 An Example of Generalisation

Let us illustrate the previous methodology with an example. A key lemma in the HMA library is the one which states the link between matrices and linear maps:

```
theorem matrix_works:
assumes "linear f"
shows "matrix f *v x = f (x::real ^ 'n)"
```

It is stated for linear maps between real vector spaces. The `linear` predicate in the premise is introduced by the following **locale** definition:

```
locale linear = additive f for f :: "'a::real_vector ⇒ 'b::real_vector"
+  assumes scaleR: "f (scaleR r x) = scaleR r (f x)"
```

Only one parameter is required: a map f. In the heading, the type of f is fixed as a map between two real vector spaces (`real_vector` type class; do note that the definition of the **locale** is using underlying type classes). In order to generalise it to *arbitrary* vector spaces over the same field, we propose the following definition:

```
locale linear = B: vector_space scaleB + C: vector_space scaleC
  for scaleB :: "('a::field ⇒ 'b::ab_group_add ⇒ 'b)" (infixr "*b" 75)
  and scaleC :: "('a ⇒ 'c::ab_group_add ⇒ 'c)" (infixr "*c" 75) +
  fixes f :: "('b⇒'c)"
  assumes cmult: "f (r *b x) = r *c (f x)"
  and add: "f (a + b) = f a + f b"
```

This new **locale** has three parameters, instead of one: the scalar multiplications *scaleB* and *scaleC*, which fix both the vector spaces and the field, and the map *f*. Now we can interpret F^n (where F is a field) as a vector space over F and prove the linear interpretation for F^n (the corresponding linear map is the multiplication of a matrix by a vector):

```
interpretation vec: vector_space "op *s :: 'a::field ⇒ 'a^'b ⇒ 'a^'b"
interpretation vec: linear "op *s" "op *s" "(λx. A *v (x::'a::field^_))"
```

After reproducing in the new **locale** the lemmas involved in the proof, we obtain the generalised version. Note the differences between both statements:

```
theorem matrix_works:
assumes "linear (op *s) (op *s) f"
shows "matrix f *v x = f (x:: 'a::field ^ 'n)"
```

4.4.1.2 The Generalisation of the Gauss-Jordan Algorithm

Our aim is to generalise the Gauss-Jordan algorithm to generate verified code for matrices with elements belonging to a generic field. In Subsection 4.4.1.1, we have shown an example of how to carry out this generalisation. As Harrison pointed out [85], in many cases the proof is essentially the same. However, the procedure is not immediate and almost every demonstration involves subtle design decisions: introduce new locales, syntactic details, interpretations inside the lemma to reuse previous facts, change the types properly and so on. In broad terms, we have carried out four kinds of generalisations in the HMA library to achieve verified execution over matrices with elements belonging to a field:

1. Lemmas involving real vector spaces (a **class**) are now generalised to arbitrary vector spaces (a **locale**).
2. Lemmas involving Euclidean spaces (a **class**) have been generalised to finite-dimensional vector spaces (a **locale**).
3. Lemmas involving real matrices have been generalised to matrices over any field (thanks to the previous two points).
4. Lemmas about determinants of matrices with coefficients in a real vector space are now proven for matrices with coefficients in a commutative ring.

In the HMA library the first time that the notion of a finite basis appeared was in the *euclidean_space* class. Now, we have introduced a new **locale** *finite_dimensional_vector_space* and generalised several proofs from the *euclidean_space* class to that locale. Thanks to those generalisations, some lemmas that were stated in the HMA library only over real matrices are now

proven over more general types. Let us take a look at the following lemma, which claims that a matrix is invertible iff its determinant is not null. The following version is the one available in the HMA library, stated for integral domains:

```
lemma det_identical_rows:
  fixes A :: "'a::linordered_idom^'n^'n"
  assumes ij: "i ≠ j" and r: "row i A = row j A"
  shows "det A = 0"
proof-
  have tha: "⋀(a::'a) b. a = b ⟹ b = - a ⟹ a = 0"  by simp
  have th1: "of_int (-1) = - 1" by simp
  let ?p = "Fun.swap i j id"
  let ?A = "χ i. A $ ?p i"
  from r have "A = ?A" by (simp add: vec_eq_iff row_def Fun.swap_def)
  then have "det A = det ?A" by simp
  moreover have "det A = - det ?A" by (simp add: det_permute_rows[OF
    permutes_swap_id] sign_swap_id ij th1)
  ultimately show "det A = 0" by (metis tha)
qed
```

The original statement comes from Harrison's formalisation [85], where the lemma is proven over real matrices. The Isabelle proof presented above follows the one presented in most of the literature. Essentially, in the proof it is deduced that $det(A) = -det(A)$ and thus $det(A) = 0$. But such a property does not hold in rings whose characteristic is 2 (such as $\mathbb{Z}_2$). For instance, in a book by Axler [22] the statement is presented for commutative rings but it is proven without taking into account rings with characteristic 2. The same appears in a book by Strang [142], but the author warns that the demonstration fails in the case of $\mathbb{Z}_2$ matrices. To generalise the result to an arbitrary ring, we had to change totally the proof and work over permutations.[1]

Not only we need to change some proofs, sometimes we have to introduce new definitions. For instance, to multiply a matrix by a scalar. The HMA library works with real matrices, so the next operation is used: `(op *R) :: real ⇒ 'a ⇒ 'a`. In the generalisation, we would like to multiply a matrix of type `'a^'n^'m` by an element of type `'a`. We cannot use `(op *R)` to do that. The most similar operation presented in the HMA library is: `(op *s) :: 'a⇒'a^'n⇒'a^'n`.

We cannot reuse it because it is thought to multiply a vector (and not a matrix) by a scalar. Then, we define the multiplication of a matrix by a scalar as follows:

```
definition matrix_scalar_mult :: "'a ⇒ 'a^'n^'m ⇒ 'a^'n^'m"
(infixl "*k" 70) where "k *k A ≡ (χ i j. k * A $ i $ j)"
```

The statements for the real matrix version and the general one are different:

```
lemma scalar_matrix_vector_assoc:
  fixes A :: "real^'m^'n"
```

[1] We followed the proof presented in `http://hobbes.la.asu.edu/courses/site/442-f09/dets.pdf`.

```
  shows "k *R (A *v v) = (k *R A) *v v"

 lemma scalar_matrix_vector_assoc:
  fixes A :: "'a::{field}^'m^'n"
  shows "k *s (A *v v) = (k *k A) *v v"
```

Some other particularities arose in the generalisation. For instance, we had to completely change another demonstration: the row rank and the column rank of a matrix are equal. We had followed an elegant proof but only valid for real matrices, see [110]. We based its generalisation on the output of the Gauss-Jordan algorithm (a reduced row echelon form) following [1]. This change forced us to completely reorganise the files of our development. Another example arises in systems of linear equations: in the real field there could be infinite solutions, but in other fields such as $\mathbb{Z}_2$ there is always finitely many solutions.

Finally, we have generalised more than 2500 lines of code: about 220 theorems and 9 definitions, introducing 6 new locales, 3 new sublocales and 8 new interpretations. The generalised version of the Gauss-Jordan formalisation was published in the AFP [54]. Moreover, the generalisations are also useful for the Rank-Nullity theorem [52], as it has already been explained in Section 4.2. The final generalised statements of this theorem are the following ones:

```
theorem rank_nullity_theorem:
  shows "V.dimension = V.dim {x. f x = 0} + W.dim (range f)"

lemma rank_nullity_theorem_matrices:
  fixes A::"'a::{field}^'cols::{finite, wellorder}^'rows"
  shows "ncols A = vec.dim (null_space A) + vec.dim (col_space A)"
```

Let us note that the Rank-Nullity theorem has been proven in a context where a linear map `f` between a finite-dimensional vector space `V` and an arbitrary vector space `W` is fixed. Thus, the Rank-Nullity theorem has been formalised in the most general case.

4.4.2 Conclusions

The generalisation of the HMA library is useful and desirable, but doing it can be overwhelming at a first glance. The process can be partially automated with suitable scripts, but the full goal cannot be discharged automatically and it requires to make some design decisions. The careful combination of locales, type classes and interpretations has been shown to be a sensible methodology. A remarkable number of proofs have been reused in this way. This contribution shows that the aim is feasible and the generalisation has served for our initial purposes of executing a verified version of the Gauss-Jordan algorithm over fields such as $\mathbb{Z}_2$ and $\mathbb{Q}$.

4.5 The QR Decomposition

4.5.1 Introduction

Interactive theorem proving is a field in which impressive problems are being challenged and overcome successfully. Still, new challenges usually require

an accordingly impressive previous infrastructure to succeed (for instance, the SSReflect extension created for the four colour theorem [70] and the odd order number theorem [72], and the Simpl imperative language and the AutoCorres translator which are keystones in the seL4 verification [99]). This infrastructure, once developed, shall be reusable and applicable enough to overcome new challenges. Even if some developments or libraries reach a status of "keystones", and new projects are regularly built on top of them, this design principle does not always hold. Blanchette *et al.* [30] presented a survey about the reutilisation of the AFP entries. Directly quoting the conclusions of this survey, "There is too little reuse to our taste; the top 3 articles are reused 9, 6 and 4 times."; we assume that these conclusions can be spread to theorem provers as a whole. In this section we present various pieces of work which take great advantage of previously developed tools (either well-established parts of the Isabelle library or 10 existing developments of the AFP, including our infrastructure presented in Chapter 3 and the formalisation of the Gauss-Jordan algorithm presented in Section 4.3) to fulfil in an affordable number of lines a complete work in Linear Algebra, which includes the formalisation of the Fundamental Theorem of Linear Algebra, the Gram-Schmidt process, the QR decomposition, and its application to the least squares problem.

Hence, this work can be divided into four different parts. Firstly, we introduce the formalisation of Theorem 7, a result called by Strang the *Fundamental Theorem of Linear Algebra* (see [141, 142]), which establishes the relationships between the dimensions and bases of the four fundamental subspaces (the row space, column space, null space, and left null space) associated to a given linear map between two finite-dimensional vector spaces. This theorem is also closely tied to the notion of *orthogonality* of subspaces. The notion is already present in the HMA library, but related concepts such as the projection of a vector onto a subspace are not. Secondly, we formalise the Gram-Schmidt process, which permits to obtain an *orthogonal set of vectors* from a given set of vectors. Gram-Schmidt possesses remarkable features, such as preserving the spanning set of collections of vectors and providing *linearly independent* vectors, whose formalisation we present. Thirdly, as a natural application of the Gram-Schmidt process we implement the QR decomposition of a matrix into the product of two different matrices (the first one containing an *orthonormal* collection of vectors, the second one being upper triangular). We formalise the relevant properties of this decomposition reusing some of the previous work that we presented in Chapter 3 and that was successfully applied in the formalisation of the Gauss-Jordan algorithm (Section 4.3 and [13]). Fourthly, we formalise the application of the QR decomposition of a matrix to compute the *least squares approximation* to an unsolvable system of linear equations, exploiting some of the infrastructure for *norms* and distances in the HMA library, and reusing the computation of solutions to consistent systems of linear equations provided by the *Gauss-Jordan* algorithm.

Finally, and based on our previous infrastructure, we present examples of execution of the least squares problem (which internally uses Gram-Schmidt and QR decomposition) inside Isabelle and also from the code generated to SML. More particularly, taking advantage of an existing development in Isabelle for *symbolically* computing with $\mathbb{Q}$ extensions of the form $\mathbb{Q}[\sqrt{b}]$ by Thiemann [145], exact symbolic computations of the QR decomposition, and thus of the least squares solution of systems of linear equations are performed. We also present

some optimisations performed over the original algorithm to improve its performance, and explore its numerical properties, in comparison to the Gauss-Jordan algorithm. Consequently, this work completes our previous development about the Gauss-Jordan algorithm and its applications (Section 4.3) where the computation of the solution to systems of linear equations was formalised (for systems with unique solution, multiple solutions and without solution), computing also the *least squares approximation* to systems without solution.

In Computational Linear Algebra, the *raison d'être* of the QR decomposition is that it significantly reduces round-off errors when computing the least squares approximation (see for instance Björck [29] for details). Additionally, the decomposition comprises a valuable method to approximate the eigenvalues of a matrix, by successively computing QR decompositions [63, 100]; this method approximates the eigenvalues of the original matrix rather quickly and accurately, especially when compared to other known methods (it has been declared "one of the top ten algorithms with the greatest influence on the development and practice of science and engineering in the 20th century" [58]). The previous applications make QR decomposition more appealing than the conventional LU decomposition in that particular use cases. In this work we do not directly tackle numerical considerations (even if we introduce some examples of execution with floating-point numbers in Subsection 4.5.6.1), but we refer the interested reader to the works by Björck *et al.* [29, 43].

In Linear Algebra the QR decomposition is of interest by itself because of the insight that it offers about the vector subspaces of a linear map (it provides orthonormal bases of the four fundamental subspaces). We point the interested reader to the work by Strang [142, Chap. 4] for further information and applications. The method of *least squares* is usually attributed to Gauss, who already had a solution to it in 1795 that successfully applied to predict the orbit of the asteroid Ceres in 1801 (see [29]).

This section is divided as follows. In Subsection 4.5.2 we introduce the formalisation of the Fundamental Theorem of Linear Algebra. In Subsection 4.5.3 we present our formalisation of the Gram-Schmidt process. Subsection 4.5.4 presents the formalisation of the QR decomposition. Subsection 4.5.5 shows the application of the QR decomposition to the *least squares approximation*. In Subsection 4.5.6 we describe code generation of the aforementioned algorithms, introduce some code optimisations to improve performance, and present some examples of execution. Subsection 4.5.7 presents a brief survey of related formalisations. Finally, in Subsection 4.5.8 we present some conclusions of the completed work. The development is available from the Isabelle Archive of Formal Proofs [14] and it relies upon 10 AFP articles.

4.5.2 The Fundamental Theorem of Linear Algebra

In Subsection 2.1.2 we introduced Theorem 7, which is named as the Fundamental Theorem of Linear Algebra (see [142]). Both the *Gram-Schmidt* algorithm and the QR decomposition are built upon the notion of *orthogonality*, a concept which is also important for some items in Theorem 7. This notion requires a new type class based upon vector spaces, named in the HMA library `real_inner` (which describes an inner product space over the reals). It introduces an *inner* or dot product, which is then used to define the *orthogonality* of vectors.

```
context real_inner
begin
 definition "orthogonal x y ⟷ x · y = 0"
end
```

Item 1 in Theorem 7 is usually labelled as the *Rank-Nullity theorem*, and we completed its formalisation generalised to matrices over fields (see Section 4.2). From an existing basis of the null space and its completion up to a basis of F^m, it is proven that the dimension of the column space is equal to r. Then, the column space is proved equal to the range by means of algebraic rewriting.

In order to prove Item 2 in Theorem 7, we apply again the *Rank-Nullity theorem* to A^T. Additionally, it must be proven that the dimension of the row space is equal to the rank of A. This particular proof, for matrices over generic fields, involves the computation of the reduced row echelon form (or *rref*) of A, and it requires reusing our formalisation of the *Gauss-Jordan algorithm* (Section 4.3). The key idea is to prove that elementary row operations preserve both the row rank and the column rank of A, and then to compare the row and column ranks of the rref of A, concluding that they are equal. We describe this proof in Subsection 4.4. Up to now, proofs have been carried out in full generality (for matrices over a generic field F).

Items 3 and 4 in Theorem 7 claim that the *row space* and the *null space* of a given linear map are orthogonal complements, and so are the *column space* and the *left null space*. The *orthogonal complement* of a subspace W is the set of vectors of V orthogonal to every vector in W (note that, following the HMA library, the notion of orthogonality already places us in *inner product spaces over* $\mathbb{R}$):

```
definition "orthogonal_complement W = {x. ∀y∈W. orthogonal x y}"
```

Since the definition of the null space claims that this space is equal to the x such that $A\,x = 0$ and the row space of A is the one generated by the rows of A, both spaces are proven to be *orthogonal*; a similar reasoning over A^T proves that the left null space and the column space are also *orthogonal*.

```
lemma complementary_subspaces:
  fixes A::"real^'cols::{finite,wellorder}^'rows::{finite,wellorder}"
  shows "left_null_space A = orthogonal_complement (col_space A)"
    and "null_space A = orthogonal_complement (row_space A)"
```

Note that the *Rank-Nullity theorem* is the key result to prove that the fundamental subspaces are *complementary*. The definitions and proofs introduced in this section can be found in the files *Fundamental_Subspaces.thy* and *Least_Squares_Approximation.thy* (where the *Rank-Nullity theorem* is already incorporated to the development) of our development [14]. Thanks to the intensive reuse of these definitions and results, the complete proof of Theorem 7 took us 80 lines of Isabelle code.

As a matter of experiment, we tried to generalise the notion of *inner product space over* $\mathbb{R}$ to that of *inner product space over a field* F, and then replay the proof of Theorem 7. This generalisation can be found in file *Generalizations2.thy* of our development [14] and it follows the same approach as the one presented

in Section 4.4. The number of lines devoted to define the required notions, state Items 3 and 4 of Theorem 7 in Isabelle and prove them in full generality was *ca.* 650. We can now obtain explicit statements for concrete instances of inner product spaces over other fields than $\mathbb{R}$, such as the version of Items 3 and 4 in Theorem 7 involving complex numbers:

```
theorem left_null_space_orthogonal_complement_col_space_complex:
fixes A::"complex^'cols::{finite,wellorder}^'rows::{finite,wellorder}"
shows "left_null_space A = complex_matrix.orthogonal_complement
(col_space (χi j. cnj (A $ i $ j)))"

lemma null_space_orthogonal_complement_row_space_complex:
fixes A::"complex^'cols::{finite,wellorder}^'rows::{finite,wellorder}"
shows "null_space A = complex_matrix.orthogonal_complement (row_space
(χi j. cnj (A $ i $ j)))"
```

Let us note that the statements presented above together with the generalised version of the Rank-Nullity theorem presented in Section 4.4 constitutes a formalisation of the Fundamental Theorem of Linear Algebra in full generality (Items 1 and 2 involving matrices over a field, Items 3 and 4 involving inner product spaces over a field). The use of arbitrary inner product spaces requires using `cnj` (the conjugate element) in the statements (in the concrete case of real numbers, `cnj` is just the identity, obtaining the Isabelle statement `complementary_subspaces`).

Being the generalisation of the results presented in this work to inner product spaces over a field a sensible and interesting work, we stick in this section to *inner product spaces over* $\mathbb{R}$ since this decision gives us the chance to reuse the results in the HMA library, instead of starting from scratch and reproducing them in full generality.

4.5.3 A Formalisation of the Gram-Schmidt Algorithm

In this subsection we introduce the Gram-Schmidt process that leads to the computation of an orthogonal basis of a vector space and its formalisation. Let us note that *orthonormal* vectors are *orthogonal* vectors whose norm is equal to 1. Sets of orthogonal vectors play a crucial role in Linear Algebra. For instance, thanks to the *orthogonality* of the row and null spaces, every vector $x \in \mathbb{R}^m$ can be decomposed as $x = x_r + x_n$, where x_r belongs to the row space of A and x_n belongs to the null space of A, and therefore $x_r \cdot x_n = 0$. Now, $A\,x = A\,(x_r + x_n)$ and this is equal to $A\,x_r$ (this hint will be crucial for the *least squares approximation* of systems of linear equations that we describe in Subsection 4.5.5). Another relevant concept is the *projection* of a vector v onto a vector u, and that of the projection onto a set. Our Isabelle definitions follow (they can be found in file *Projections.thy* of our development [14] together with some of their relevant properties):

```
definition "proj v u = (v · u / (u · u)) *R u"
definition "proj_onto a S = setsum (λx. proj a x) S"
```

The Gram-Schmidt process takes as input a (finite) set of vectors $\{v_1 \ldots v_k\}$ (which need not be linearly independent, neither be a set with size less than or

equal to the dimension of the underlying vector space) and iteratively subtracts from each of them their *projection* onto the previous ones. The process can be implemented in Isabelle as follows. The following definition takes a vector a and a list of vectors ys and subtracts from a its projections onto the vectors of ys. The obtained vector will be *orthogonal* to every vector in the input list ys. Note that we have replaced *sets of vectors* by *lists of vectors* for simplicity (`op @` denotes the appending operation on lists); a similar process could be applied to finite indexed sets:

```
definition "Gram_Schmidt_step a ys = ys @ [a - proj_onto a (set ys)]"
```

This *step* is *folded* over a list of vectors xs and the empty list, obtaining thus a list of vectors whose projections onto each other are 0 (*i.e.*, are orthogonal).

```
definition "Gram_Schmidt xs = foldr Gram_Schmidt_step xs []"
```

The defined function `Gram_Schmidt` satisfies two properties. First, the vectors in its output list must be *pairwise orthogonal* (which means that any two different vectors in the output list must be orthogonal):

```
lemma Gram_Schmidt_pairwise_orthogonal:
  fixes xs::"('a::{real_inner}^'b) list"
  shows "pairwise orthogonal (set (Gram_Schmidt xs))"
```

Second, the *span* of the sets associated to both the output and input lists must be equal (note that here the definition `real_vector.span` does not require the underlying scalar product, as it was the case with `vector_space.span`):

```
lemma Gram_Schmidt_span:
  fixes xs::"('a::{real_inner}^'b) list"
  shows "real_vector.span (set (Gram_Schmidt xs)) = real_vector.span
(set xs)"
```

The proofs of the properties `Gram_Schmidt_pairwise_orthogonal` and `Gram_Schmidt_span` are carried out by induction over the input list. Under these two conditions, whenever the input list is a basis of the vector space, the output list will also be a basis (the predicate `distinct` is used to assert that there are no repeated vectors in the input and output lists).

```
corollary orthogonal_basis_exists':
  fixes V :: "(real^'b) list"
  assumes B: "is_basis (set V)" and d: "distinct V"
  shows "is_basis (set (Gram_Schmidt V)) ∧ distinct (Gram_Schmidt V)
  ∧ pairwise orthogonal (set (Gram_Schmidt V))"
```

As a previous step for the QR decomposition of matrices, we introduced a definition of the *Gram-Schmidt* process directly over the *columns of a matrix*. To get that, the above operation `Gram_Schmidt` could be applied to the list of columns of the matrix (indeed, that was our first version), but that would require two conversions between matrices and lists. Instead, in order to improve efficiency, we have preferred to build a new matrix from a function, using the χ binder (the morphism defining a `vec` from a function).

The operation `Gram_Schmidt_column_k` returns a matrix where *Gram-Schmidt* is performed over column `k` and the remaining columns are not changed. This operation is then folded over the list of the input matrix columns. Note that `k` is a natural number, whereas the rows and columns indexes are elements of the finite types introduced in Subsection 3.2.1, and thus the operation `from_nat` is applied to convert between them.

```
definition "Gram_Schmidt_column_k A k = (χ a b. (if b = from_nat k
  then (column b A - (proj_onto (column b A) {column i A|i. i < b}))
  else (column b A)) $ a)"
definition "Gram_Schmidt_upt_k A k = foldl Gram_Schmidt_column_k A
[0..<k+1]"
definition "Gram_Schmidt_matrix A = Gram_Schmidt_upt_k A (ncols A - 1)"
```

The definition of `Gram_Schmidt_matrix` has been proven to satisfy similar properties to `Gram_Schmidt`. Additionally, both definitions have been set up to enable code generation and execution from Isabelle to both SML and Haskell.

The following expression can be now evaluated in Isabelle. In this setting, the function `Gram_Schmidt_matrix` is being evaluated. In Subsection 4.5.6 we will improve its performance using a refinement of these functions to *immutable arrays*, following the infrastructure developed in Chapter 3:

```
value "let A = list_of_list_to_matrix [[4,-2,-1,2], [-6,3,4,-8],
[5,-5,-3,-4]]::real^4^3 in matrix_to_list_of_list (Gram_Schmidt_matrix
A)"
```

The obtained result is:

```
"[[4,50/77,15/13,0], [-6,-75/77,10/13,0], [5,-130/77,0,0]]"
```

Note that the output vectors are *orthogonal*, but not *orthonormal*. We address this issue in the next subsection, when formalising the QR decomposition. The formalisations presented in this subsection are available in the file *Gram_Schmidt.thy* from [14].

4.5.4 A Formalisation of the QR Decomposition Algorithm

The QR decomposition of a matrix A is defined as a pair of matrices, $A = QR$, where Q is a matrix whose columns are *orthonormal* and R is an *upper triangular* matrix (which in fact contains the elementary column operations that have been performed over A to reach Q). The literature includes different variants of this decomposition (see for instance [29, Chapt. 3 and 4]). More concretely, it is possible to distinguish two different decompositions of a given matrix $A \in M_{m\times n}(\mathbb{R})$:

1. If A has *full column rank*, A can be decomposed as QR, where $Q \in M_{m\times n}(\mathbb{R})$ and its columns are orthonormal vectors, and $R \in M_{n\times n}(\mathbb{R})$ is an upper triangular and invertible matrix.

2. A can be also decomposed as QR, where $Q \in M_{m\times m}(\mathbb{R})$ and is orthonormal, and $R \in M_{m\times n}(\mathbb{R})$ is an upper triangular (but neither square, nor

invertible) matrix. This case is called *full QR decomposition*.

In this work we formalise the first case, where the number of rows of A will be greater than or equal to the number of columns. Indeed, this is the version of the decomposition which is directly applicable to solve the *least squares problem*, as we explain in Subsection 4.5.5. In the particular case where A has no full column rank, we do not compute the QR decomposition, but, as we present in Subsection 4.5.5, we solve the problem by means of the *Gauss-Jordan* algorithm. Let us describe how the decomposition is performed.

Given a matrix $A = (a_1 \mid \cdots \mid a_n) \in M_{m\times n}(\mathbb{R})$ (where $n \leq m$) whose columns a_i are linearly independent vectors, the matrix $Q \in M_{m\times n}(\mathbb{R})$ is the matrix with columns $(q_1 \mid \cdots \mid q_n)$, where q_i is the normalised vector a_i minus its projections onto $q_1 \ldots q_{i-1}$ (and thus, *orthogonal* to both $a_1 \ldots a_{i-1}$ and $q_1 \ldots q_{i-1}$). The matrix $R \in M_{n\times n}(\mathbb{R})$ can be expressed as $R = Q^T A$.

Once we have computed the *Gram-Schmidt* process over the vectors of a matrix in Subsection 4.5.3 (recall the Isabelle function `Gram_Schmidt_matrix`), and introducing an operation to *normalise* every column in a matrix, the computation of the QR decomposition is defined in Isabelle as follows:

```
definition "divide_by_norm A  = (χ a b. normalize (column b A) $ a)"
definition "QR_decomposition A =
 (let Q = divide_by_norm (Gram_Schmidt_matrix A)
  in (Q, (transpose Q) ** A))"
```

The literature suggests some other ways to compute the matrices Q and R, in such a way that the coefficients of matrix R are computed in advance, and then used in the computation of the columns of Q; see for instance the algorithms labelled as *Classical Gram-Schmidt* and *Modified Gram-Schmidt* by Björck [29, Chap. 2.4]. These algorithms avoid some unnecessary operations in our Isabelle formalisation (in particular, they avoid the computations of Q^T and the product $Q^T A$). Instead, our formalised version directly uses the output of the Gram-Schmidt orthogonalisation process presented in Subsection 4.5.3 and computes *a posteriori* the coefficients in R.

The properties of Q and R need to be proved. Once that in Subsection 4.5.3 we proved that the columns of the matrix computed with `Gram_Schmidt_matrix` are pairwise orthogonal and that they have a span equal to the one of the input matrix, these properties are straightforward to prove for Q. The property of the columns of Q having norm equal to 1 is proven also directly from the definition of Q. For its intrinsic interest we illustrate the property of Q and A having equal *column space*:

```
corollary col_space_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  shows "col_space A = col_space Q"
```

Another crucial property of Q (and Q^T) that is required later in the least squares problem is the following one (note that it is stated for possibly *non-square* matrices with more rows than columns):

```
lemma orthogonal_matrix_fst_QR_decomposition:
```

```
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "transpose Q ** Q = mat 1"
```

This property is *commutative* for square matrices ($Q^TQ = QQ^T = I_n$ and thus $Q^{-1} = Q^T$) but it does not hold that $QQ^T = I_m$ for *non-square* ones. Its proof is completed by case distinction in the matrix indexes; being $Q = (q_1 \mid \cdots \mid q_n)$, and thus $Q^T = \begin{pmatrix} q_1 \\ \cdots \\ q_n \end{pmatrix}$, when multiplying row i of Q^T (which is q_i) times column i of Q, the result is 1 since the vectors are *orthonormal*. On the contrary, when multiplying row i of Q^T (which is q_i) times column j of Q, the result is 0 because of *orthogonality*.

Then, the most relevant properties of R are being upper triangular and invertible. Indeed, being $A = (a_1 \mid \cdots \mid a_n)$, $R = Q^TA = \begin{pmatrix} q_1 \cdot a_1 & q_1 \cdot a_2 & \cdots \\ q_2 \cdot a_1 & q_2 \cdot a_2 & \cdots \\ \cdots & \cdots & \cdots \end{pmatrix}$. The following lemma proves the matrix R being upper triangular:

```
lemma upper_triangular_snd_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "upper_triangular R"
```

The matrix R is also *invertible*:

```
lemma invertible_snd_QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "invertible R"
```

The properties satisfied by the QR decomposition (in this statement, of non-square matrices) can be finally stated in a single result (the result for square matrices of size n also proves the columns of Q being a *basis* of $\mathbb{R}^n$). The result sums up the properties of Q and R that have been formalised along Subsections 4.5.3 and 4.5.4:

```
lemma QR_decomposition:
  fixes A::"real^'n::{mod_type}^'m::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "A = Q ** R ∧
      pairwise orthogonal (columns Q) ∧ (∀ i. norm (column i Q) = 1) ∧
      (transpose Q) ** Q = mat 1 ∧ vec.independent (columns Q) ∧
      col_space A = col_space Q ∧ card (columns A) = card (columns Q) ∧
      invertible R ∧ upper_triangular R"
```

The formalisations carried out in this subsection are available in the file *QR_Decomposition.thy* from our development [14].

4.5.5 Solution of the Least Squares Problem

The previous decomposition can be used to different aims. In this work we focus on finding the best approximation of a system of linear equations *without solution*. In this way, we complete our previous work presented in Section 4.3, in which the computation of the solution of systems of linear equations was formalised thanks to the Gauss-Jordan algorithm.

The *best approximation* of a system $Ax = b$, in this setting, means to find the elements $\hat{x}$ such that minimise $\|e\|$, where $e = A\hat{x} - b$. Our aim is to prove that $\hat{x}$ is the solution to $A\hat{x} = \hat{b}$, where $\hat{b}$ denotes the projection of b onto the column space of A. The solution for the general case (also known as the *rank deficient case*) is usually performed by means of the *Singular Value Decomposition* (or SVD); this decomposition provides, for any real or complex matrix A, three matrices U, Σ, V such that $A = U\Sigma V^H$ (where V^H is the result of conjugating each element of V and then transposing the matrix, and $\Sigma = \left(\begin{smallmatrix}\Sigma_1 & 0\\ 0 & 0\end{smallmatrix}\right)$, where $\Sigma_1 = \mathrm{diag}\,(\sigma_1 \ldots \sigma_r)$ and σ_i denotes the singular values of A, in such a way that $A = \sum_{i=1}^{n} \sigma_i u_i v_i^H$).

The existence of the SVD decomposition of a matrix can be proven by induction without particular difficulties (see [29, Th. 1.2.1]). On the contrary, the computation of the SVD decomposition (see, for instance, [29, Sect. 2.6]) requires the computation of eigenvalues and eigenvectors of matrices, whose computation requires numerical methods. In this work we solve the case where the input matrix A of the system $A\,x = b$ has *full column rank* by means of the QR decomposition, and the general case will be solved applying the Gauss-Jordan algorithm.

We define the characterisation of the least squares solution of a system as follows (following [29, Th. 1.1.2]):

```
definition "set_least_squares_solution A b =
   {x. ∀y. norm (b - A *v x) ≤ norm (b - A *v y)}"
```

Prior to showing the utility of the QR decomposition to solve the previous problem, we prove that the closest point to a point $v \notin S$ in a subspace S (being X an orthogonal basis of S) is its projection onto that subspace:

```
lemma least_squares_approximation:
  fixes X::"'a::{euclidean_space} set"
  assumes "real_vector.subspace S" and "real_vector.independent X"
  and "S = real_vector.span X"
  and "pairwise orthogonal X"
  and "proj_onto v X ≠ y"
  and "y ∈ S"
  shows "norm (v - proj_onto v X) < norm (v - y)"
```

The lemma `least_squares_approximation` states that the projection of b onto the range of A (that we denote by $\hat{b}$) is the closest point to b in this subspace. Let $\hat{x}$ be such that $A\,\hat{x} = \hat{b}$. Thanks to Theorem 7, $b - A\,\hat{x}$ belongs to the orthogonal complement of range A, which happens to be the *left null space*. Consequently,

we know that the solutions to the least squares problem must satisfy the equation $A^T(b - A\,x) = 0$ (the converse also holds). From this property, the standard characterisation of the set of least squares solutions is obtained [29, Th. 1.1.2]:

```
lemma in_set_least_squares_solution_eq:
  fixes A::"real^'cols::{finite,wellorder}^'rows"
  defines "A_T == transpose A"
  shows "(x ∈ set_least_squares_solution A b) =
         (A_T ** A *v x = A_T *v b)"
```

The proof of lemma `least_squares_approximation` makes use of the Pythagorean Theorem of real inner product spaces, whose proof we include because of its intrinsic interest:

```
lemma Pythagorean_theorem_norm:
  assumes o: "orthogonal x y"
  shows "norm (x+y)^2=norm x^2 + norm y^2"
proof -
  have "norm (x+y)^2 = (x+y) · (x+y)" unfolding power2_norm_eq_inner ..
  also have "... = ((x+y) · x) + ((x+y) · y)"
    unfolding inner_right_distrib ..
  also have "... = (x · x) + (x · y) + (y · x) + (y · y) "
    unfolding real_inner_class.inner_add_left by simp
  also have "... = (x · x) + (y · y)" using o unfolding orthogonal_def
    by (metis comm_monoid_add_class.add.right_neutral inner_commute)
  also have "... = norm x^2 + norm y^2"
    unfolding power2_norm_eq_inner ..
  finally show ?thesis .
qed
```

Once we have characterised the set of least squares solutions, we distinguish whether A has full column rank or not:

- If A has no full column rank, A^TA does not have inverse, and the solution to the least squares problem can be obtained by applying the Gauss-Jordan algorithm (that we formalised in Section 4.3) to the system $A^TA\hat{x} = A^Tb$ ([29, Eq. 1.1.15]). Our Gauss-Jordan implementation would compute a single solution of the system plus a basis of the null space of A^TA:

  ```
  lemma in_set_least_squares_solution_eq:
    fixes A::"real^'cols::{finite,wellorder}^'rows"
    defines "A_T == transpose A"
    shows "(x ∈ set_least_squares_solution A b) =
           (A_T ** A *v x = A_T *v b)"
  ```

- Otherwise, A^TA is an invertible matrix, and $\hat{x} = (A^TA)^{-1}A^Tb$. In this case, the solution is *unique*, and the set in the right hand side is a singleton. The following result proves the uniqueness and the explicit expression of the solution [29, Eq. 1.1.16]:

  ```
  lemma in_set_least_squares_solution_eq_full_rank:
    fixes A::"real^'cols::mod_type^'rows::mod_type"
  ```

```
  defines "A_T ≡ transpose A"
  assumes r: "rank A = ncols A"
  shows "(x ∈ set_least_squares_solution A b) =
           (x = matrix_inv (A_T ** A) ** A_T *v b)"
```

As it may be noticed, the solution to the *least squares problem* does not demand the QR decomposition of A. The decomposition is used when A is an (full column rank) almost singular matrix (*i.e.*, its condition number, σ_1/σ_r is "big", and the computation of $(A^TA)^{-1}$ seriously compromises floating-point precision). Even if numerical methods are not central to our aim, we point the interested reader to the works by Björck [29, Sect. 1.4] or [42, Sect. 2.4].

Since $\hat{x} = (A^TA)^{-1}A^Tb$, and using that $A = QR$ and $A^T = R^TQ^T$ (note that $Q^TQ = I$), one also has $\hat{x} = (R^TQ^TQR)^{-1}R^TQ^Tb$, and this equation can be reduced to $\hat{x} = R^{-1}Q^Tb$. Now, the matrices Q and R are obtained through the Gram-Schmidt process, and the inverse of R, which is upper triangular, can be performed by backward substitution. The Isabelle statement of this new equality follows [29, Th. 1.3.3]:

```
corollary in_set_least_squares_solution_eq_full_rank_QR2:
  fixes A::"real^'cols::{mod_type}^'rows::{mod_type}"
  defines "Q ≡ fst (QR_decomposition A)"
  and "R ≡ snd (QR_decomposition A)"
  assumes r: "rank A = ncols A"
  shows "(x ∈ set_least_squares_solution A b) =
           (x = matrix_inv R ** transpose Q *v b)"
```

The formalisations of the results in this subsection are available in the file *Least_Squares_Approximation.thy* from our development [14]. In the next subsection we show how the previous results can be used to *compute* the least squares solution of a linear system.

4.5.6 Code Generation from the Development

Up to now we have proved that given a matrix $A \in M_{m\times n}(\mathbb{R})$ and a system of linear equations $Ax = b$ without solution there exists one or multiple least squares approximations to that system, and we have also provided and proved explicit expressions to identify them. In the case where A has no full column rank, computing the solutions requires solving the system $A^TA\hat{x} = A^Tb$; when A has full column rank, the solution can be directly computed by means of the expression $\hat{x} = R^{-1}Q^Tb$.

The computation of the solutions, based on the previous expressions, requires various features.

- First, the underlying `real` type (and the required operations) needs an *executable* representation.
- Then, the representation (and the operations) chosen for matrices needs an *executable* version.
- For the case where A has no full column rank, an executable version of the Gauss-Jordan algorithm applied to compute the solution of systems of linear equations needs to be provided.

As we have already explained in Chapter 3, the first point admits various solutions. If we seek execution *inside* of Isabelle, we can rely on the `real` type definition and its operations, which have been already set up to be executable. In this setting, exact real computations are obtained. Unfortunately, Isabelle is not specially designed as an execution environment, and the performance obtained is rather poor. As a matter of example, computing the Gram-Schmidt process of 10 vectors in $\mathbb{R}^{10}$ (that is, apply `Gram_Schmidt_matrix` over a 10×10 real matrix) in Isabelle consumes various minutes, whereas this same computation is immediate using the code generated to SML or Haskell.

Therefore, our alternative solution consists in making use of the aforementioned Isabelle code generation facility (see [78,80] and Section 2.2.3) that translates Isabelle specifications to specifications in various functional languages (we make use of SML and, in some particular cases, Haskell in this development). Nevertheless, as it was explained in Chapter 3, the type `real` is generated by default to quotients of rational numbers, obtaining again a poor performance. Additionally, square roots computations are not possible in this setting (only `Gram_Schmidt` could be executed).

As in the Gauss-Jordan formalisation, we can also reuse the serialisations presented in Section 3.4, from the Isabelle type `real` to SML and Haskell native types. These serialisations allow us to *compute* in SML and Haskell the formalised algorithm of the least squares problem, but computations fall in the conventional rounding errors of double-precision floating-point numbers (despite the original algorithm being formalised, the computations cannot be trusted). We present an example of this possibility in Subsection 4.5.6.1.

Fortunately, as we were completing this work, an Isabelle development by Thiemann [145] was published in the Isabelle Archive of Formal Proofs. This development provides, among many other interesting features, a data type refinement for real numbers of the form $p + q\sqrt{b}$ (with $p, q \in \mathbb{Q}$, $b \in \mathbb{N}$). In other words, the refinement provides *symbolic computation* for that range of numbers. This refinement is available for computations inside of Isabelle, and also for code generation to both SML and Haskell. We make use of this development, and obtain exact symbolic computations, as long as we restrict ourselves to matrices whose inputs are in $\mathbb{Q}$ (if we input matrices in $\mathbb{Q}[\sqrt{b}]$ their normalisation may belong to $\mathbb{Q}[\sqrt{b}][\sqrt{a}]$, that is out of the scope of the presented Isabelle development).

The second concern to obtain computations is the *representation of matrices*. The representation we have used along the formalisation is the one presented in the HMA library: it relies on a type `vec` representing vectors (and then its iterated construction to represent matrices) which corresponds to functions over finite domains. Again, we have reused the infrastructure presented in Chapter 3 in order to refine the algorithm to immutable arrays. Then, we have reproduced in Isabelle the definitions of the QR decomposition for immutable arrays and proved their equivalence with respect to the vector versions. These lemmas are then labelled as `code` lemmas and are used in the evaluation and code generation processes.

A significant difference appears here with respect to the use of immutable arrays and the Gauss-Jordan algorithm presented in Section 4.3 (actually, even compared to any other algorithm presented in this thesis). In the QR decomposition, we have made use of the second possibility explained in Section 3.3 in order to implement the addition of two immutable arrays:

```
definition plus_iarray :: "'a iarray ⇒ 'a iarray ⇒ 'a iarray"
  where "plus_iarray A B =
  (let length_A = (IArray.length A);
  length_B= (IArray.length B);
  n=max length_A length_B ;
  A'= IArray.of_fun (λa. if a < length_A then A!!a else 0) n;
  B'=IArray.of_fun (λa. if a < length_B then B!!a else 0) n
  in
  IArray.of_fun (λa. A' !! a + B' !! a) n)"
```

This new definition is *commutative*, and thus it permits to show that *iarrays* over a commutative monoid is an instance of the Isabelle type class `comm_monoid_add`; then, several proofs involving *finite sums* (for instance, ranging over the columns of a matrix) of *iarrays* are simplified. On the other hand, this definition is more time consuming than the previous one, and it has some impact on performance since more operations are involved.

The previous setup, together with the refinement of *real numbers* to *field extensions* $\mathbb{Q}[\sqrt{b}]$, gives place to the following symbolic computations of the matrices Q and R in Isabelle (computations are internally being performed in SML transparently to the user; they can also be internally performed in Isabelle). Since symbolic computations are performed, matrix coefficients are then shown as strings. Do note that, once the refinement to *iarrays* has been performed, the operations internally being executed are the ones over *iarrays*:

```
definition "A ==
  list_of_list_to_matrix [[1,3/5,3],[9,4,5/3],[0,0,4],[1,2,3]]::real^3^4"
value "print_mat (fst (QR_decomposition A))"
value "print_mat (snd (QR_decomposition A))"
```

The results obtained follow:

```
"[["1/83*sqrt(83)",    "4/4233*sqrt(8466)",  "95/65229*sqrt(130458)"],
  ["9/83*sqrt(83)", "-11/8466*sqrt(8466)", "-19/130458*sqrt(130458)"],
  [            "0",                   "0",     "3/1279*sqrt(130458)"],
  ["1/83*sqrt(83)", "91/8466*sqrt(8466)",  "-19/130458*sqrt(130458)"]]"

"[["sqrt(83)", "193/415*sqrt(83)",       "21/83*sqrt(83)"],
  [       "0", "7/415*sqrt(8466)", "418/12699*sqrt(8466)"],
  [       "0",                "0",   "2/153*sqrt(130458)"]]"
```

We can also compute the *least squares approximation* to systems of equations with no solution. As we mentioned in Subsection 4.5.5, when $A = QR$ has full column rank, solving this problem requires computing the *inverse* of the matrix R, and this is done thanks to the *Gauss-Jordan* algorithm that we formalised in Section 4.3. An interesting situation shows up here, related to the use of $\mathbb{Q}[\sqrt{b}]$ extensions.[2] The solution to the least squares problem $Ax = b$ can be computed as $\hat{x} = R^{-1}Q^T b$.

[2]To clarify the notation: the b presented in $\mathbb{Q}[\sqrt{b}]$ and in $Ax = b$ are different. In the fist case b is a natural number. In the second case, b is a vector.

Given a matrix A, the computation of the matrix Q may involve the use of field extensions $\mathbb{Q}[\sqrt{b}]$, where b could be different in each column. Then, the computation of $R = Q^T A$ gives place to an upper triangular matrix (with each row in a possibly different extension of $\mathbb{Q}[\sqrt{b}]$), whose inverse is computed by means of elementary row operations, based on our implementation of the Gauss-Jordan algorithm.

The *least squares solution* $\hat{x}$ of a system $A\,x = b$ is computed symbolically as shown in the following example:

```
definition "b ≡ list_to_vec [1,2,3,sqrt(2)]::real^4"

value "let Q = fst (QR_decomposition A);
           R = snd (QR_decomposition A)
       in print_vec ((the (inverse_matrix R) ** transpose Q *v b))"
```

The computed solution (in negligible time, in SML) is `"["12269/17906 - 10443/35812 * sqrt(2)", "-11840/8953 + 5900/8953 * sqrt(2)", "1605/2558 - 57/5116 * sqrt(2)"]"`.

As we illustrate with the following computation, being $\hat{b} = A\,\hat{x}$, the difference $b - \hat{b}$ lies on the *left null space* of A, and therefore $A^T(b - \hat{b}) = 0$:

```
value "let Q = fst (QR_decomposition A); R = snd (QR_decomposition A);
           b2 = (A *v (the (inverse_matrix R) ** transpose Q *v b))
       in print_vec (transpose A *v (b - b2))"
```

Its output, as expected, is: `"["0", "0", "0"]"`.

Finally, we present the result of a problem related to the computation of the orbit of the comet Tentax [42, Ex. 1.3.4]:

```
value "let A = list_of_list_to_matrix
    [[1,-0.6691],[1,-0.3907],[1,-0.1219],[1,0.3090],[1,0.5878]]::real^2^5;
    b = list_to_vec [0.3704,0.5,0.6211,0.8333,0.9804]::real^5;
    QR = (QR_decomposition A); Q = fst QR; R = snd QR
    in print_vec (the (inverse_matrix R) ** transpose Q *v b)"
```

The obtained solution is `"["3580628725341/5199785740000", "251601193/519978574"]"`. It is obtained in SML thanks to the refinement to iarrays (computing time is again negligible).

The previous algorithms and computations have been performed by means of code generation to SML, using the refinement of vectors to `iarray` and the one of `real` to field extensions $\mathbb{Q}[\sqrt{b}]$, with $b \in \mathbb{N}$.

4.5.6.1 A Numerical Experiment

We have also explored a different possibility, which consists in refining the Isabelle type `real` to a type implementening the SML signature *Real*, *i.e.* floating-point numbers. The computations performed through this refinement cannot be considered *formalised* anymore, but we introduced them because they neatly illustrate the precision of the QR decomposition in comparison with, for instance, Gauss-Jordan (and therefore stress its utility), and also because they serve us to better illustrate the performance of the algorithm with floating-poing numbers.

In order to study the precision of the algorithm with floating-point numbers, we have used as input for the algorithm the *Hilbert matrices*, which are well-known for being ill-conditioned, and thus prone to round-off errors. Hilbert matrices are defined as:

$$H_{ij} = \frac{1}{i+j-1}$$

For instance, the Hilbert matrix in dimension 6, H_6 has determinant equal to 1/186313420339200000 and the order of magnitude of its condition number is 10^7:

```
[ [ 1 ,1/2,1/3,1/4,1/5,1/6],
  [1/2,1/3,1/4,1/5,1/6,1/7],
  [1/3,1/4,1/5,1/6,1/7,1/8],
  [1/4,1/5,1/6,1/7,1/8,1/9],
  [1/5,1/6,1/7,1/8,1/9,1/10],
  [1/6,1/7,1/8,1/9,1/10,1/11] ]
```

We have computed the least squares solution to the system $H_6\, x = (1\,0\,0\,0\,0\,0\,5)^T$ using both the QR decomposition and also the Gauss-Jordan algorithm. The exact solution of the least squares approximation can be obtained symbolically:

```
"["-13824","415170","-2907240","7754040","-8724240","3489948"]"
```

If we now use the refinement from Isabelle `real` to SML *floats*, and then use the QR decomposition and Gauss-Jordan to solve the least squares problem, the following solutions are obtained (they shall be compared to the exact solution presented above):

- QR solution using floats:

  ```
  [-13824.0,415170.0001,-2907240.0,7754040.001,-8724240.001,
  3489948.0]: real list
  ```

- Gauss-Jordan solution using floats:

  ```
  [-13808.64215,414731.7866,-2904277.468,7746340.301,-8715747.432,
  3486603.907]: real list
  ```

As it can be noticed, the rounding-off errors introduced by the Gauss-Jordan algorithm are several orders of magnitude greater than the ones by the QR decomposition. At the end of the following subsection, we present the performance obtained with this particular refinement.

4.5.6.2 Code Optimisations

The implementation of our first version of the QR algorithm admitted different types of performance optimisation that we also applied. Here we comment on three of them.

- First, there was an issue with the conversion from sets to lists in the code generation process. Let us recover the Isabelle definition introduced in Subsection 4.5.3, `proj_onto a S = setsum (λx. proj a x) S`. The definition applies an operation to the elements of a set `S` and then computes their sum. The Isabelle code generator is set up to refine sets to lists (whenever sets are finite), and thus the previous sum is turned into a list sum, by means of the following code equation (note that sums are abstracted to a generic "big operator" `F` defined for both sets and lists, and that we have omitted that the underlying structure is a commutative monoid):

```
lemma set_conv_list [code]:
 "set.F g (set xs) = list.F (map g (remdups xs))"
```

It is relevant to pay attention to the operation `remdups`; the input list `xs` that represents the set could contain duplicated elements, and therefore they have to be removed from that list for the equality to hold (for instance, the set $\{1, 2, 3\}$ can be originated by both $xs = [1, 2, 3, 3]$ and $xs = [1, 2, 3]$). When we applied code generation and by means of SML profiling techniques, we detected that `remdups` was one of the most time consuming operations in the QR executions. In our particular case, after applying `Gram_Schmidt_column_k` (this operation explicitly uses `proj_onto`, and therefore `remdups`, see Subsection 4.5.3) to the first k columns of a set, the obtained columns are either 0 or orthogonal. In the second case, there are no duplicated columns. Interestingly, in the first case, there might be duplicated columns equal to 0; these columns, when used in later iterations of `Gram_Schmidt_column_k`, do not affect the final result. In any case (with the previous columns being 0 or orthonormal), the following operation, that avoids removing duplicates, is more efficient than `Gram_Schmidt_column_k`, and returns the same result when applied to the column $k + 1$ of a matrix where `Gram_Schmidt_column_k` has been applied to the first k columns. The following definition (where `remdups` over the list of columns of the matrix has been avoided) is to be compared with the one of `Gram_Schmidt_column_k`):

```
definition "Gram_Schmidt_column_k_efficient A k = (χ a b.
  (if b = from_nat k
   then (column b A - listsum
     (map (λx.  ((column b A · x) / (x · x)) *R x)
     (map (λn. column (from_nat n) A) [0..<to_nat b]))
   else column b A) $ a)"
```

The proof of the equivalence between both definitions can be found in file *QR_Efficient.thy* [14]; let us remark that the property only holds for a column $k + 1$ when the first k columns have been already orthogonalised.

We have also used the standard strategy of providing code generation for sets as lists where duplicates are removed in the computation of inner products. By default, the inner product is computed over the *set* of indexes of the vectors (that are turned into lists to which `remdups` is applied, even when the set of indexes is known not to contain repeated elements). The

following code equation avoids this (in our case, unnecessary) check:

```
lemma [code]:
 "inner_iarray A B = listsum (map (λn. A!!n * B!!n)
{0..<IArray.length A})"
```

These changes divide by > 30 the running time of the QR decomposition algorithm applied to the Hilbert matrix H_{100} (whose dimension is 100×100).

- A second improvement on code performance was directly introduced by the Isabelle developers during the process of completing this work, and is related to the code generation setup of the function `map_range`, that profiling showed as another bottleneck of our programs execution. The function `map_range` is internally used to apply a function to a range of numbers (and therefore it is being used, for instance, in our previous definitions `inner_iarray` or `Gram_Schmidt_column_k_efficient`):

```
definition map_range[code_abbrev]: "map_range f n = map f [0..<n]"
```

The original code equation for this definition follows:

```
lemma map_range_simps [simp, code]:
 "map_range f 0 = []"
 "map_range f (Suc n) = map_range f n @ [f n]"
```

This definition, in each iteration, builds two different lists and concatenates them. The operation can be completed over a single list, improving both memory usage and performance. The previous definition of `map_range` was removed from the Isabelle library on lists, and the conventional definition of map over lists used instead:

```
lemma [code]:
  "map f [] = []"
  "map f (x # xs) = f x # map f xs"
```

This change permitted us to divide by a factor of 7 the amount of time used to compute the QR decomposition of the Hilbert matrix H_{100}.

- Finally, we also realised that the definition of `Gram_Schmidt` had room for improvement, from a computational point of view. The definition of `Gram_Schmidt_column_k_iarrays_efficient` can be replaced by the following one (they are extensionally equal):

```
definition "Gram_Schmidt_column_k_iarrays_efficient2 A k =
  tabulate2 (nrows_iarray A) (ncols_iarray A)
  (let col_k = column_iarray k A;
       col = (col_k - listsum
              (map (λx. ((col_k ·i x) / (x ·i x)) *R x)
              ((map (λn. column_iarray n A) [0..<k]))))
  in (λa b. (if b = k then col else column_iarray b A) !! a))"
```

This definition makes use of `let` definitions to bind variables (such as `col_k`) that in `Gram_Schmidt_column_k_iarrays_efficient` where being performed (indeed, they were simply access operations) many times. Proving the equivalence between both definitions is straightforward (the proof is lemma `Gram_Schmidt_column_k_iarrays_efficient_eq` in file *QR_Efficient.thy* of our development [14]).

This final change permitted us to divide by a factor of 8 the amount of time used to compute the QR decomposition of the Hilbert matrix H_{100}.

Adding up the previous optimisations (whose formalisations sums up 700 lines in file *QR_Efficient.thy*), the time required to compute the QR decomposition of the Hilbert matrix H_{100} was decreased by a factor of *ca.* 1500 (we used Hilbert matrices in this example because of their numerical instability; the computing time improvements shall be comparable for matrices of similar dimension).

The refinement from reals to symbolic computations that we have presented can be used in SML with matrices of sizes up to 10×10. In that sense, the refinement from Isabelle reals to floating-point numbers outperforms the symbolic version. The version over floating-point numbers can be applied to matrices of sizes up to 400×400 in reasonable time (*ca.* 3 minutes in a standard[3] laptop; see the complete benchmarks in Appendix A).

The development also allows further computations, such as the projection of a vector onto a subspace, the *Gram-Schmidt* algorithm, *orthogonality* of vectors, solution of the least squares problem for matrices without full rank, and can be used to grasp the geometrical implications of the *Fundamental Theorem of Linear Algebra*. The previous computations and some other carried out with the refinement to floating-point numbers can be found in files *Examples_QR_Abstract_Symbolic.thy*, *Examples_QR_IArrays_Symbolic.thy*, *Examples_QR_Abstract_Float.thy* and *Examples_QR_IArrays_Float.thy* of [14].

4.5.7 Related Work

Abstract Algebra is a common topic for the theorem proving community. Some milestones in this field have been already pointed out in Section 1.4.

Some relevant works in the Coq proof assistant are worth mentioning, such as the CoqEAL (standing for Coq Effective Algebra Library) effort by Dénès *et al* [44]. In Subsection 4.3.3.2 we presented a thorough comparison of our work and theirs, surveying also their most recent works [39]. Here we just emphasise that (up to our knowledge) the QR decomposition has not been formalised in CoqEAL (or in any other theorem prover), that, over similar algorithms, computation times were favourable to our approach (probably because of the use of immutable arrays instead of lists to implement vectors and matrices), and also that symbolic computation has not been considered in CoqEAL (again, to the best of our knowledge).

Also in Coq, Gonthier [71] implemented a version of Gaussian elimination (producing two different matrices, one describing the column space, the other one the row space, and a number, equal to the rank of the input matrix) that he

[3]The benchmarks have been carried out in laptop with an Intel Core i5-3360M processor, 4 GB of RAM, PolyML 5.5.2-3 and Ubuntu 14.04.

later applied for the computation of several basic operations over linear maps; for instance, the computation of the four fundamental subspaces of a given matrix (we formalised similar computations in Section 4.3 by means of the Gauss-Jordan algorithm, and thus performing, a priori, more elementary operations than he does in Gaussian elimination), and also basic operations between subspaces (union, intersection, inclusion, addition). One of its most prominent features is the simplicity that he obtains in proofs of properties concerning the rank and the fundamental subspaces. It seems that he has formalised neither inner product spaces nor orthogonality and the subsequent concepts (such as Gram-Schmidt, QR decomposition, and the least squares approximation). Since the focus of Gonthier's work seems to be in the formalisation of Linear Algebra, concerns about computability are neither tackled.

4.5.8 Conclusions

This formalisation can be considered as an advanced exercise in theorem proving, but at the same time it has required the use of well-established libraries (such as the HMA library) and the adaptation and setup of some previous works. Some of these works had been completed by us (such as the Rank-Nullity theorem, the Gauss-Jordan development, and the code refinements to *iarrays* and real numbers in Haskell) but some others tools were new to us (for instance, real numbers and the representation of field extensions). It is worth noting that our development relies on another 10 previous developments in the Isabelle AFP, two of them developed by us, and 8 of them from other authors. In that sense, with an affordable effort (the complete development sums up *ca.* 2700 lines, plus 2100 lines devoted to refinements, code generation, and examples), and also with a certain amount of previous knowledge of the underlying system, we have developed a set of results and formalised programs in Linear Algebra that corresponds to various lessons of an undergraduate text in mathematics (for instance, material which sums up 60 pages in the textbook by Strang [142, Sect. 3.6 and Chap. 4]). As a matter of comparison, the results presented in [142, Chap. 1 to Sect. 3.5], which include at least all the results that we formalised in our previous work about the Gauss-Jordan algorithm and its applications (Section 4.3) and additional previous notions of the HMA library and code generation (explained in Chapter 3), took us more than 14000 lines of Isabelle code (see Tables A.1 and A.2). From that point of view, we have to stress that this work would have been impossible without such previous developments; as it usually happens in mathematics, new results have to be built on top of established results. This principle is well-known in the formal methods community, but it is difficult to achieve; this work shows a successful case study where the principle has been materialised.

As further work, at least two possibilities show up. First, the results presented in Subsections 4.5.3 and 4.5.4 admit a generalisation from real inner product spaces to inner product spaces. The HMA library would also benefit from such a generalisation. Second, the application of the QR decomposition to numerical problems such as the computation of the eigenvalues, and also the formalisation of the related Singular Value Decomposition would be of wide interest.

Chapter 5

Algorithms involving Matrices over Rings

5.1 Introduction

The previous chapter shows the formalisation of two Linear Algebra algorithms involving matrices over fields: the Gauss-Jordan algorithm and the QR decomposition. Despite the fact that matrices over fields have been more studied throughout time, matrices over rings possess remarkable applications. Two well-known examples of this kind of matrices are integer matrices and matrices of polynomials over a field (also denoted as $\mathbb{Z}$ and $F[x]$-matrices respectively).

Integer matrices are widely used, for instance, in graph theory [27] and combinatorics [107]. Systems of diophantine equations can be represented using those matrices as well. Polynomial matrices are primarily used to compute the characteristic polynomial of a matrix A. The roots of the characteristic polynomial are the *eigenvalues*. Once the eigenvalues are known (λ), the *eigenvectors* can be obtained solving the homogeneous system of equations $(A - \lambda I) \cdot v = 0$ or by means of the Cayley-Hamilton theorem. The characteristic polynomial of a matrix has several applications, such as computing its inverse and performing powerful simplifications computing its powers. Eigenvalues and eigenvectors are useful both in mathematics (for instance, in differential equations [154] and factor analysis in statistics [37]) and in other varied fields, such as biometrics [65], quantum mechanics [114] and solid mechanics [25]. They are also used in the PageRank computation [102] (the algorithm used by Google Search to rank websites in their search engine results).

In this chapter we will show the formalisation of two algorithms involving matrices over more general rings than fields. Section 5.2 shows how we have formalised in Isabelle/HOL an algorithm to compute the echelon form (see Definition 8) of a matrix. This allows carrying out similar computations to the ones that we did by means of the Gauss-Jordan algorithm. Let us note that the Gauss-Jordan algorithm can only be applied to matrices over a field and the echelon form algorithm enables computations to be done over more general rings. It is worth remarking again that the Gauss-Jordan algorithm developed in Section 4.3 is applied to matrices whose elements belong to a field. Therefore, it is useful for doing computations, for instance, with matrices over $\mathbb{Z}_2$, $\mathbb{Q}$, $\mathbb{R}$

and $\mathbb{C}$, but it cannot be used for matrices over rings, such as integer matrices. In Section 5.3 the first ever formalisation (to the best of our knowledge) of the Hermite normal form of a matrix (see Definition 13) in an interactive theorem prover is presented, as well as a proof of its uniqueness.

5.2 Echelon Form

5.2.1 Introduction

A classical mathematical problem is the transformation of a matrix over a ring into canonical forms, which have many applications in computational Linear Algebra. These canonical forms contain fundamental information of the original matrix, such as the determinant and the rank. Examples of canonical forms are the Hermite normal form (usually applied to integer matrices), the Smith normal form and the reduced row echelon form. The most basic canonical form is the one called *echelon form*. Some authors, for instance Leon [104], use such a term to mean the output of the Gaussian elimination (which can only be applied to matrices over fields). However, the concept can be generalised to more general rings; other authors [140] have studied this generalisation and present the algorithm to *compute* the echelon form of a matrix over a principal ideal domain. Nevertheless, this canonical form can be *defined* over more general structures: its existence can be *proved* over matrices whose coefficients belong to a Bézout domain [89].

An algorithm to transform a matrix to its echelon form has many applications, such as the computation of determinants and inverses, since it is the analogous to the Gaussian elimination but involving more general rings. In addition, the echelon form can be used as a preparation step to compute the Hermite normal form of a matrix, and thus, to compute ranks and solutions of systems of linear diophantine equations.

Another advantage of having an algorithm to transform a matrix into its echelon form is that, as a by-product, the characteristic polynomial can be easily obtained (even though there exist other more efficient ways of computing characteristic polynomials).

In this section, we present a formalisation in Isabelle/HOL of an algorithm to obtain the echelon form of a given matrix. (The full development was published in the AFP [50].) We have formalised the algorithm over Bézout domains, but its executability is guaranteed only over Euclidean domains. This is possible since we have formalised the algorithm including an additional parameter: the operation that given two elements returns their Bézout coefficients. Let a and b be two elements of a Bézout domain; it is known that there exist p, q and g such that $pa + qb = g$ where g is the greatest common divisor of a and b. Bézout domains pose this operation, but neither its uniqueness nor its executability are guaranteed. The executability of this operation is at least guaranteed on Euclidean domains, based on the division algorithm. This way, we have been able to formalise the existence and correctness of an algorithm to obtain an echelon form over Bézout domains and get the computation over Euclidean domains. Even more, if one were able to provide an executable operation to compute Bézout coefficients in a concrete Bézout domain, the algorithm would become computable in that structure as well. That is, we can define a

computable version of the Bézout operation for Euclidean rings. Since there is no generally computable version of the Bézout operation for all Bézout domains, other Bézout domains need their own version of computable Bézout operations. These transformations into echelon forms allow computing inverses and determinants of matrices, as well as the characteristic polynomial. The wide range of applications of these concepts constitute a motivation to formalise such an algorithm.

The utility of this part of the thesis is threefold. First, we have enhanced the Isabelle ring library based on type classes including some structures, concepts and theorems that were missing: Bézout domains, principal ideal domains, GCD domains, subgroups, ideals, and more. Second, we have formalised an algorithm to transform a matrix into an echelon form, parametrised by a Bézout coefficients operation (that establishes if the algorithm will or will not be computable). As we have already said, this allows us to formalise the existence of the algorithm over Bézout domains and the computation over Euclidean domains. To improve the performance, a refinement to immutable arrays has also been carried out. Verified code to compute determinants and inverses (over matrices whose elements belong to a Euclidean domain, such as the integers and the univariate polynomials over a field) is generated, and also applied to compute characteristic polynomials of matrices over fields. Finally, we have successfully reused the infrastructure presented in Chapter 3, which was developed to formalise and refine Linear Algebra algorithms. This work shows its usefulness.

This section is divided as follows. Subsection 5.2.2 presents the algebraic structures we have implemented in Isabelle and the hierarchy of the involved classes. In Subsection 5.2.3 we explain both the formalisation of the algorithm and its relationship with the Gauss-Jordan development presented in Section 4.3. Subsection 5.2.4 presents the formalisation of some of the direct applications of the algorithm: computation of determinants, inverses and the characteristic polynomial of a matrix. Moreover, some other computations related to the Cayley-Hamilton theorem are shown as well as the verified refinement of the algorithm to immutable arrays in order to improve the performance. Subsection 5.2.5 shows some related work presented in the literature. Finally, conclusions and possible further work are presented in Subsection 5.2.6.

5.2.2 Algebraic Structures, Formalisation, and Hierarchy

Let us recall mathematical definitions of the main algebraic structures involved in this development as well as their formalisation that we have carried out in Isabelle.

Figure 5.1 shows the hierarchy of the main Isabelle type classes involved in the development. The arrows express strict inclusions (all of them have been proven in Isabelle); hence by the transitivity property of the inclusion one could figure out the dependencies and subclasses among the structures. As we will see later, there exist more classes involved in the formalisation, but Figure 5.1 shows the main ones. The structures that we have had to introduce are presented in bold.

The algebraic structures presented in this subsection and their properties have been formalised in the file *Rings2.thy* of [50]. Their mathematical definitions can be obtained from standard literature [36, 64, 134]. Let us start with

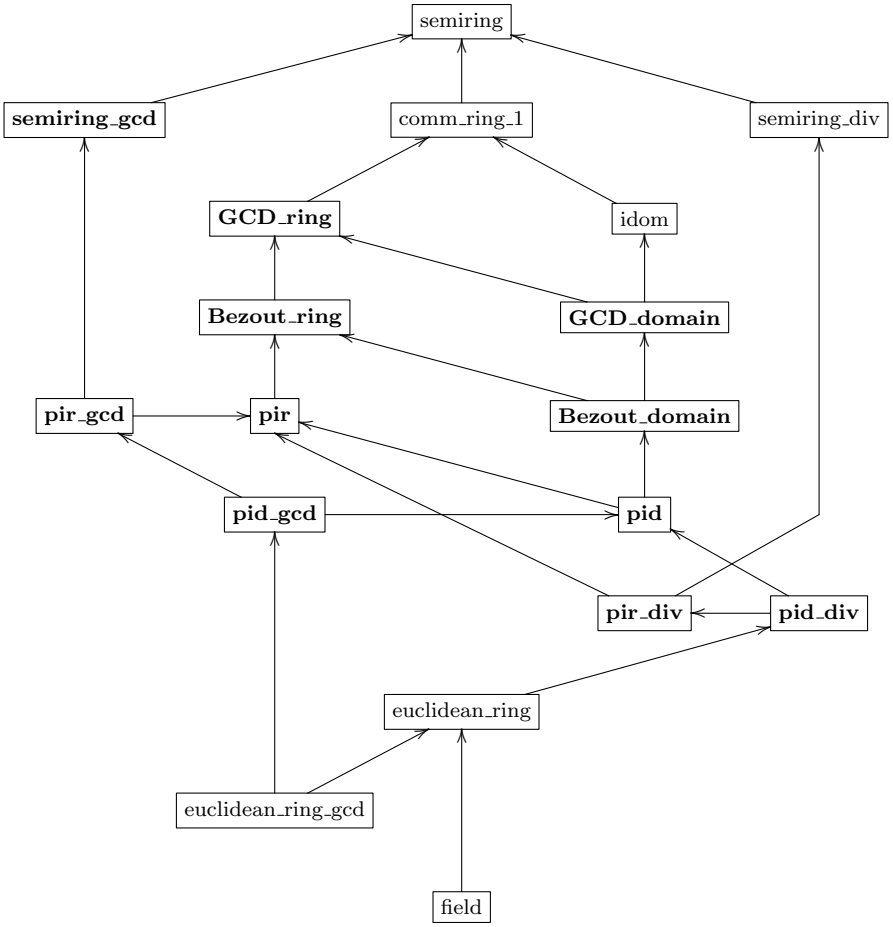


Figure 5.1: Hierarchy of the main classes appearing in the echelon form development (arrows express inclusions; transitive ones are left out). The structures that we have had to introduce are presented in bold.

the mathematical concept of *GCD ring* and *GCD domain*.

Definition 14. *A **GCD ring** R is a ring where every pair of elements has a greatest common divisor, that is, for $a, b \in R$ there exists $gcd(a, b) \in R$ such that:*

- $gcd(a, b) \mid a$
- $gcd(a, b) \mid b$
- $g \in R \wedge (g \mid a) \wedge (g \mid b) \Longrightarrow g \mid gcd(a, b)$

*If the ring is an integral domain (it has no zero divisors), the structure is called **GCD domain**.*

We have implemented them in Isabelle avoiding fixing a *gcd* operation. The existence of a *gcd* for each pair of elements a and b is just assumed:

```
class GCD_ring = comm_ring_1 +
assumes exists_gcd: "∃d. d dvd a ∧ d dvd b
  ∧ (∀d'. d' dvd a ∧ d' dvd b ⟶ d' dvd d)"

class GCD_domain = GCD_ring + idom
```

Bézout rings are a structure closely related to the previous one:

Definition 15. *A **Bézout ring** R is a ring where Bézout's identity holds, that is, for $a, b \in R$ there exist $p, q, d \in R$ such that:*

- $pa + qb = d$
- *d is a greatest common divisor of a and b*

*If the ring is an integral domain, the structure is called **Bézout domain**.*

Thus, the implementation in Isabelle that we have chosen for Bézout rings is quite similar to the one presented for GCD rings:

```
class bezout_ring = comm_ring_1 + assumes exists_bezout:
"∃p q d. (p*a + q*b = d) ∧ (d dvd a) ∧ (d dvd b)
 ∧ (∀d'. (d' dvd a ∧ d' dvd b) ⟶ d' dvd d)"

class bezout_domain = bezout_ring + idom
```

It is simple to prove that any Bézout ring is a GCD ring:

```
subclass GCD_ring
proof
  fix a b
  show "∃d. d dvd a ∧ d dvd b
    ∧ (∀d'. d' dvd a ∧ d' dvd b ⟶ d' dvd d)"
    using exists_bezout[of a b] by auto
qed
```

However, it is worth noting that the structures just assume the existence of the *gcd* and Bézout coefficients for any two elements, but say nothing about how

to compute them (the structures could be non-constructive, in the sense that no witnesses could be obtained). Moreover, both the *gcd* and Bézout coefficients could be non-unique.

Before introducing the concept of principal ideal ring, ideals must be presented:

Definition 16. *Let R be a ring. A nonempty subset I of R is called an* ***ideal*** *if:*

- *I is a subgroup of the abelian group R, that is, I is closed under subtraction;*
$$a, b \in I \Longrightarrow a - b \in I$$
- *I is closed under multiplication by any ring element, that is,*
$$a \in I, r \in R \Longrightarrow ra \in I$$

The ***ideal generated by*** *a set $S \subseteq R$ is the smallest ideal containing S, that is,*
$$\langle S \rangle = \bigcap \{I \mid ideal\ I \wedge S \subseteq I\}$$
A ***principal ideal*** *is an ideal that can be generated by an element $a \in R$, that is,*
$$I = \langle a \rangle = \{ra \mid r \in R\}$$

The implementation in Isabelle is done in a straightforward manner:

```
definition "ideal I = (subgroup I ∧ (∀a∈I. ∀r. r*a ∈ I))"
definition "ideal_generated S = ⋂{I. ideal I ∧ S ⊆ I}"
definition "principal_ideal S = (∃a. ideal_generated {a} = S)"
```

Definition 17. *A* ***principal ideal ring*** *(denoted as PIR) R is a ring where every ideal is a principal ideal. If the ring is also an integral domain, the structure is said to be a* ***principal ideal domain*** *(denoted as PID).*

Once the concepts of ideal and principal ideal have been defined in Isabelle, principal ideal rings are implemented in a direct way:

```
class pir = comm_ring_1
 + assumes all_ideal_is_principal: "ideal I ⟹ principal_ideal I"

class pid = idom + pir
```

In addition, some important lemmas have been demonstrated over this structure, for instance the *ascending chain condition*, which is fundamental for proving that any PID is a unique factorization domain (i.e., every element can be written in a unique way as a product of prime elements).

Theorem 13 (The ascending chain condition)**.** *Any principal ideal domain D satisfies the ascending chain condition, that is, D cannot have a strictly increasing sequence of ideals*
$$I_1 \subset I_2 \subset \ldots$$
where each ideal is properly contained in the next one.

Proof. Suppose to the contrary that there is such an increasing sequence of ideals. Consider the ideal

$$U = \bigcup_{i \in \mathbb{N}} I_i$$

which must have the form $U = \langle a \rangle$ for some $a \in U$. Since $a \in I_k$ for some k, we have $I_k = I_j$ for all $j \geq k$, contradicting the fact that the inclusions are proper. □

Our corresponding proof in Isabelle requires 30 lines. We have formalised an equivalent statement.

```
context pir
begin

lemma ascending_chain_condition:
  fixes I::"nat⇒'a set"
  assumes all_ideal: "∀n. ideal (I(n))" and inc: "∀n. I(n) ⊆ I(n+1)"
  shows "∃n. I(n)=I(n+1)"
```

Thanks to this result, it is straightforward to present the ascending chain condition in Isabelle by means of a statement closer to the mathematical one.

```
lemma ascending_chain_condition2:
  "∄I::nat⇒'a set. ∀n. ideal (I n) ∧ I n ⊂ I (n + 1)"
```

Let us show that any PIR is a Bézout ring. This proof is not immediate, but we have just needed about 90 lines of code in Isabelle. The demonstration is done as follows: given two elements a and b of a PIR, since every ideal is principal we can obtain an element d such that the ideal generated by d is equal to the ideal generated by the set $\{a, b\}$. Finally, it is shown that d is indeed a *gcd*, completing the proof.

```
subclass (in pir) bezout_ring
```

The mathematical definition of *Euclidean ring* is the following one:

Definition 18. *A **Euclidean ring** is a ring R with a Euclidean norm $f : R \to \mathbb{N}$ such that, for any $a \in R$ and nonzero $b \in R$:*

- *$f(a) \leq f(ab)$;*
- *There exist $q, r \in R$ such that $a = bq + r$ and either $r = 0$ or $f(r) < f(b)$.*

*If the ring is also an integral domain, the structure is said to be a **Euclidean domain**.*

We have reused the representation of Euclidean ring that was already in the Isabelle library.[1] It was developed by Eberl for a univariat form of Sturm's theorem [61]. Note that he uses ring to refer to an integral domain.

[1] Let us note that Isabelle libraries are continuously evolving. We present the current situation in the time this thesis is written, that is, before the Isabelle 2016 official release. Many modifications on the library are being carried out (specially by Eberl and Haftmann), and probably some results, instances and structures presented here will be part of the standard library after the Isabelle 2016 release.

```
class euclidean_semiring = semiring_div +
  fixes euclidean_size :: "'a ⇒ nat"
  fixes normalisation_factor :: "'a ⇒ 'a"
  assumes mod_size_less [simp]:
    "b ≠ 0 ⟹ euclidean_size (a mod b) < euclidean_size b"
  assumes size_mult_mono:
    "b ≠ 0 ⟹ euclidean_size (a * b) ≥ euclidean_size a"
  assumes normalisation_factor_is_unit [intro,simp]:
    "a ≠ 0 ⟹ is_unit (normalisation_factor a)"
  assumes normalisation_factor_mult: "normalisation_factor (a * b) =
    normalisation_factor a * normalisation_factor b"
  assumes normalisation_factor_unit:
    "is_unit x ⟹ normalisation_factor x = x"
  assumes normalisation_factor_0 [simp]: "normalisation_factor 0 = 0"

class euclidean_ring = euclidean_semiring + idom
```

Both the integers ($\mathbb{Z}$) and the univariate polynomials over a field ($F[x]$) form Euclidean domains. In the case of the integer numbers, a Euclidean norm is the absolute value. In the case of the polynomials, a Euclidean norm is $2^{deg(p(x))}$ (note that we assume $deg(0) = -\infty$). We have proved that $F[x]$ is an instance of the *euclidean_ring* class ($\mathbb{Z}$ was already proved to be an instance of it). In addition, we have proved that any Euclidean domain is a PID (about 50 lines) and that any field is a Euclidean domain.

```
instantiation poly :: (field) euclidean_ring
instantiation int :: euclidean_ring
```

In a Euclidean ring, a Euclidean algorithm can be defined to compute the gcd of any two elements. Furthermore, this algorithm can always be used to obtain Bézout coefficients. This constructive operation is presented in Isabelle in the *euclidean_ring* class with the name *gcd_euclid*.

Note that one additional operation was fixed by Eberl: the normalisation factor, which returns a unit such that dividing any element of the ring by that unit yields the same result for all elements in the same associationclass, effectively normalising the element. For instance, for integers, the normalisation factor is the sign of the number; for polynomials, it is the leading coefficient.

Eberl also defined two more classes: *euclidean_semiring_gcd* and *euclidean_ring_gcd*, where the operations gcd, lcm, Gcd (the gcd of the elements of a given set) and Lcm (analogous to the previous one) are fixed as part of the structure. We have proved that both $\mathbb{Z}$ and $F[x]$ are also instances of the *euclidean_ring_gcd* class. In addition, some theorems presented in the *euclidean_ring_gcd* class have been generalised to the *euclidean_ring* one.

For the sake of completeness, we have also implemented rings where for each two elements there exists a gcd in a constructive way, i.e. not only assuming the existence of a *gcd* operation but fixing it. The corresponding subclasses have also been proved:

```
class semiring_gcd = semiring + gcd +
  assumes "gcd a b dvd a"
  and "gcd a b dvd b"
```

```
  and "c dvd a ⟹ c dvd b ⟹ c dvd gcd a b"

class pir_gcd = pir + semiring_gcd
class pid_gcd = pid + pir_gcd

subclass (in euclidean_ring_gcd) pid_gcd proof
  fix a b c
  show "gcd a b dvd a" by (simp add: gcd_dvd1)
  show "gcd a b dvd b" by (simp add: gcd_dvd2)
  assume ca: "c dvd a" and cb: "c dvd b" show "c dvd gcd a b"
    by (simp add: ca cb dvd_gcd_iff)
qed

subclass (in euclidean_semiring_gcd) semiring_gcd
proof
  fix a b c
  show "gcd a b dvd a" by (simp add: gcd_dvd1)
  show "gcd a b dvd b" by (simp add: gcd_dvd2)
  assume ca: "c dvd a" and cb: "c dvd b" show "c dvd gcd a b"
    by (simp add: ca cb dvd_gcd_iff)
qed
```

Let us note that when proving that a given type is an instance of the *euclidean_ring_gcd* class, one has to prove, apart from the properties for being a Euclidean domain, that the type includes a *gcd* operation and provide a witness of it.

The *semiring_div* class is defined as a structure where there are two fixed operations: *div* and *mod*, so there is an explicit (constructive) divisibility (note that *semiring_div* is not a subclass of *semiring*, because *semiring* does not have such fixed operations). Hence we can distinguish between constructive structures (where the operations are fixed, for instance *semiring_div*, *semiring_gcd*, ...) and possibly non-constructive structures (where it is just assumed the existence of the operations, for instance *pir*, *bezout_domain*, ...).

For a full description of other algebraic structures related to the ones presented here (semirings, fields, unique factorization domains) and the relationships among them, see [64, 94, 134]. The following chain of strict inclusions is satisfied (as it has been explained in this section, all of them have been proven in Isabelle):

Field $\subset$ Euclidean ring $\subset$ Principal ideal ring $\subset$ Bézout ring $\subset$ GCD ring

5.2.3 Parametricity of Algorithms and Proofs

At the beginning of this section, we have said that our aim is to formalise an algorithm proving that there exists the echelon form of any matrix whose elements belong to a Bézout domain. In addition, we want to compute such an echelon form, so we will need computable *gcd* and *bezout* operations which exist, at least, over Euclidean domains. On the contrary, over more general algebraic structures the existence of *gcd* and Bézout coefficients is guaranteed, but maybe its computation is not. In order to specify *gcd* and *bezout* in such a way that they can be introduced in Bézout domains (*bezout_domain* class) and

linked to their already existing computable definitions in Euclidean domains (*euclidean_ring* class), we have considered several options:

1. We could define a gcd in Bézout rings and GCD rings as follows:

   definition *"gcd_bezout_ring a b = (SOME d. d dvd a ∧ d dvd b ∧ (∀d'. d' dvd a ∧ d' dvd b ⟶ d' dvd d)"*

   The operator *SOME* arises since the *gcd* could be non-computable and there could be more than one gcd for the same two elements. If an operation to compute Bézout coefficients is defined in a similar way, using it one could formalise the existence of an algorithm to obtain the echelon form over a Bézout domain.

   However, one would not be able to execute such an operation over Euclidean domains (neither over constructive Bézout domains) because it is not possible to demonstrate that *gcd_bezout_ring* is equal to *gcd_eucl* (the constructive operation over Euclidean domains to compute the gcd of two elements). Let us remark that the gcd is not unique over Bézout rings and GCD rings, and with the *gcd_bezout_ring* we would not know which of the possible greatest common divisors is returned by the operator *SOME*.

2. Based on the previous option, one could create a *bezout_ring_norm* class where the normalisation factor is fixed. Then, one could define a normalised gcd over such a **class**:

   definition *"gcd_bezout_ring_norm a b = gcd_bezout_ring a b div normalisation_factor (gcd_bezout_ring a b)"*

   Now one could demonstrate that: *gcd_bezout_ring_norm = gcd_eucl*. This would allow us to execute the *gcd* function, but with Bézout coefficients this is not possible since they are never unique.

3. The third option (and the chosen one) consists in defining the echelon form algorithm over Bézout domains and parameterising the algorithm by a *bezout* operation which must satisfy the predicate *is_bezout_ext*, which is presented below. From the caller's point of view, the operation can be considered an oracle. Then the correctness of the algorithm can be proven over Bézout domains since in such structures there always exists a possibly non-constructive operation which satisfies such a predicate. In addition, it can be executed over Euclidean domains, since we can demonstrate that there exists a computable *bezout* operation which satisfies the properties.

The properties that a *bezout* operation must satisfy are fixed by means of the predicate *is_bezout_ext*. It checks if the input is a function that given two parameters a and b returns 5 elements (p, q, u, v, d) where d is a gcd of a and b, p and q are the Bézout coefficients such that $pa + qb = d$, and u and v are such that $du = -b$ and $dv = a$. [2] We cannot define directly $u = -b/d$ and $v = a/d$ because in abstract structures, such as Bézout rings, we do not have

[2] The elements u and v appear since they will be used in the echelon form algorithm.

explicit divisibility (if we do so, we would have to work over a *bezout_ring_div* class instead of using the more general *bezout_ring* one).

```
context bezout_ring
begin

definition is_bezout_ext :: "('a⇒'a⇒('a×'a×'a×'a)) ⇒ bool"
  where "is_bezout_ext (bezout) =
    (∀a b. let (p, q, u, v, gcd_a_b) = bezout a b in
    p*a+q*b=gcd_a_b ∧ (gcd_a_b dvd a) ∧ (gcd_a_b dvd b)
    ∧ (∀d'. d' dvd a ∧ d' dvd b ⟶ d' dvd gcd_a_b)
    ∧ gcd_a_b * u = -b ∧ gcd_a_b * v=a)"

end
```

We have proven that there exists a (non-computable) function satisfying such a predicate over a *Bézout ring*:

```
context bezout_ring
begin

lemma exists_bezout_ext: "∃bezout_ext. is_bezout_ext bezout_ext"
```

Finally, we can define a computable operation (*euclid_ext2*) over Euclidean domains which satisfies the predicate *is_bezout_ext*.

```
context euclidean_ring
begin

definition "euclid_ext2 a b = (let e = euclid_ext a b;
  p = fst e; q = fst (snd e); d = snd(snd e)
  in (p,q,-b div d,a div d,d))"

lemma is_bezout_ext_euclid_ext2: "is_bezout_ext (euclid_ext2)"
```

Thanks to the lemma presented above, we know that there exists a constructive *bezout* operation over Euclidean domains. Therefore, if we define an algorithm based on it, it will be executable. Nevertheless, if one wants to work in more abstract structures than Euclidean domains, one must provide a computable operation if execution is pursued.

Finally, the approach to demonstrate the existence and correctness of the algorithm over Bézout domains and the execution over Euclidean domains is the following:

1. Define the algorithm over Bézout domains. The algorithm itself (operation *echelon_form_of*) will have a *bezout* operation as an additional parameter:

   ```
   definition "echelon_form_of A bezout = echelon_form_of_upt_k A
   (ncols A - 1) bezout"
   ```

2. Formalise the correctness of the algorithm over Bézout domains, under the premise that *bezout* satisfies the corresponding properties (the pred-

icate *is_bezout_ext*). We have shown previously that an operation satisfying such properties always exists over Bézout domains (see the lemma *exists_bezout_ext*). For example, the following lemma is the final result that says that the algorithm (*echelon_form_of*) indeed produces an echelon form (the predicate *echelon_form*) by means of elementary transformations (so there exists an invertible matrix which transforms the original matrix into its echelon form).

```
theorem echelon_form_of_invertible:
  fixes A::"'a::{bezout_domain}^'cols::{mod_type}^'rows::{mod_type}"
  assumes "is_bezout_ext bezout"
  shows "∃P. invertible P ∧ P**A = (echelon_form_of A bezout)
         ∧ echelon_form (echelon_form_of A bezout)"
```

3. Finally, as we know that the operation *euclid_ext2* has been defined over Euclidean domains, it is computable, and it satisfies the predicate *is_bezout_ext*, we will have a computable algorithm and the lemma stating that the algorithm produces a matrix in echelon form contains no premises.

```
corollary echelon_form_of_euclidean_invertible:
  fixes A::"'a::{euclidean_ring}^'cols::{mod_type}^'rows::{mod_type}"
  shows"∃P. invertible P ∧ P**A = (echelon_form_of A euclid_ext2)
        ∧ echelon_form (echelon_form_of A euclid_ext2)"
```

5.2.3.1 An Algorithm Computing the Echelon Form of a Matrix, Parametrically

It is time to introduce the parametrised algorithm to compute the echelon form of a matrix that we have formalised. In broad terms, the algorithm will be implemented traversing the columns. The reduction of a column k works as follows. Given the column k and a triple $(A, i, bezout)$, where A is the matrix, i the position of the column k where the pivot should be placed and *bezout* is the operation that must satisfy the predicate *is_bezout_ext*, the output is another triple $(A', i', bezout)$, where A' is the matrix A with the elements of column k equal to zero below i, i' is the position where the next pivot should be placed in column $k+1$ and *bezout* is the same operation. The main steps are:

1. If the pivot (the element in the position (i, k)) and all elements below it are zero, then it is necessary to do nothing. Just $(A, i, bezout)$ is returned.

2. If not, if all elements below the pivot are zero but the pivot is not, then we just have to increase the pivot, i.e. $i' = i + 1$. Thus, $(A, i + 1, bezout)$ is returned.

3. If not, then we have to look for a nonzero element below i, move it to the position (i, k) (where the pivot must be placed) interchanging rows and reduce the elements below the pivot by means of Bézout coefficients. We call this matrix A'. Hence, $(A', i + 1, bezout)$ is returned.

4. Apply the previous steps to the next column.

Let us explain in detail the algorithm. First of all, we have to define a special kind of matrices: the ones that have Bézout coefficients in the suitable place to reduce two elements of a column of a matrix. We will call this the (elementary) Bézout matrix:

$$E_{Bezout} = \begin{pmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ & \ddots & & & & & & \\ 0 & \cdots & p & \cdots & q & 0 & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & & \\ 0 & \cdots & u & \cdots & v & 0 & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 1 & \cdots & 0 \\ & & & & & & \ddots & \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \end{pmatrix}$$

The coefficients not explicitly shown in this matrix are assumed to be 0, except for the presented ones in the diagonal, which are 1. If p and q are Bézout coefficients of $A_{a,j}$ and $A_{b,j}$ such that $pA_{a,j} + qA_{b,j} = d$; $du = -A_{b,j}$ and $dv = A_{a,j}$ (that is, (p, q, u, v, d) are the output of a function which satisfies the predicate `is_bezout_ext` applied to the elements $A_{a,j}$ and $A_{b,j}$) then this matrix satisfies that:

$$E_{Bezout} \cdot \begin{pmatrix} * & & \cdots & & * \\ & & & & \\ * & \cdots & A_{a,j} & \cdots & * \\ & & \vdots & & \\ * & \cdots & A_{b,j} & \cdots & * \\ * & & \cdots & & * \end{pmatrix} = \begin{pmatrix} * & & \cdots & & * \\ & & & & \\ * & \cdots & d & \cdots & * \\ & & \vdots & & \\ * & \cdots & 0 & \cdots & * \\ * & & & & * \end{pmatrix}$$

Bézout matrices have good properties, such as being invertible and having determinant always equal to 1. Their implementation in Isabelle is:

```
context bezout_ring
begin

definition bezout_matrix ::"'a^'cols^'rows ⇒ 'rows ⇒ 'rows ⇒ 'cols⇒
('a⇒'a⇒('a × 'a × 'a × 'a × 'a))⇒'a^'rows^'rows"
where "bezout_matrix A a b j bezout
  = (let (p, q, u, v, d) = bezout (A $ a $ j) (A $ b $ j) in
        (χ x y. if x = a ∧ y = a then p else
                if x = a ∧ y = b then q else
                if x = b ∧ y = a then u else
                if x = b ∧ y = b then v else
                 (if x=y then 1 else 0)))"
end
```

We can multiply the input matrix A by different elementary Bézout matrices in order to reduce all the elements below the pivot i in a column j. This is carried out by means of a recursive function:

```
primrec bezout_iterate :: "'a::{bezout_ring}^'cols^'rows::{mod_type} ⇒
nat ⇒ 'rows::{mod_type}⇒'cols ⇒ ('a⇒'a⇒('a × 'a × 'a × 'a × 'a))
⇒ 'a^'cols^'rows::{mod_type}"
where "bezout_iterate A 0 i j bezout = A"
  | "bezout_iterate A (Suc n) i j bezout =
    (if (Suc n) ≤ to_nat i then A else
    bezout_iterate (bezout_matrix A i (from_nat (Suc n)) j bezout ** A) n
    i j bezout)"
```

The following definition is the key operation that applies the previous operation in one column k of the matrix, that is, the algorithm chooses the pivot, puts it in the suitable place (the position (i, k)) and reduces the elements below it by means of the recursive *bezout_iterate* operation presented above.

```
definition "echelon_form_of_column_k A' k
  = (let A = fst A'; i = fst (snd A'); bezout = snd (snd A');
     from_nat_k = from_nat k; from_nat_i = from_nat i in
        if (∀m≥from_nat_i. A $ m $ from_nat_k = 0) ∨ (i = nrows A) then
            (A, i, bezout)
        else
          if (∀m>from_nat_i. A $ m $ from_nat_k = 0) then
              (A, i + 1, bezout)
          else
            let n = LEAST n. A $ n $ from_nat_k ≠ 0 ∧ from_nat_i ≤ n;
            interchange_A = interchange_rows A from_nat_i n
           in
            (bezout_iterate (interchange_A) (nrows A - 1) from_nat_i
            from_nat_k bezout, i + 1, bezout))"
```

The previous operation is the one which carries out the four main steps presented at the beginning of this section. Interestingly, we have reused some of the operations presented in Chapter 3. Folding the operation *echelon_form_of_column_k* over all columns of the matrix, we define the algorithm:

```
definition "echelon_form_of_upt_k A k bezout =
(fst (foldl echelon_form_of_column_k (A,0,bezout) [0..<Suc k]))"

definition "echelon_form_of A bezout =
echelon_form_of_upt_k A (ncols A - 1) bezout"
```

It is worth remarking that every operation used in the algorithm has *bezout* (an operation that must return appropriate elements (p, q, u, v, d)) as an additional parameter.

5.2.3.2 Formalising the Computation of the Echelon Form of a Matrix, Parametrically

Let us explain how the formalisation has been accomplished. The mathematical definition of echelon form was presented in Subsection 2.1.3. Now we have to implement it in Isabelle. To do that, we define the concept *echelon form up to the column k* by means of a predicate. Then the predicate *echelon_form* will

just be *echelon form up to the last column*:

```
definition echelon_form_upt_k ::
"'a::{bezout_ring}^'cols::{mod_type}^'rows::{finite,ord}⇒nat⇒bool"
  where "echelon_form_upt_k A k = ((∀i. is_zero_row_upt_k i k A ⟶
  ¬ (∃j. j>i ∧ ¬ is_zero_row_upt_k j k A)) ∧
    (∀i j. i<j ∧ ¬ (is_zero_row_upt_k i k A) ∧ ¬ (is_zero_row_upt_k j
    k A) ⟶ ((LEAST n. A $ i $ n ≠ 0) < (LEAST n. A $ j $ n ≠ 0))))"

definition "echelon_form A = echelon_form_upt_k A (ncols A)"
```

The sketch of the proof is the following:

1. Show the basic properties of the `bezout_matrix`: it is invertible and its determinant is equal to 1.

2. Show by induction that the recursive function `bezout_iterate` indeed reduces all the elements below the pivot.

3. Show that `echelon_form_of_column_k` works properly, which means that it reduces the column k and preserves the elements of the previous columns.

4. Apply induction: if a matrix A is in echelon form up to the column k and `echelon_form_of_column_k` is applied to A in the column $k+1$, then the output will be a matrix in echelon form up to the column $k+1$.

The formalisation is presented in the file *Echelon_Form.thy* of [50]. Just one remark: remember that our approach consists of including an additional parameter `bezout` that must satisfy `is_bezout_ext`. So each lemma must have such an assumption. For instance, the following lemma states that `bezout_matrix` has determinant equal to 1:

```
lemma det_bezout_matrix:
  fixes A::"'a::{bezout_domain}^'cols^'rows::{finite,wellorder}"
  assumes ib: "is_bezout_ext (bezout)"
  and a_less_b: "a < b" and aj: "A $ a $ j ≠ 0"
  shows "det (bezout_matrix A a b j bezout) = 1"
```

Thanks to the infrastructure developed in Chapter 3 and in the Gauss-Jordan development (Section 4.3), we have been able to reuse many definitions and properties, saving effort. Even so, the proof of the correctness of the algorithm has taken about 3000 lines. The final theorem is the following one:

```
theorem echelon_form_of_invertible:
  fixes A::"'a::{bezout_domain}^'cols::{mod_type}^'rows::{mod_type}"
  assumes "is_bezout_ext bezout"
  shows "∃P. invertible P ∧ P ** A = echelon_form_of A bezout
         ∧ echelon_form (echelon_form_of A bezout)"
```

5.2.3.3 Computing the Formalised Version of the Echelon Form of a Matrix

Computation can be achieved parameterising `echelon_form_of` by an executable `bezout` operation. When working with Euclidean domains, we could use `euclid_ext2` for this purpose as we have explained at the beginning of this section.

The presented Isabelle formalisation of the echelon form algorithm can be directly executed inside of Isabelle (by rewriting specifications and code equations) with some setup modifications obtaining, thus, formalised computations. Such setup modifications are quite related to the ones that we presented in previous chapters. Essentially, we are carrying out the natural refinement: from the abstract and non-executable datatype `vec` to its executable implementation as functions over finite types. The code equations are established again by means of the `code_abstract` label.

Let us show an example of execution: the following command computes the echelon form of a 3×3 integer matrix (the computation is almost immediate):

```
value "let A=(list_of_list_to_matrix[[1,-2,4],[1,-1,1],[0,1,-2]]::int^3^3)
 in matrix_to_list_of_list (echelon_form_of A euclid_ext2)"
```

The output is: `[[1,-1,1],[0,1,-2],[0,0,1]]::int list list`

As in the Gauss-Jordan case, additional operations for conversion between lists of lists and functions (`list_of_list_to_matrix` and `matrix_to_list_of_list`) appear to avoid inputting and outputting matrices as functions, which can become rather cumbersome. Hence the input and output of the algorithm are presented to the user as lists of lists.

More examples of execution can be found in the file *Examples_Echelon_Form_Abstract.thy* of the development [50]. As expected, this way of executing the algorithm is rather slow, since the matrix representation based on functions over finite types is inefficient, but very suitable for formalisation purposes. For instance, the computation of the echelon form of a 8×8 integer matrix takes several minutes. To improve the performance, we have also refined the algorithm to immutable arrays and exported code to functional languages, reusing the infrastructure presented in Chapter 3 and following the same approach as in the Gauss-Jordan development (see Section 4.3) and the QR decomposition (see Section 4.5). This refinement is presented in Subsection 5.2.4.1.

5.2.3.4 Relation to the Reduced Row Echelon Form: Code Reuse and Differences

As we have said previously, the echelon form formalisation is highly based on other developments of ours: the framework presented in Chapter 3 and the formalisation of the Gauss-Jordan algorithm and its applications (see Section 4.3), since echelon form and reduced row echelon form (rref) are very related. The main difference is that the Gauss-Jordan algorithm (used to obtain the rref) works over matrices whose coefficients belong to a field, whereas the computation of the echelon form is carried out involving matrices over Bézout domains, a more abstract type of rings. Nevertheless, the elementary operations were defined and their properties were proven over general rings in the Gauss-Jordan

development [54], so we have been able to reuse them to implement and prove the correctness of the echelon form algorithm.

It is clear that rref implies echelon form and we have proven this fact in Isabelle. Thus, each lemma proved for echelon forms is also valid for rref. Furthermore, in the Gauss-Jordan development there were many properties stated over matrices in rref that have now been generalised to matrices in echelon form. These properties were fundamental in the development; in fact, we have reused the proofs presented in the Gauss-Jordan formalisation because some demonstrations were exactly the same. For instance, we had proved a lemma stating that a matrix in rref is upper triangular. Since the proof was essentially based on properties of echelon forms (the conditions 2 and 3 of Definition 9 were not necessary in the proof of the statement), changing rref by echelon form we got the theorem generalised.

The proof scheme in both developments is quite similar, except for the idea of parameterising the algorithm with the `bezout` operation. We followed the same strategy to define the algorithm and induction is applied over the columns.

5.2.4 Applications of the Echelon Form

There are three important applications of the echelon form that we have formalised:

1. Computation of determinants.
2. Computation of inverses.
3. Computation of characteristic polynomials.

All of them are closely related: inverses and characteristic polynomials are based on the computation of determinants. To compute the determinant of a matrix first we have to apply the algorithm to transform it to an echelon form. Since the echelon form is upper triangular and the transformation has been based on elementary operations, we just have to multiply the elements of the diagonal and maybe change the sign of the result (depending on the elementary operations performed in the transformation) to compute the determinant.

A notion of invariant appears in its formalisation. Given a matrix A, after n elementary operations the pair (b_n, A_n) is obtained, and it holds that $b_n \cdot (\det A) = \det A_n$. Since the algorithm terminates, after a finite number, m, of operations, we obtain a pair (b_m, `echelon_form_of A`) such that $b_m \cdot (\det A) = \det($`echelon_form_of` $A)$. The function `echelon_form_of_det` is the one which returns that pair of elements (b_m, `echelon_form_of A`). Since we are working in structures more general than a field, we have to prove that b_m is a unit of the ring (is invertible), in order to be able to isolate the determinant of A. In fact, we have proven that b_m will be a unit (for instance, in the case of integer matrices it can only be either 1 or -1). Finally, we proved that the determinant of an echelon form is the product of its diagonal elements, thus the computation is completed. From this, we have the final lemma:

```
corollary det_echelon_form_of_det_setprod:
  fixes A::"'a::{bezout_domain_div}^'n::{mod_type}^'n::{mod_type}"
  assumes ib: "is_bezout_ext bezout"
  shows "det A = ring_inv (fst (echelon_form_of_det A bezout))
```

```
  * setprod (λi. snd (echelon_form_of_det A bezout)$i$i) (UNIV::'n set)"
```

The inverse can be computed thanks to the fact that the following formula has been formalised in Isabelle: $A^{-1} = \frac{adjugate\ A}{\det A}$. The *adjugate* matrices were defined in the Cayley-Hamilton development in the AFP (see [8]), we have made that definition executable. The determinant will tell us if a matrix is invertible (a matrix is invertible iff its determinant is a unit).[3] So we will take care of the invertibility of the input matrix computing the determinant and making use of the Isabelle *option* type (whose elements are of the form `(Some x)` and `None`). The final statement for computing inverses over Euclidean domains is the one presented below:

```
lemma inverse_matrix_code_rings[code_unfold]:
  fixes A::"'a::{euclidean_ring}^'n::{mod_type}^'n::{mod_type}"
  shows "inverse_matrix A = (let d=det A in if is_unit d
    then Some (ring_inv d *ss adjugate A) else None)"
```

It is worth noting that both determinants and inverses can already be computed over fields, such as $\mathbb{C}$ and $\mathbb{Z}_2$, using the Gauss-Jordan algorithm. Thanks to this formalisation for computing echelon forms, the computation can be extended to Euclidean domains, such as $\mathbb{Z}$ and $F[x]$, and even to Bézout domains providing a *bezout* executable operation.

The characteristic polynomial of a matrix A is $\det(tI - A)$, so once determinants can be computed over a Euclidean domain thanks to the echelon form, characteristic polynomials come for free: it just consists of computing the determinant of a polynomial matrix. We had to prove that univariate polynomials over a field are a Euclidean domain and make executable some definitions presented in the Cayley-Hamilton development [8], where the characteristic polynomial was defined. The execution of all of these applications is carried out in a similar way to the ones of the Gauss-Jordan algorithm and the echelon form itself:

```
value "let A=(list_of_list_to_matrix [[3,2,8],[0,3,9],[8,7,9]]::int^3^3)
  in det A"
value "let A=list_of_list_to_matrix([[3,5,1],[2,1,3],[1,2,1]])::real^3^3
  in (charpoly A)"
value "let A=list_of_list_to_matrix([[3,5,1],[2,1,3],[1,2,1]])::int^3^3
  in (inverse_matrix A)"
```

The corresponding outputs are the following ones:

```
-156::int
[:7,-10,-5,1:]::real poly
None
```

[3] In fields all nonzero elements are units, but in more abstract rings there can be nonzero elements which are not units.

Note that the last output is `None`, since its corresponding input matrix was not invertible (it is an integer matrix whose determinant is -7, which is not a unit in $\mathbb{Z}$). `[:7,-10,-5,1:]::real poly` represents the polynomial $x^3 - 5x^2 - 10x + 7$.

Finally, another contribution of our work is that we have made executable most of the definitions presented in the Cayley-Hamilton development [8], such as minors, adjugates, cofactor matrix, the evaluation of polynomials of matrices and more, which have important applications in Linear Algebra. This part of the work is presented in the file *Code_Cayley_Hamilton.thy* of our development [50].

5.2.4.1 Code Refinement

As we have said in Subsection 5.2.3.3, the formalised algorithm is computable but the performance is not as good as it is desirable. Since the Isabelle code is not suitable for computing purposes, the original Isabelle specifications are refined to immutable arrays and translated to SML and Haskell, intensively reusing the infrastructure presented in Chapter 3.

The previous algorithm `echelon_form_of` has to be redefined over immutable arrays. After that, we have had to demonstrate the equivalence between the formalised algorithm over matrices represented as functions over finite types, and matrices represented as immutable arrays. The following lemma states that the echelon form computed over functions is the same as the one computed over immutable arrays (let us recall that `matrix_to_iarray` represents a *type morphism*):

```
lemma matrix_to_iarray_echelon_form_of[code_unfold]:
shows "matrix_to_iarray (echelon_form_of A bezout)
    = echelon_form_of_iarrays (matrix_to_iarray A) bezout"
```

Every operation presented in this section and every application (determinants, inverses, characteristic polynomial) has been refined to immutable arrays. Additionally, we make use again of serialisations. In our case, we have made use of the serialisations presented in Chapter 3 (such as *Vector.vector* and *IArray.array* to encode immutable arrays in SML and Haskell respectively; and the type for representing integer numbers in the target languages). As in the Gauss-Jordan development, we have included some more serialisations for the `gcd`, `div` and `mod` integer operations. Serialising the `gcd` Isabelle operation to the corresponding built-in Poly/ML [130] and MLton [113] functions (which are not part of the SML Standard Library, but particular to each compiler), increases notably the performance.

The generated SML code has about 2400 lines. We have said in Subsection 5.2.3.3 that the computation of the echelon form of a 8×8 integer matrix took several minutes using the matrix representation based on functions. Thanks to this refinement and the serialisations, when code is exported to SML the determinant of a random 20×20 integer matrix using immutable arrays needs 0.254 seconds to be computed in a basic laptop.[4] More examples of execution of our algorithm are shown in the file *Examples_Echelon_Form_IArrays.thy*.

[4] Intel® Core™ i5-3360M processor (2 cores, 4 threads) with 4GB of RAM.

5.2.5 Related Work

There are several formalisations of Linear Algebra in most proof systems, above all focusing the point on vector spaces properties. But only a few of them have explored algorithmic aspects. Probably the closest work to ours is the one presented in [34]. It is a formalisation of Linear Algebra over elementary divisor rings in Coq. In that paper it is presented a formalisation of the Smith normal form. The algorithm performs similar transformations to the ones we have presented in this part of the thesis. The main difference between their work and ours is that they are restricted to use constructive structures, such as *constructive principal ideal domains*. On the other hand, we can work with more abstract structures where we know the existence of divisions and greatest common divisors, but maybe not how to compute them. This allows us to formalise the algorithm involving (not necessarily constructive) Bézout domains. In addition, the computation of inverses, determinants and characteristic polynomials are not tackled in such a paper.

As other related work, the computation of the determinant of matrices over general rings has also been explored in a later formalisation in Isabelle/HOL about matrices and Jordan Normal Forms [148] by Thiemann and Yamada. The algorithm presented in such a work is specific to compute determinants and it is not based on elementary operations, so it cannot be applied to obtain canonical forms of matrices and thus to compute other objects such as ranks of matrices and solutions of systems of linear diophantine equations. In addition, they define and prove the algorithm just over computable structures, since a computable division operation is required.

Besides, the Sasaki-Murao algorithm has been formalised in Coq [41]. The Sasaki-Murao algorithm is specially designed to compute the determinant of square matrices over a commutative ring. In Section 4 of the last reference, the authors study the performance of such a formalised algorithm: computing the determinant of a random 20×20 integer matrix needs 62.83 seconds over the Coq virtual machine. Although the algorithm is specially designed for that computation, when they generate code to Haskell that determinant is computed in 0.273 seconds, a similar time to the one obtained by us using our echelon form algorithm (0.254 seconds, as it was presented in Subsection 5.2.4.1).

5.2.6 Conclusions and Future Work

In this work we have presented a formalisation of an algorithm to compute the echelon form of a matrix. The correctness of the algorithm has been proven over Bézout domains and its executability is guaranteed over constructive structures, such as Euclidean domains. In order to do that we have parametrised the functions of the algorithm by the operation *bezout*. This operation will be the key: if *bezout* is treated as an oracle, we can prove the correctness of the algorithm but we cannot compute it. That is, the algorithm is proved correct for any choice for bezout operation. By instancing *bezout* by a computable operation (according to suitable properties), the echelon form will be computable. As far as we know, it is the first time that the correctness of an algorithm is demonstrated over non-constructive algebraic structures and executed over constructive ones. Furthermore, the algorithm has been refined to immutable arrays in order to improve the performance. The applications of the algorithm

(determinants, inverses, characteristic polynomials) have also been formalised and refined, increasing the work that we did in the Gauss-Jordan development (Section 4.3) to more abstract rings. Such a Gauss-Jordan formalisation and the framework presented in Chapter 3 have intensively been reused: the infrastructure developed there (elementary operations, code generator setup, refinement statements, relationship between matrices and linear maps, matrix properties, ...) has shown to be very useful. One sign of it is that the whole development of the echelon form took *ca.* 8000 lines of Isabelle code, whereas the Gauss-Jordan formalisation plus the preliminary infrastructure developed needed *ca.* 14000 lines. This is remarkable because the echelon form algorithm is a more difficult algorithm than the Gauss-Jordan one (mainly because more abstract rings are involved and not each division is exact) and shows how much code has been reused and the helpfulness of the developed infrastructure in such a formalisation. As a by-product, some algebraic structures (Bézout rings, principal ideal domains, ...) and their properties (ideals, subgroups, relationships among them) have been formalised, enhancing the Isabelle library of rings using type classes.

As further work, it would be desirable to increase the developed library of rings with some other concepts, such as irreducible and prime elements, and with more algebraic structures, such as Prüfer domains and Noetherian rings. In addition, it would be interesting to provide more instances to Bézout domains, apart from the already existing ones $\mathbb{Z}$ and $F[x]$. As a natural continuation to our work, the formalisation of the Smith normal form would be very interesting. This is feasible thanks to both the infrastructure already developed and the ring theory presented in this contribution. In addition, the computation of eigenvalues and eigenvectors from the characteristic polynomial would be desirable.

Our algorithm to compute the echelon form (and hence, the characteristic polynomial) of a matrix relies on a function to compute Bézout coefficients and the *gcd* of a couple of elements, so performance strongly depends on the efficiency of such a function. Thus, the formalisation of efficient algorithms to compute *gcds*, both exact [31] and approximate [123], would be interesting.

5.3 Hermite Normal Form

The previous section has shown how we have carried out a formalisation to compute the echelon form of a matrix. Now, we present a continuation of such a work: the computation of the Hermite normal form. As it has already been pointed out in Chapter 2, the Hermite normal form is an analogue of the reduced row echelon form, but for matrices over more general rings than fields.

Let us remark again that the Hermite normal form is normally presented only in the case of integer matrices, but indeed this form exists for matrices whose elements belong to a Bézout domain. Accordingly, the algorithm is commonly stated for integer matrices; but, as we have formalised, it can be executed (at least) over matrices over any Euclidean domain.

It is worth remarking that the Hermite normal form is a well-known canonical matrix due to the fact that it plays an important role in many different applications. Concretely, it can be used for solving systems of linear diophantine equations [32], loop optimisation techniques [131], algorithmic problems in lattices [76], cryptography [149], and integer programming [66, 93] among other

applications.

In this section, we present a formalisation of an algorithm to compute the Hermite normal form of a matrix based on the echelon form algorithm presented in Section 5.2. In order to obtain better performance, the algorithm has also been refined to immutable arrays following the infrastructure explained in Chapter 3.

The main notions required for this section are introduced in Chapter 2. They are the complete set of nonassociates (Definition 10), the complete set of residues (Definition 11), the definition of Hermite normal form (Definition 12), and the definition of the Hermite normal form of a matrix (Definition 13).

As we have already said in Subsection 2.1.3, there is no one single definition of the Hermite normal form of a matrix in the literature, but all the different possibilities can be represented selecting a complete set of nonassociates and a complete set of residues. We are presenting the formalisation of an algorithm to compute the Hermite normal form. This algorithm will be parametrised by functions which obtain appropriate leading coefficients and the suitable elements above them (the residues). This way, one can instantiate the algorithm with different functions to get exactly *the* Hermite normal form one wants, not only involving integer matrices but matrices whose elements belong to any Bézout domain. Once such a complete set of nonassociates and the corresponding complete set of residues are fixed, the Hermite normal form is unique (this result will also be formalised).

5.3.1 Formalising the Hermite Normal Form

The whole development was published in the AFP [57], and directly relies on our echelon form algorithm [50] (in order to transform any matrix into its echelon form, as a preparatory step of the Hermite normal form) and therefore, in the Gauss-Jordan development [54] (to reuse the code generation setup presented in Chapter 3 and Section 4.3) and the formalisation of the Rank-Nullity theorem [52] as well (to reuse the link between linear maps and matrices). Thanks to these dependences it was completed in an affordable amount of lines: it took us *ca.* 2300 Isabelle code lines, including the proof of its uniqueness, the refinement to immutable arrays, and some examples of execution. As a comparison, the formalisation of the the Gauss-Jordan algorithm explained in Section 4.3 took us 10000 lines.

Before starting with the definition and the proof of the Hermite algorithm, we had to enhance the HMA library including some preliminary results, specially about triangular matrices. Such results will be crucial when demonstrating the uniqueness of the Hermite normal form. More concretely, we have proven the following properties for matrices over a ring (the properties also hold for lower triangular matrices, we just show here the case of the upper triangular ones):

1. If a matrix is upper triangular and its determinant is a unit, then each element of the diagonal is a unit.

2. The product of two upper triangular matrices is an upper triangular matrix.

3. If a matrix is upper triangular, then its adjugate is too.

4. If a matrix is invertible and upper triangular, its inverse is upper triangular.
5. If A and B are upper triangular, then $\forall i.(AB)_{i,i} = A_{i,i}B_{i,i}$.

Their corresponding Isabelle's statements look as follows (we are using again the HMA library representation for matrices as functions over finite domains, that is, based on the data type `vec`). They can be reused in other non-related developments.

```
lemma is_unit_diagonal:
  assumes U: "upper_triangular U"
  and det_U: "is_unit (det U)"
  shows "∀ i. is_unit (U $ i $ i)"

lemma upper_triangular_mult:
  assumes A: "upper_triangular A"
  and B: "upper_triangular B"
  shows "upper_triangular (A**B)"

lemma upper_triangular_adjugate:
  assumes A: "upper_triangular A"
  shows "upper_triangular (adjugate A)"


lemma upper_triangular_inverse:
  assumes A: "upper_triangular A"
  and inv_A: "invertible A"
  shows "upper_triangular (matrix_inv A)"

lemma upper_triangular_mult_diagonal:
  assumes A: "upper_triangular A"
  and B: "upper_triangular B"
  shows "(A**B) $ i $ i = A $ i $ i * B $ i $ i"
```

Let us introduce the Isabelle definition of associated elements and congruent elements. The definition of associated elements is already present in the Isabelle library thanks again to Eberl:

```
definition associated :: "'a ⇒ 'a ⇒ bool"
where
  "associated x y ⟷ x dvd y ∧ y dvd x"
```

We have defined that two elements $a, b \in R$ are congruent modulo c in the following way:

```
context ring_1
begin

definition cong::"'a⇒'a⇒'a⇒bool"
  where "cong a b c = (∃k. (a - b) = c * k)"

lemma cong_eq: "cong a b c = (c dvd (a - b))"
```

The latter lemma is just an equivalent definition, showing that a and b are congruent modulo c if c divides $(a - b)$.

Now, we have to define the corresponding relationships of associates and congruence introduced by the definitions. In Isabelle, this is carried out by means of a set of pairs: two elements (a, b) will belong to the set if they are related. Hence:

```
definition "associated_rel = {(a,b). associated a b}"
definition "congruent_rel c = {(a,b). cong a b c}"
```

Once we have such definitions, we prove both of them to be reflexive, transitive, and symmetric. This shows that they form equivalence relationships, as it is stated in the first part of Definitions 10 and 11.

```
lemma equiv_associated:
  shows "equiv UNIV associated_rel"

lemma equiv_congruent:
  shows "equiv UNIV (congruent_rel c)"
```

Let us tackle the second part of Definitions 10 and 11, that is, introducing predicates describing the complete set of nonassociates and the complete set of residues. They are very important, since they fix how the Hermite algorithm works and its possible variations. Our approach consists of defining two predicates, stating if a function is an *associates function* and a *residues function*, respectively. The *associates function* will define Hermite algorithm's behaviour in the leading entries, which must belong to a complete set of nonassociates. Then, the *residues function* will define Hermite algorithm's behaviour in the elements above the leading entries. Those elements must belong to a complete set of residues. Then, we will parametrise the Hermite algorithm by two functions (among other things): an *associates function* and a *residues function*. This way, we can obtain the different versions of the Hermite algorithm. For example, in the case of integer matrices if we fix the *associates function* as the absolute value, we will obtain the Hermite version where the leading entries are positive integers. On the contrary, we could also parametrise the algorithm by other *associates function*: minus the absolute value, in order to get a different Hermite version where the leading entries are negative integers. The case of residues functions is similar: using different ones we can obtain, for example, the Hermite versions where the elements above the leading entry are nonnegative integers or nonpositive integers.

Let us show what we have called *associates function*. A function f is an *associates function* if for all $a \in R$, then a and $f(a)$ are associated. In order to obtain a complete set of nonassociates, we also impose the elements belonging to the range of f to be pairwise nonassociates (that is, if $f(a) \neq f(b)$, then $f(a)$ and $f(b)$ are not associated) The Isabelle definition is the following one:

```
definition ass_function :: "('a ⇒ 'a) ⇒ bool"
  where "ass_function f = ((∀a. associated a (f a))
    ∧ pairwise (λa b. ¬ associated a b) (range f))"
```

Thus, a set S will be a complete set of nonassociates if there exists an

associates function f whose range is S:

```
definition "Complete_set_non_associates S
  = (∃f. ass_function f ∧ f`UNIV = S)"
```

The following desired properties can be proven:

1. Any associates function forms a complete set of nonassociates (by means of its range).

2. For any two elements which belong to a complete set of nonassociates, if they are different then they are not associated.

Their corresponding proofs in Isabelle are almost straightforward from the definitions:

```
lemma ass_function_Complete_set_non_associates:
  assumes f: "ass_function f"
  shows "Complete_set_non_associates (f`UNIV)"

lemma in_Ass_not_associated:
  assumes Ass_S: "Complete_set_non_associates S"
  and x: "x∈S" and y: "y∈S" and x_not_y: "x≠y"
  shows "¬ associated x y"
```

Let us present now what we have called *residues function*. A function f is a *residues function* if, fixed $c \in R$, the following conditions are satisfied.

1. For all $a, b \in R$ then a and b are congruent modulo c if and only if $f\ c\ a = f\ c\ b$.

2. The elements which belong to the range of f are pairwise noncongruent modulo c.

3. For all $a \in R$, there exists $k \in R$ such that $f\ c\ a = a + kc$.

The latter condition just means that we can transform elements above the leading entry by means of elementary operations (in the algorithm, c would represent the leading entry and a would be the element above the leading entry we want to transform to $f\ c\ a$).

In order to clarify this, let us introduce a toy example. Let A be the following integer matrix:

$$A = \begin{pmatrix} 7 & -3 \\ 0 & 5 \end{pmatrix}$$

A is a matrix in echelon form. The leading entry of the second row is $A_{2,2} = 5$. There is just one element above it, $A_{1,2} = -3$. Let the modulo operation be the residues function (which indeed is a residues function and it satisfies the properties presented above, as we will see later). With the previous notation, we have $a = -3$, $c = 5$. Thus $f\,c\,a = 2$ and $k = 1$. The matrix can be transformed into the following one by means of elementary operations (adding to the first row the second one):

$$A' = \begin{pmatrix} 7 & 2 \\ 0 & 5 \end{pmatrix}$$

In addition, let us note that the absolute value is not a residues function (we cannot transform $A_{1,2}$ into 3 by means of elementary operations, since there exists no $k \in \mathbb{Z}$ such that $3 = (-3) + 5k$).

We have defined the predicate to know if a function is a *residues function* in Isabelle as follows:

```
definition res_function :: "('a ⇒ 'a ⇒ 'a) ⇒ bool"
  where "res_function f = (∀c. (∀a b. cong a b c ⟷ f c a = f c b)
  ∧ pairwise (λa b. ¬ cong a b c) (range (f c))
  ∧ (∀a. ∃k. f c a = a + k*c))"
```

Now we have to define the concept of complete set of residues. In fact, we have one complete set of residues for each element $c \in R$. So, we will say that g is a complete set of residues[5] if there exists a residues function f such that each set $g\ c$ is exactly the range of $f\ c$. We have modelled it in Isabelle by means of the following definition:

```
definition "Complete_set_residues g
  = (∃f. res_function f ∧ (∀c. g c = f c‘UNIV))"
```

Again, we can prove the desired properties:

1. Any residues function induces a complete set of residues.
2. For any two elements $x, y \in g\ b$ if $x \neq y$ then they are not congruent with respect to b.

Fortunately, such proofs are quite direct from the definitions (we avoid presenting them now, we just show the statements).

```
lemma res_function_Complete_set_residues:
  assumes f: "res_function f"
  shows "Complete_set_residues (λc. (f c)‘UNIV)"

lemma in_Res_not_congruent:
  assumes res_g: "Complete_set_residues g"
  and x: "x ∈ g b" and y: "y ∈ g b" and x_not_y: "x≠y"
  shows "¬ cong x y b"
```

This completes Definitions 10 and 11. As we have already said, the Hermite algorithm will be parametrised by an associates function and a residues function to obtain its existing different versions. The Hermite algorithm can only be parametrised with certain functions, the proof of its correctness will assume that such functions are really associates and residues functions respectively. Therefore, we have to provide instances for them. Let us remark that we are not working exclusively in the case of integer matrices, but involving matrices over more general rings. As an example, we can provide (executable) associates and

[5] Strictly speaking, g is not a set but a function than given an element $c \in R$ returns a set.

residues function involving elements over Euclidean domains, which include for instance, the integers and the univariate polynomials over a field. The general definitions involving Euclidean domains are presented below. Concretely, the associates function will return the absolute value of the element in the case of the integers ring and the corresponding monic polynomial in the case of the ring of univariate polynomials over a field.

```
definition "ass_function_euclidean (p::'a::{euclidean_ring}) = p div
normalisation_factor p"
definition "res_function_euclidean b (n::'a::{euclidean_ring}) = (if b =
0 then n else (n mod b))"
```

In order to be able to use such functions in the algorithm, we have to prove them to be associates and residues functions respectively:

```
lemma ass_function_euclidean: "ass_function ass_function_euclidean"

lemma res_function_euclidean: "res_function (res_function_euclidean)"
```

We could provide other different instances of associates and residues function, or even define them explicitly to concrete structures. For instance, the associates function for integer elements could have been defined by:

```
definition "ass_function_int (n::int) = abs n"
```

We can prove this definition to be an associates function and it indeed corresponds to the nonnegative elements of $\mathbb{Z}$.

```
lemma ass_function_int: "ass_function ass_function_int"
lemma ass_function_int_UNIV: "(ass_function_int` UNIV) = {x. x≥0}"
```

Let us present here how we have implemented in Isabelle the predicate in Isabelle that characterises the Hermite normal form (Definition 12):

```
definition "Hermite associates residues A = (
  Complete_set_non_associates associates
  ∧ (Complete_set_residues residues)
  ∧ echelon_form A
  ∧ (∀i. ¬ is_zero_row i A ⟶ A $ i $ (LEAST n. A $ i $ n ≠ 0) ∈
associates)
  ∧ (∀i. ¬ is_zero_row i A ⟶ (∀j. j<i ⟶ A $ j $ (LEAST n. A $ i $
     n ≠ 0) ∈ residues (A $ i $ (LEAST n. A $ i $ n ≠ 0))))
  )"
```

Essentially, the definition is parametrised by a matrix A and two functions, *associates* and *residues*, which are demanded to be associates and residues functions respectively. After that, A is required to satisfy the three conditions of Definition 12, that is, A is in echelon form, any leading entry belongs to the complete set of nonassociates and any element above a leading entry belongs to the complete set of residues with respect to its corresponding leading entry.

We have implemented the Hermite algorithm in Isabelle following a similar idea to the one we used in the Gauss-Jordan algorithm, but in this traversing

over rows. That is, we have defined an operation that carries out the transformations over one row and then we have defined the Hermite algorithm folding such an operation over all rows (note that in the Gauss-Jordan algorithm we traversed the operation over the columns).

Our Hermite algorithm relies on the echelon form algorithm and again we follow the idea of parametrising it. In this case, the algorithm is parametrised by three functions:

- The function that computes Bézout's identity of two elements (necessary to compute the echelon form).
- The function that given an element, returns its representative element [6] in the associated equivalent class, which will be an element in the complete set of nonassociates.
- The function that given two elements a and b, returns its representative element in the congruent equivalent class of b, which will be an element in the complete set of residues of b.

Given a matrix A and a Bézout operation, the associates, and residues functions, our Hermite algorithm works as follows:

1. A is transformed to its echelon form by means of the formalised algorithm which was presented in Section 5.2.
2. Each nonzero row of the matrix is transformed to have its leading entry in the complete set of nonassociates by means of the associates function.
3. For each nonzero row, the elements above its leading entry are transformed to belong to the complete set of residues of the leading entry by means of the residues function.

The following functions do the job. `Hermite_reduce_above` reduces an element above the leading entry (it has as a parameter the function which computes an appropriate residue). In the previous examples (Gauss-Jordan, QR decomposition, echelon form) we have implemented the traversing operations over columns by means of a `foldl` operator over the columns' indexes; we use here a primitive recursive definition (by means of the **primrec** constructor) over the representation of the columns indexes as natural numbers. Both approaches are similar in practice.

```
primrec Hermite_reduce_above ::
    "'a::ring_div^'cols::mod_type^'rows::mod_type⇒nat
    ⇒'rows⇒'cols⇒ ('a⇒'a⇒'a) ⇒ 'a^'cols::mod_type^'rows::mod_type"
where "Hermite_reduce_above A 0 i j res  = A"
    | "Hermite_reduce_above A (Suc n) i j res  =
(let i'=((from_nat n)::'rows);
    Aij = A $ i $ j;
    Ai'j = A $ i' $ j
    in
    Hermite_reduce_above (row_add A  i' i (((res Aij (Ai'j)) - (Ai'j))
```

[6] By representative element, we mean an element of the class that will represent all the elements related to it

```
    div Aij)) n i j res)"
```

This function will also be reused in the `Hermite_of_row_i` function, which transforms the leading entry of a nonzero row to belong to the corresponding complete set of nonassociates as well (and it also has both the associates and the residues functions as parameters).

```
definition "Hermite_of_row_i ass res A i = (
  if is_zero_row i A
     then A
  else
    let j = (LEAST n. A $ i $ n ≠ 0); Aij= (A $ i $ j);
    A' = mult_row A i ((ass Aij) div Aij)
    in Hermite_reduce_above A' (to_nat i) i j res)"
```

`Hermite_of_upt_row_i` iterates the process up to a row i.

```
definition "Hermite_of_upt_row_i A i ass res = foldl (Hermite_of_row_i
ass res) A (map from_nat [0..<i])"
```

Finally, `Hermite` will apply `Hermite_of_upt_row_i` up to the last row.

```
definition "Hermite_of A ass res bezout =
  (let A'= echelon_form_of A bezout in Hermite_of_upt_row_i A' (nrows A)
ass res)"
```

Once the Hermite algorithm has been defined, we have to prove its correctness. Concretely, there are four facts to prove (see Definition 13):

1. The output matrix is in echelon form.
2. Each leading entry belongs to the complete set of nonassociates.
3. Each element above a leading entry (in the same column) belongs to the corresponding complete set of residues.
4. The algorithm is carried out by means of elementary row operations (that is, the resulting matrix is actually the Hermite normal form of the input matrix).

The first one means that the operations we carried out to transform the leading coefficients and the elements above them preserve the echelon form. After some effort (*ca.* 1400 Isabelle code lines), we can obtain the final theorem which corresponds to Definition 13.

```
theorem Hermite:
  assumes a: "ass_function ass"
  and r: "res_function res"
  and b: "is_bezout_ext bezout"
  shows "∃P. invertible P ∧ (Hermite_of A ass res bezout) = P ** A ∧
```

```
Hermite (range ass) (λc. range (res c)) (Hermite_of A ass res bezout)"
```

`(range ass)` and `(λc. range (res c))` represent the complete sets of associates and residues obtained from the associates and residues functions `ass` and `res` respectively. Let us remark again that the theorem has been proven involving matrices over Bézout domains, not only for the concrete case of integer matrices.

Furthermore, with no so much effort (*ca.* 150 Isabelle code lines), we can refine the algorithm to immutable arrays (and export code to SML and Haskell) to obtain a better performance thanks to the work presented in Chapter 3.

For the sake of completeness, let us provide two examples of execution of our formalised algorithm. The examples involve square matrices, but the algorithm can also be applied to non-square ones. Both of them use the *standard* associates and residues functions we have defined for Euclidean domains.

The first example is the computation of the Hermite normal form of the following integer matrix:

$$\begin{pmatrix} 37 & 8 & 6 \\ 5 & 4 & -8 \\ 3 & 24 & -7 \end{pmatrix}$$

The computation within Isabelle is done by means of the following command:

```
value[code] "let A = list_of_list_to_matrix
  ([[37,8,6],[5,4,-8],[3,24,-7]])::int^3^3
  in matrix_to_list_of_list (Hermite_of A ass_function_euclidean
    res_function_euclidean euclid_ext2)"
```

Let us note that the function is parametrised by four things:

- The input matrix A
- The *standard* associates function for Euclidean domains (`ass_function_euclidean`).
- The *standard* residues function for Euclidean domains (`res_function_euclidean`).
- The function which computes Bézout coefficients in Euclidean domains, which is necessary for the echelon form algorithm (`euclid_ext2`).

The output is:

```
"[[1, 44, 57], [0, 108, 52], [0, 0, 63]]" ::  "int list list"
```

Which corresponds to the matrix:

$$\begin{pmatrix} 1 & 44 & 57 \\ 0 & 108 & 52 \\ 0 & 0 & 63 \end{pmatrix}$$

Let us note that, due to the fact that we have used the *standard* associates and residues functions for Euclidean domains, both the leading entries and the

elements above them are all positive. The elements above a leading entry are less than such a leading entry (in the example, 57 and 52 are less than 63, which is the leading entry of the third row). The associates function is indeed the absolute value and the residues function is the modulo of the division by the leading coefficient.

In the second example, the Hermite algorithm is applied to the following matrix of polynomials:

$$\begin{pmatrix} 5x^2+4x+3 & x-2 \\ 2x^2-1 & x^3+4x^2+1 \end{pmatrix}$$

The command is:

```
value[code] "let A = list_of_list_to_matrix
  ([[[:3,4,5:],[:-2,1:]],[[:-1,0,2:],[:0,1,4,1:]]])::real poly^2^2
  in matrix_to_list_of_list (Hermite_of A ass_function_euclidean
     res_function_euclidean euclid_ext2)"
```

The corresponding result is:

```
"[[[:1:], [:- (44/89), 31/89, - (68/89), 137/89, 40/89:]],
  [0, [:- (2/5), 4/5, 4, 22/5, 24/5, 1:]]]" ::  "real poly list list"
```

which corresponds to the matrix:

$$\begin{pmatrix} 1 & \frac{40}{89}x^4+\frac{137}{89}x^3-\frac{68}{89}x^2+\frac{31}{89}x-\frac{44}{89} \\ 0 & x^5+\frac{24}{5}x^4+\frac{22}{5}x^3+4x^2+\frac{4}{5}x-\frac{2}{5} \end{pmatrix}$$

We have used again the same *standard* associates and residues functions for Euclidean domains as in the first example. The difference is that now the functions are applied to polynomials instead of integers. Thus, the associates function transforms a polynomial to its corresponding monic one and then the leading entries are monic. The residues (the elements above a leading entry), will be a modulo of the division by their corresponding leading entry (that is, an element above a leading entry will always have degree smaller than the polynomial placed in its corresponding leading entry).

5.3.2 Formalising the Uniqueness of the Hermite Normal Form

Once we have proven the correctness of the algorithm and refined it to immutable arrays, we can prove a well-known theorem: the uniqueness of the Hermite normal form (Theorem 12 in Chapter 2). In order to prove it, we have followed the proof presented in [115, Theorem II.3] by Newman. The only difference is that Newman considers the Hermite normal form as a lower triangular matrix and we consider it to be upper triangular. Our Isabelle statement is the following one:

```
lemma Hermite_unique:
  fixes K::"'a::bezout_domain^'n::mod_type^'n::mod_type"
  assumes A_PH: "A = P ** H"
  and A_QK: "A = Q ** K"
  and inv_A: "invertible A"
  and inv_P: "invertible P"
  and inv_Q: "invertible Q"
  and H: "Hermite associates residues H"
  and K: "Hermite associates residues K"
  shows "H = K"
```

The proof comprises 28 lines in the book [115, Theorem II.3]. Our Isabelle formalisation took us 174 lines, thanks to the previous results that we formalised, and it has been developed carefully to follow firmly the book's proof line by line (see the file *Hermite.thy* in [57]).

5.3.3 Conclusions and Future Work

The Hermite normal form is a canonical matrix analogue of reduced row echelon form, but involving matrices over more general rings. In this section we have shown a formalisation of an algorithm to compute the Hermite normal form of a matrix by means of elementary row operations, taking advantage of some of the results presented in previous sections. More concretely, we have based its formalisation on the framework presented in Chapter 3 and the echelon form algorithm introduced in Section 5.2. The formalisation has been carried out involving matrices over Bézout domains and not only in the well-known case of integer matrices. Following the same approach as in the echelon form formalisation, we have again parametrised the algorithm by functions, in this case to be able to get the different existing versions of the Hermite algorithm. Concretely, this is done thanks to the associates and residues functions, which induce complete sets of associates and residues. We have proven in an affordable number of lines the correctness of the algorithm to obtain the Hermite normal form and we have also refined it to immutable arrays. Furthermore, we have formalised the uniqueness of the Hermite normal form of a matrix as well. As in the other algorithms presented in this thesis, code can be exported to functional languages (SML and Haskell). As far as we know, this is the first ever formalisation of the Hermite normal form and its uniqueness in any interactive theorem prover.

As a future work, it would be desirable to refine the Hermite normal form algorithm to other versions, in order to prevent the exponential growth of the entries in the case of integer matrices (see the work by Kannan and Bachem [95]).

Chapter 6

Formalising in Isabelle/HOL a Simplicial Model for Homotopy Type Theory: a Naive Approach

6.1 Introduction

In this chapter, we give up Linear Algebra algorithms and we present an experiment with Isabelle, its logics and a topic which defines a new trend: Homotopy Type Theory.

Homotopy Type Theory (HoTT) is a pushful area of research presented as a new foundation for Mathematics [150]. This is not the place to undertake a review of HoTT; see [2] for a sample of the activities being developed around HoTT. Let us simply stress that, since HoTT defines a formal language (to model mathematics), the question of its intended semantics is a central one. From the beginning, Voevodsky, the creator of HoTT, looked for a model. The first one, documented in [96], is known as *simplicial model* (other approaches have been proposed; see for instance [28]).

Another distinguishing feature of HoTT is that it is defined in a formalised context (or, at least, in a *formalisable* one). Therefore, it is natural to explore the real implementation of concepts and processes in interactive theorem provers (again, see [2]). In particular, one could ask herself whether Voevodsky's simplicial model could be formalised in a proof assistant. Since one of the characteristics of this model is that it is not *constructive*, one possibility would be to try its formalisation in a classical logic tool, such as Isabelle/HOL. In this chapter we report on a concrete attempt in that direction: formalising Voevodsky's model in Isabelle/HOL.

Once stated our purpose, it is necessary to define a methodology for accomplishing it. Our idea is to keep ourselves in the frame of HOL (that is to say, without imposing any additional axioms) and to reuse code as much as possible. In other words, to build on top of already-developed Isabelle/HOL libraries. Reusing libraries is something desirable, but unfortunately it is not

done as often as expected (see [30]). This strategy gave good results in other projects undertook by us, as it has been shown in the previous chapters of this thesis.

We qualify this approach as *naive*, because we are assuming that previous work in Isabelle/HOL could provide a solid core to model quite complicated novel structures. In fact, as we will show in the chapter, a very small part of the simplicial model can be formalised that way. However, it is not clear at all whether a more sophisticated approach could not find the same kind of obstructions (and, it is indeed clear, the proving effort would be much bigger). Another possible strategy would be to encode Voevodsky's model inside of Isabelle/HOLZF [120], where Zermelo-Fraenkel sets are axiomatised. The drawback is that the Isabelle/HOLZF developments which make use of Zermelo-Fraenkel sets are not comparable in number and depth with those of Isabelle/HOL which make use of typed sets. Nevertheless, we explore cautiously this path in the chapter, with the corresponding code [49].

The rest of the chapter is organised as follows. A brief introduction to other Isabelle's logics different from HOL is presented in Subsection 6.1.1. After that, we present the mathematical infrastructure underlying the simplicial model in Section 6.2, and then the corresponding Isabelle/HOL formalisation is described in Section 6.3. It is worth noting that not all the formalised machinery is required to exhibit the found obstruction; nevertheless we consider that the general subjects formalised could have an interest on their own, for instance, in the field of Category Theory and simplicial sets. Some details of the simplicial model extracted from [96] are reproduced in Section 6.4 (actually, simply the first result from [96] stopped our trial), while Section 6.5 is devoted to show the limitation preventing us from continuing the formalisation in Isabelle/HOL. Our first trials inside the Isabelle/HOLZF framework are described in Subsection 6.5.1. The chapter ends with conclusions. All the Isabelle code written for this experiment is accessible through [48, 49].

6.1.1 HOL, ZF, and HOLZF

As it was explained in Section 2.2, Isabelle [118] is a generic proof assistant, on top of which different logics can be implemented and the most explored of this variety of logics is Higher-Order Logic (or HOL). It is by far the logic where the greatest number of tools (code generation, automatic proof procedures) are available and the one where most of developments are based on. These two reasons encourage us to carry out this part of the thesis also in Isabelle/HOL. Furthermore, all articles (over 200) presented in the Archive of Formal Proofs [3] use HOL as their object logic. Nevertheless, we must give here further details about other logics that are implemented on top of Isabelle, since these logics will play a fundamental role in this chapter.

Zermelo-Fraenkel set theory [83] formulates a theory of sets free of paradoxes such as Russell's paradox. Isabelle's version of the Zermelo-Fraenkel set theory is known as Isabelle/ZF. Its axioms resemble those presented by Suppes [144] and the whole implementation is explained in [125, 126, 129]. Essentially, Isabelle/ZF is based on classical First-Order Logic and the axioms of ZF set theory. Unfortunately, Isabelle/ZF has not been as developed as Isabelle/HOL, it lacks the automatic procedures that Isabelle/HOL posses. On the other hand, it formalises a great part of elementary set theory.

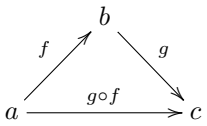
HOLZF [120] is a combination of HOL and ZF (HOLZF = HOL + ZF), that is, Higher-Order Logic extended with ZF axioms. Its corresponding Isabelle implementation is called Isabelle/HOLZF and was developed by Obua. Concretely, it has been implemented following the axioms presented in [74], except from the axiom of separation, since it can be obtained from the axiom of choice that HOL provides by means of the Hilbert choice operator. Although HOLZF has not been as used as HOL, it seems to be stronger than ZF and HOL, since it is very suitable to formalise set-theoretic notions but still offers the advantages of HOL. As far as we know, this logic has only been used in the formalisation of *Partizan Games* [120] and a piece of Category Theory [98]. For a further explanation of this logic see [120, Sect. 3].

6.2 Mathematics Involved

First of all, we present some well-known definitions and theorems which play an important role in the development, following the canonical reference [109] and the article [96]. Voevodsky's simplicial model is based on Category Theory notions, thus our formalisation is useful not only for the concrete case of the simplicial model, but for the general field of Category Theory. Further details, examples, and proofs can be obtained from such references as well. Let us start with the definition of *category*.

Definition 19 (Category)**.** *A category C consists of*

- *A class (a collection) Ob of objects.*
- *A class (a collection) Ar of arrows (also called morphisms or maps) between the objects.*
- *Two functions $dom : Ar \to Ob$ and $cod : Ar \to Ob$.*
- *A composition, which assigns to each pair (g, f) of arrows with $dom\, g = cod\, f$ an arrow $g \circ f : dom\, f \to cod\, g$. That is, if $f : a \to b$ and $g : b \to c$,*



This composition satisfies the following axioms:

  - **Associativity:** *For $f : a \to b$, $g : b \to c$ and $k : c \to d$, the following equality holds:*
$$k \circ (g \circ f) = (k \circ g) \circ f$$
  - **Identity:** *For every object x there exists a morphism $Id_x : x \to x$ called the* identity morphism *for x (also denoted 1_x), such that for every morphism $f : a \to x$ and every morphism $g : x \to b$, we have $Id_x \circ f = f$ and $g \circ Id_x = g$.*

Furthermore, we present the definition of *small category*.

Definition 20 (Small category)**.** *A category C is a small category if the collection of objects Ob and the collection of arrows Ar are sets (instead of being proper classes).*

It is worth noting that there exist many related concepts, such as metacategory (purely axiomatic and does not use set concepts), large and locally small categories and more. Mathematically speaking, most of the examples which will be presented in this chapter are indeed categories. However, only the notion of *small category* has been formalised in Isabelle/HOL [121]. Since we have decided to reuse such a development, all the instances of categories presented in this chapter will be formalised as small categories. Throughout the chapter, we will also denote as Ob_C the collection of objects of a category C. If the category can be inferred from the context, then we will omit it and just say Ob (and equivalently Ar, dom, cod, Id and $\circ$). Let us remark that in the literature it is usually written $X \in C$ and $f \in C$ for $X \in Ob_C$ and $f \in Ar_C$ if that causes no misunderstanding. We will adopt such a convention as well.

Definition 21 (Hom-set)**.** *Given a category C and two objects $X, Y \in Ob_C$, the collection of all morphisms from X to Y is denoted $Hom\, X\, Y$ ($Hom_C\, X\, Y$ if we want to write explicitly the category) and called the hom-set between X and Y. That is, $Hom\, X\, Y = \{f \in C \mid dom\, f = X \wedge\ cod\, f = Y\}$*

Definition 22 (Opposite category)**.** *Given a category C, the opposite category of C (denoted as C^{op}) is a category where*

$$Ob_{C^{op}} = Ob_C$$

$$Ar_{C^{op}} = Ar_C$$

$$dom_{C^{op}} = cod_C$$

$$cod_{C^{op}} = dom_C$$

$$Id_{C^{op}} = Id_C$$

$$g \circ_{C^{op}} f = f \circ_C g$$

Definition 23 (Δ category)**.** *Δ is the small category with objects all finite ordinals (e.g. m, n) and arrows $f : m \to n$ all order-preserving functions ($i \leq j$ implies $fi \leq fj$).*

Definition 24 (Set category)**.** Set category *is the category (denoted as simply Set) where the objects are all sets and the set of arrows all functions between them.*

Let us note that, with this unrestricted definition, *Set* is not a small category, and then it is not covered by Definition 20 but by Definition 19.

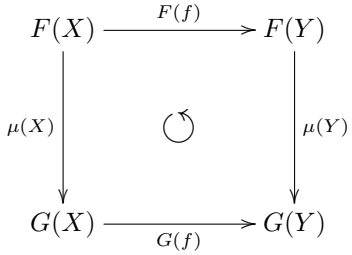
Definition 25 (Functor)**.** *Let C and D be categories. A functor F from C to D is a morphism (or map) of categories that:*

- *associates to each object $X \in Ob_C$ an object $F(X) \in Ob_D$;*
- *associates to each morphism $f : X \to Y \in Ar_C$ a morphism $F(f) : F(X) \to F(Y) \in Ar_D$ such that the following two conditions hold:*

- $F(Id_X) = Id_{F(X)}$ *for every object* $X \in Ob_C$*;*
- $F(g \circ f) = F(g) \circ F(f)$ *for all morphisms* $f : X \to Y$ *and* $g : Y \to Z$*.*

For each $X \in Ob_C$ *and* $f : X \to Y \in Ar_C$*, we will write* $F(X)$ *and* $F(f)$ *if such a notation does not cause confusion. If it does, we will explicitly state whether the functor is applied to objects,* $F_o(X)$*, or arrows,* $F_a(f)$*.*

Definition 26 (Natural Transformation)**.** *Given two functors* $F, G : C \to D$*, where* C *and* D *are two categories, a natural transformation (often called a* morphism of functors*) is a function* $\mu : Ob_C \to Ar_D$ *which assigns to each object of* C *an arrow of* D *in such a way that every arrow* $f : X \to Y$ *in* Ar_C *yields a diagram:*



Equivalently, in another notation:

$$\forall X, Y \in Ob_C.\ \forall f \in Hom_C\, X\, Y.\ G(f) \circ_D \mu(X) = \mu(Y) \circ_D F(f)$$

Definition 27 (Functor category)**.** *Given two categories* C *and* D*, it is possible to construct the* functor category *of* C *and* D *whose objects are the functors* $F : C \to D$ *and arrows the natural transformations between two such functors.*

A particular, but important example of functor category is the case where the objects are the functors $F : \Delta^{Op} \to Set$. This functor category is often called the *simplicial set category*, and denoted as *sSet*.

Definition 28 (Simplicial Set)**.** *A simplicial set* K *is a graded set indexed on the naturals together with maps* $\partial_i : K_q \to K_{q-1}$ *and* $s_i : K_q \to K_{q+1}$*,* $0 \leq i \leq q$*, which satisfy the following identities:*

$$\partial_i \partial_j = \partial_{j-1} \partial_i \text{ if } i < j$$
$$s_i s_j = s_{j+1} s_i \text{ if } i \leq j$$
$$\partial_i s_j = s_{j-1} \partial_i \text{ if } i < j$$
$$\partial_j s_j = identity = \partial_{j+1} s_j$$
$$\partial_i s_j = s_{j-1} \partial_{i-1} \text{ if } i > j + 1$$

The elements of K_q *are called q-simplices. The* ∂_i *and* s_i *maps are called face and degeneracy operators.*

Definition 29 (Simplicial morphism)**.** *A simplicial morphism (also called simplicial map or just morphism)* $f : K \to L$ *is a map of degree zero of graded sets which commutes with the face and degeneracy operators, that is,* f *consists of* $f_q : K_q \to L_q$ *and*

$$f_q \partial_i = \partial_i f_{q+1}$$
$$f_q s_i = s_i f_{q-1}$$

Definition 30 (Simplicial set category: *osSet*)**.** *The* simplicial set category *is the category (denoted as osSet) where the set of objects consists of all simplicial sets and arrows are all morphisms between them.*

Theorem 14 (Equivalence between *sSet* and *osSet*)**.** *There exist two functors* $F : sSet \to osSet$ *and* $F' : osSet \to sSet$ *such that*

$$F \circ_{sSet} F' = Id_{osSet}$$

$$F' \circ_{osSet} F = Id_{sSet}$$

Since the previous theorem states that *osSet* and *sSet* are isomorphic, both are usually referred as simplicial set category. Even more, both are normally denoted as *sSet* indistinctly, but it is worth noting that their definitions are different and sometimes it is more convenient to use one of them than the other one. This is the reason why we prefer to denote them in a different way, to make it clear in each moment which definition we are working with.

Theorem 15 (Product of simplicial sets)**.**

- *Let* Y *and* X *be simplicial sets with the corresponding face and degeneracy operators* ∂_Y, s_Y, ∂_X *and* s_X. *Then, the following construction is a simplicial set (also known as the cartesian product of simplicial sets):*

$$\begin{aligned} Y \times X &= \{(y, x).\ y \in Y \wedge x \in X\} \\ \partial_{Y \times X} &= (\lambda(y, x) \in Y \times X.\ (\partial_Y(y), \partial_X(x)) \\ S_{Y \times X} &= (\lambda(y, x) \in Y \times X.\ (s_Y(y), s_X(x))) \end{aligned}$$

- *Let* Y_1, Y_2 *and* X *be simplicial sets together with* ∂_{Y_1}, s_{Y_1}, ∂_{Y_2}, s_{Y_2}, ∂_X *and* s_X *as the corresponding face and degeneracy operators. Let* $t : Y_1 \to X$ *and* $f : Y_2 \to X$ *be morphisms. Then the following construction is a simplicial set (also known as the pullback of simplicial sets):*

$$\begin{aligned} Y_1 \times_{(t,f)} Y_2 &= \{(y_1, y_2).\ y_1 \in Y_1 \wedge y_2 \in Y_2 \wedge t(y_1) = f(y_2)\} \\ \partial_{Y_1 \times_{(t,f)} Y_2} &= (\lambda(y_1, y_2) \in Y_1 \times_{(t,f)} Y_2.\ (\partial_{Y_1}(y_1), \partial_{Y_2}(y_2)) \\ s_{Y_1 \times_{(t,f)} Y_2} &= (\lambda(y_1, y_2) \in Y_1 \times_{(t,f)} Y_2.\ (s_{Y_1}(y_1), s_{Y_2}(y_2)) \end{aligned}$$

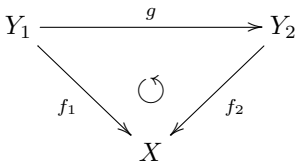
Definition 31 (Pullback on objects)**.** *Let* Y_1, Y_2, X *be simplicial sets,* $t : Y_1 \to X$ *and* $f : Y_2 \to X$ *morphisms. Then, the* pullback on objects *is defined by means of the following diagram:*

$$\begin{array}{ccc} Y_1 \times_{(t,f)} Y_2 & \xrightarrow{\Pi_2} & Y_2 \\ \Big\downarrow{\scriptstyle \Pi_1} & & \Big\downarrow{\scriptstyle f} \\ Y_1 & \xrightarrow[t]{} & X \end{array}$$

Following [96], we have only formalised a particular case of pullback between arrows: that related to well-ordered morphisms.

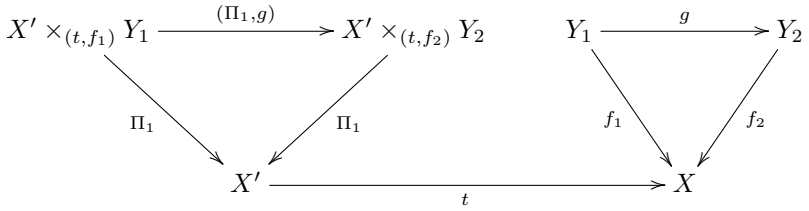
Definition 32 (Well-ordered morphism)**.** *Let X and Y be simplicial sets. A* well-ordered morphism *$f : X \to Y$ is a pair consisting of a morphism into X (also denoted by f) and a function assigning to each simplex $x \in X_n$ a well-ordering on the fiber $Y_x := f^{-1}(x) \subseteq Y_n$.*

Definition 33 (Isomorphism of well-ordered morphisms)**.** *If $f_1 : Y_1 \to X$ and $f_2 : Y_2 \to X$ are well-ordered morphisms, an isomorphism of well-ordered morphisms from f_1 to f_2 is an isomorphism $Y_1 \cong Y_2$ preserving the well-orderings in the fibers. That is,*



where $f_2 \circ g = f_1$.

Definition 34 (Pullback on morphisms)**.** *Let X', X, Y_1, Y_2 be simplicial sets, $f_1 : Y_1 \to X$ and $f_2 : Y_2 \to X$ well-ordered morphisms, $t : X' \to X$ a morphism and $g : Y_1 \to Y_2$ an isomorphism between the well-ordered morphisms f_1 and f_2. Then, the pullback on morphisms is defined as follows:*



Theorem 16 (Yoneda embedding)**.** *There exists a functor $y : \Delta \to (\Delta^{op} \to Set)$ (equivalently $y : \Delta \to sSet$), called the* Yoneda embedding.

6.3 Formalising the Infrastructure

The previous section presents the mathematical definitions and results which are fundamental for the simplicial model. Now we show how we have carried out their formalisation. In order to formalise such an infrastructure in Isabelle/HOL, we try to reuse some already-developed libraries. In our case, we make use of an existing development about categories published in the Isabelle Archive of Formal Proofs [122]. This library is based on HOL and includes relevant facts such as the definition of (small) categories, functors and the Yoneda's lemma. O'Keefe [121] explains the formalisation of such results. This section explains the formalisation of the results presented in the previous section, but there also exist some other intermediate facts which are not explained here, although they have been included in our library since they are basic results of Category Theory (such as composition of morphisms is a morphism, composition of isomorphisms between simplicial sets is an isomorphism and more). Throughout this section we just present the main statements, omitting the proofs.

Let us note that the simplicial model presented in [96] involves *categories* (see Definition 19), but our formalisation is based on the more specific *small categories*, where *Ob* and *Ar* are sets instead of being proper classes. The definition of small categories presented in [122] has been named in Isabelle/HOL as `category` and it is the following one:

```
record ('o, 'a) category =
  ob :: "'o set" ("Ob\<^sub>1"  70)
  ar :: "'a set" ("Ar\<^sub>1"  70)
  dom :: "'a ⇒ 'o" ("Dom\<^sub>1 _" [81] 70)
  cod :: "'a ⇒ 'o" ("Cod\<^sub>1 _" [81] 70)
  id :: "'o ⇒ 'a" ("Id\<^sub>1 _" [81] 80)
  comp :: "'a ⇒ 'a ⇒ 'a"  (infixl "⋅\<^sub>1" 60)

definition
  hom :: "[('o,'a,'m) category_scheme, 'o, 'o] ⇒ 'a set"
    ("Hom\<^sub>1 _ _" [81,81] 80) where
  "hom CC A B = { f. f∈ar CC & dom CC f = A & cod CC f = B }"

locale category =
  fixes CC (structure)
  assumes dom_object [intro]:
  "f ∈ Ar ⟹ Dom f ∈ Ob"
  and cod_object [intro]:
  "f ∈ Ar ⟹ Cod f ∈ Ob"
  and id_left [simp]:
  "f ∈ Ar ⟹ Id (Cod f) ⋅ f = f"
  and id_right [simp]:
  "f ∈ Ar ⟹ f ⋅ Id (Dom f) = f"
  and id_hom [intro]:
  "A ∈ Ob ⟹ Id A ∈ Hom A A"
  and comp_types [intro]:
  "⋀A B C. (comp CC) : (Hom B C) → (Hom A B) → (Hom A C)"
  and comp_associative [simp]:
  "f ∈ Ar ⟹ g ∈ Ar ⟹ h ∈ Ar
  ⟹ Cod h = Dom g ⟹ Cod g = Dom f
  ⟹ f ⋅ (g ⋅ h) = (f ⋅ g) ⋅ h"
```

As it can be seen, the definition is based on a **record**, where the arity and types of each operation are fixed, and a **locale**, where the properties about such operations are formulated. Roughly speaking, this implementation does not straightly follows Definition 20, since the elements of the set of objects (as well as the elements in the set of arrows) must be of the same type. All the examples of categories presented in Section 6.2 will be implemented in Isabelle/HOL based on the definition of small category presented above. Since such a frame of *small categories* has been chosen in the formalisation, from here on we will refer to them simply as *categories*.

Although the definition of *small category* was present in the library, the definition of opposite category was not. We have defined it in a quite direct way and prove the corresponding interpretation as a category in Isabelle:

```
context category
begin
```

```
definition
  "op_cat =
  (|
    ob = ob CC,
    ar = ar CC,
    dom = cod CC,
    cod = dom CC,
    id = id CC,
    comp = (λa b. comp CC b a)
  |)"

interpretation op_cat: category op_cat
proof
  ...
qed
```

In addition, we have defined and proven the properties of the $\Delta-$category. As it is done in many references, we identify a natural number n with the corresponding set $\{0..n\}$. That is, when referring $n \in Ob_{\Delta}$ we mean $\{0..n\} \in Ob_{\Delta}$. In our implementation, the elements of the set of arrows will be triples (f, m, n) where $f : m \to n$ and it satisfies the conditions presented in Definition 23. The use of triples is mandatory, otherwise, domain and codomain of an arrow f would be unknown. The Isabelle predicate `f ∈ extensional {0..m}` just means that `f` returns `undefined` outside `{0..m}` (the domain).

```
definition "delta_ob = {{0..n} | n. n ∈ (UNIV::nat set)}"
definition "delta_ar = {(f,m,n) |(f::nat⇒nat) m n. f ∈ {0..m} → {0..n}
  ∧ (∀i j. j∈ {0..m} ∧ i≤j ⟶ f i ≤ f j) ∧ f ∈ extensional {0..m}}"
definition "delta_dom (f :: (nat⇒nat) × nat × nat) = {0..fst (snd f)}"
definition "delta_cod f = {0..snd (snd f)}"
definition "delta_id = (λs∈delta_ob.(λx∈s. x, card s - 1, card s - 1))"

definition "delta_comp G F = (let f = fst F; m=fst (snd F); g= fst G;
n'=snd (snd G) in ((compose (delta_dom F) g f), m ,n'))"

definition
  "delta_cat=
  (|
    ob = delta_ob,
    ar = delta_ar,
    dom = delta_dom,
    cod = delta_cod,
    id = delta_id,
    comp = delta_comp
  |)"

interpretation delta: category delta_cat
proof
  ...
qed
```

The category of sets (Set) is an essential part in the simplicial model. It is

already defined in the Isabelle/HOL development on which we are basing our work:

```
record 'c set_arrow =
  set_dom :: "'c set"
  set_func :: "'c ⇒ 'c"
  set_cod :: "'c set"

definition
  set_arrow :: "['c set, 'c set_arrow] ⇒ bool" where
  "set_arrow U f ⟷ set_dom f ⊆ U & set_cod f ⊆ U
    & (set_func f): (set_dom f) → (set_cod f)
    & set_func f ∈ extensional (set_dom f)"

definition
  set_cat :: "'c set ⇒ ('c set, 'c set_arrow) category" where
  "set_cat U =
  (|
    ob = Pow U,
    ar = {f. set_arrow U f},
    dom = set_dom,
    cod = set_cod,
    id = set_id U,
    comp = set_comp
  |)"
```

Let us note that a variable set `U` is used. This set fixes the objects and arrows of the category: objects are the powerset of such a set `U`, arrows will be functions between such objects. We will usually work fixing `U = UNIV`, that is: the universe of all sets of a (fixed) type. However, that is not exactly the definition of the *Set*-category, since it is more related to what in [109] is called **Ens**: "category of all sets and functions within a (variable) set `U`", which is a *small* category.

It is worth noting that here we are limited by the type system: the variable set `U` will fix the underlying type `'c` of the category, since its objects will be subsets of `Pow U`. That is, objects of the *Set*-category will be sets over a fixed type `'c` and thus it is not possible to consider objects over any other type within the same category. Same occurs with the arrows: they are functions of type `'c ⇒ 'c` as arrows are maps between objects. To sum up, once `U` is fixed, objects will have the same type `'c set` (the type of the set of objects) and we cannot mix them with other objects of a type non-unifiable with `'c set`, although the mathematical definition of *Set* allows it.

Let us show an example. Let $A = \{1, 2, 3\}$ be a set of natural numbers and $B = \{True, False\}$ a boolean set. Both sets belong to the category *Set*, but in the Isabelle/HOL library we are using do not because types are different (in such a library they would belong to different categories). Moreover, the following function belongs to the set of arrows of the *Set*-category using the mathematical definition.

$$\begin{aligned} f : A &\longrightarrow B \\ 1 &\longrightarrow True \\ 2 &\longrightarrow True \\ 3 &\longrightarrow False \end{aligned}$$

However, it does not belong to the set of arrows of any *Set*-category within the presented Isabelle/HOL implementation. Our intention is to push the formalisation as far as possible in this constrained context.

The concepts of functor and natural transformation were already defined in the library. We have included an interesting property: if $F : AA \to BB$ is a functor, then we can define another functor $F^{op} : AA^{op} \to BB^{op}$. Indeed, such a functor is F (the proof can be found in the file *Yoneda_Embedding.thy* of the development [48]):

```
lemma op_functor:
assumes "Functor F : AA ⟶ BB"
shows "Functor F : (AA.op_cat) ⟶ (BB.op_cat)"
```

We have also added the definition of *functor category*, see Definition 27. In a similar way to what we did for the arrows in the Δ-category, we have defined the arrows of the functor category as a triple (μ, F, G) where μ is the natural transformation between the functors F and G.

```
definition "functor_cat_ob = {F | F. Functor F: AA ⟶ BB}"
definition "functor_cat_ar = {(u,F,G)| u F G. u : F ⇒ G in Func(AA,
BB)}"
definition "functor_cat_dom u = fst (snd u)"
definition "functor_cat_cod u = snd (snd u)"
definition "functor_cat_id = (λF ∈ functor_cat_ob. (λx∈ ob AA. id BB
(F₀ x),F,F))"
definition "functor_cat_comp V U = (let
  u = fst U; F = fst (snd U); G = snd (snd U);
  v = fst V; H = fst (snd V); I = (snd (snd V))
  in (λx∈ob AA. (v x) ⋆_BB (u x),F,I))"

definition
  "functor_cat=
  (|
    ob = functor_cat_ob,
    ar = functor_cat_ar,
    dom = functor_cat_dom,
    cod = functor_cat_cod,
    id = functor_cat_id,
    comp = functor_cat_comp
  |)"

interpretation functor_cat: category functor_cat
proof
  ...
qed
```

As an instance of the functor category, we define the simplicial set category ($sSet$) which is the category of functors between Δ^{op} and Set:

```
interpretation sSet: two_cats delta.op_cat "set_cat (UNIV::'c set)"
proof
  ...
qed
```

Now we can move forward to simplicial sets according to Definition 28. There was nothing developed in Isabelle/HOL about such a structure. We have implemented it by means of a **class** as follows:

```
class simplicial_set =
  fixes X::"nat ⇒ 'a set"
  and ∂ :: "nat ⇒ nat ⇒ 'a ⇒ 'a"
  and s :: "nat ⇒ nat ⇒ 'a ⇒ 'a"
  assumes ∂_pi: "i≤q+1 ⟶ ∂ (q+1) i ∈ X (q+1) → X q"
  and s_pi: "i≤q ⟶ s q i ∈ X q → X (q + 1)"
  and condition1: "q≠0 ∧ i<j ∧ j ≤ q + 1 ∧ x ∈ X (q + 1)
      ⟶ (∂ q i ∘ ∂ (q+1) j) x = (∂ q (j - 1) ∘ ∂ (q+1) i) x"
  and condition2: "i≤j ∧ j≤q ∧ x ∈ X q
      ⟶ (s (q + 1) i ∘ s q j) x = (s (q+1) (j+1) ∘ s q i) x"
  and condition3: "i<j ∧ j≤q ∧ x ∈ X q
      ⟶ (∂ (q + 1) i ∘ s q j) x = (s (q - 1) (j - 1) ∘ ∂ q i) x"
  and condition4: "j≤q ∧ x ∈ X q ⟶ (∂ (q+1) (j+1) ∘ s q j) x = x"
  and condition5: "j≤q ∧ x ∈ X q ⟶ (∂ (q+1) j ∘ s q j) x = x"
  and condition6: "i>j+1 ∧ i≤q ∧ x ∈ X q
      ⟶ (∂ (q + 1) i ∘ s q j) x = (s (q - 1) j ∘ ∂ q (i - 1)) x"
```

HOL functions are total, but ∂ and s are not. Hence, it has been necessary to take care of the domain of the composition in all conditions, imposing that $x \in X_q$ (or X_{q+1} depending on the formulated property).

The definition of a simplicial morphism (see Definition 29) is also straightforward. We have created a **locale** called *two_simplicial_sets* where two simplicial sets Y and X are fixed, then a simplicial morphism is defined inside such a **locale**. Once again, we have to explicitly introduce in the properties the domain and codomain of the morphism, specially when defining the properties. Throughout our work, we have defined some locales where we fix several simplicial sets with their corresponding involved morphisms to avoid cumbersome notation. As we have already said, we are working in this case within a **locale** where two simplicial sets are fixed (Y and X).

```
definition "morphism f = (∀n. f n ∈ Func (Y n) (X n)
  ∧ (∀i x. i≤n+1 ∧ x ∈ Y (n+1) ⟶ (f n ∘ ∂y (n + 1) i) x = (∂x (n +
1) i ∘ f (n + 1)) x)
  ∧ (∀i x. i≤n ∧ x ∈ Y n ⟶ (f (n + 1) ∘ sy n i) x = (sx n i ∘ f n)
x))"
```

Furthermore, we have had to require the morphism to return *undefined* outside the source simplicial set (the constant *undefined::'a* describes an arbitrary element of a type *'a*). Unfortunately, to require the morphisms to go from Y to

X is not enough, since that will give room to problems about equality between functions when working outside the domain. This is done thanks to the existing definition `Func`:

```
Func A B = {f. (∀a∈A. f a ∈ B) ∧ (∀a. a ∉ A ⟶ f a = undefined)}
```

Well-ordered morphisms are the first notion that appears specifically in the simplicial model article [96]. In order to implement them in Isabelle, previously we have to define the concept of an *isomorphism between simplicial sets*.

```
definition "simplicial_set_iso f = (morphism f
  ∧ (∃g. X_Y.morphism g ∧
  (∀n. compose (X n) (f n) (g n) = restrict id (X n)
  ∧ compose (Y n) (g n) (f n) = restrict id (Y n))))"
```

In the previous definition, `morphism f` represents a morphism $f : Y \to X$, `X_Y.morphism g` the inverse morphism $g : X \to Y$ and `compose (Y n) (g n) (f n) = restrict id (Y n)` means that the composition of g_n and f_n restricted to Y_n is equal to the identity restricted to Y_n.

Moreover, we have formalised that if $f : Y \to X$ is an isomorphism between simplicial sets which respects the well-order in the fibers, then its inverse will also respect the well-order in the fibers.

Well-ordered morphisms (see Definition 32) are then implemented inside the **locale** `two_simplicial_sets` as follows:

```
definition "well_ordered_morphism F = (let f=fst F; g=snd F in morphism f
  ∧ (∀n. g n ∈ Pi (X n::'a set) (%x. {r. well_order_on {y::'b. y ∈ Y n
  ∧ f n y = x} r})))"
```

Domain and codomain of g_n have been determined thanks to the existing definition `Pi`. In contrast to the use of `Func`, `Pi` does not state anything about what happens outside the domain:

```
Pi A B = {f. ∀x. x ∈ A ⟶ f x ∈ B x}
```

Making use of the definition of well-ordered morphism, we can implement the concept of isomorphism of well-ordered morphisms (see Definition 33):

```
definition "iso_between_morphism f1 f2 g =
(
 (*Both f1 and f2 are a well ordered morphism*)
    Y1_X.well_ordered_morphism f1 ∧ Y2_X.well_ordered_morphism f2
 (*Isomorphism between simplicial set*)
    ∧ Y1_Y2.simplicial_set_iso g
 (*Commute*)
    ∧ (∀n::nat. ∀y ∈ Y1 n. (fst(f2) n) (g n y) = (fst(f1) n) y)
 (*Isomorphism between fibers that respects the well-order on them*)
    ∧ (∀n::nat. ∀x ∈ X n. iso (snd f1 n x) (snd f2 n x) (g n))
 )"
```

Now it is turn to define the $osSet$ category. Again, we have a similar problem

to the one presented in the *Set*-category: once we fix a type, each object (each simplicial set) must be of such a type. We just show here the set of objects and the set of arrows, the rest of operations of the category are defined in a straightforward way according to these sets (see Definition 30):

```
definition "osSet_ob = {(X,∂,s). class.simplicial_set X ∂ s}"

definition "osSet_ar = {(f,(Y,∂y,sy),(X,∂x,sx))| f Y ∂y sy X ∂x sx.
  class.simplicial_set Y ∂y sy ∧ class.simplicial_set X ∂x sx ∧
  two_simplicial_sets.morphism Y ∂y sy X ∂x sx f}"

definition "osSet_dom f = fst (snd f)"
definition "osSet_cod f = snd (snd f)"

definition "osSet_id =
  (λ(X,∂,s)∈osSet_ob. ((λn. λx ∈(X n). x),(X,∂,s),(X,∂,s)))"

definition "osSet_comp g f = (let f'=fst f; X=fst (osSet_dom f); g'=fst g
  in (λn::nat. compose (X n) (g' n) (f' n), osSet_dom f, osSet_cod g))"

definition "osSet =
(|
    ob = osSet_ob,
    ar = osSet_ar,
    dom = osSet_dom,
    cod = osSet_cod,
    id = osSet_id,
    comp = osSet_comp
  |)"

interpretation osSet: category osSet
proof
  ...
qed
```

Again, arrows are defined by means of a triple *(morphism, domain, codomain)*. Here, domain and codomain are simplicial sets, which are again a triple *(simplicial set, faces, degeneracies)*.

A constant approach in our work is to try to prove the properties as general as possible in Isabelle. For instance, in this case once we fix the category, the simplicial sets (the objects of `osSet`) will be of the same type. Nevertheless, we can prove the properties that a category requires without imposing both simplicial sets to be of the same type, which is more similar to what is presented in mathematical books. This way, we have demonstrated the properties in general (involving simplicial sets of different types), but when working with the Isabelle/HOL definition of category, types will be unified. Let us show an example of our approach when working in the `osSet` category.

The following property states that the domain of an arrow in `osSet` belongs to the objects. If we do not fix the category, that is, if we work with the general definitions `osSet_ob`, `osSet_ar`, ..., we have the following statement:

```
lemma osSet_dom_in_delta_ob:
```

```
fixes f::"(nat ⇒ 'a ⇒ 'b) × ((nat ⇒ 'a set) ×
      (nat ⇒ nat ⇒ 'a ⇒ 'a) × (nat ⇒ nat ⇒ 'a ⇒ 'a)) ×
      (nat ⇒ 'b set) × (nat ⇒ nat ⇒ 'b ⇒ 'b) × (nat ⇒ nat ⇒ 'b
⇒ 'b)"
assumes "f ∈ osSet_ar"
shows "osSet_dom f ∈ osSet_ob"
```

If one takes a look at the type of the arrow `f`, it can be seen that two free type variables appear: `'a` and `'b` (the type of the arrows is cumbersome because the set of arrows is a triple). These two free type variables represent the type of the source simplicial set and the final simplicial set, which can be now of different type. Hence, we have proven the property in general, somehow more related with the mathematical definition.

However, once we fix the category (that is, when working with Ob_{osSet}, Ar_{osSet},...), only one free type variable appears. It is just a particular case of the previous result and we can prove it as a corollary.

```
corollary osSet_dom_in_delta_ob_cat:
fixes f::"(nat ⇒ 'a ⇒ 'a) × ((nat ⇒ 'a set) ×
      (nat ⇒ nat ⇒ 'a ⇒ 'a) × (nat ⇒ nat ⇒ 'a ⇒ 'a)) ×
      (nat ⇒ 'a set) × (nat ⇒ nat ⇒ 'a ⇒ 'a) × (nat ⇒ nat ⇒ 'a
⇒ 'a)"
assumes "f ∈ Ar_osSet"
shows "Dom_osSet f ∈ Ob_osSet"
using assms osSet_dom_in_delta_ob
  by (simp add: osSet_def)
```

With this approach, we try to keep the statements as general as possible, minimizing the limitations of the type system when working with categories.

Theorem 14 shows the equivalence between *osSet* and *sSet*. As we will explain in Section 6.5, only the transformation from *sSet* to *osSet* is necessary in our development. Let us show how we have defined such a functor, that we will call `from_sSet`. First of all, we have to define two morphisms $\delta_i^n : \Delta_{n-1} \to \Delta_n$ and $\sigma_i^n : \Delta_{n+1} \to \Delta_n$ which are arrows in Δ, and also in Δ^{op} as long as the corresponding change between domain and codomain is done. Both are defined in [111]. We have implemented them in Isabelle/HOL by means of the following definitions (outside the domain, both functions return undefined):

```
definition "δ n i = (if n=0 then undefined else (λj::nat. if j<i ∧ i≤n
then j else if j<n then j + 1 else undefined))"

definition "σ n i = (λj::nat. if j≤i ∧ i≤n then j else  if j≤n+1 then
j - 1 else undefined)"
```

After that, we define the functor `from_sSet` for objects. We base the definition on the previous functions δ and σ. Given an object in *sSet* (a functor $F : \Delta^{op} \to Set$), we have to get another object in *osSet*. That is, a triple (Y, ∂, s) is sought. This is done by means of the following definition:

```
definition "from_sSet_ob = (λF ∈ sSet_ob.
  (λn. F_o {0..n},
  λn i. if n = 0 then undefined else set_func (F_a (δ n i, n - 1,n)),
```

```
λn i. set_func (Fa (σ n i, n+1,n))))"
```

Some remarks:

- F is a functor from Δ^{op} to Set. So, given an arrow in Δ^{op} (such as δ_i^n and σ_i^n) returns an arrow in Set: a triple *(domain, function, codomain)*. We have to use `set_func` to get only the function and define faces and degeneracies.
- As HOL functions are total, we have to deal with the case `0` when working with faces, defining it as `undefined`, since it does not make sense.

Now we want to transform arrows of $sSet$ (a triple (μ,F,G) where μ is a natural transformation between the functors F and G), to arrows in $osSet$ (a triple (f,Y,X) where f is a morphism between two simplicial sets Y and X). In this case, the morphism is defined as `(λn::nat. set_func (u {0..n}))` and the simplicial sets that are the domain and the codomain of the morphism will be obtained from the function `from_sSet_ob` applied to the functors F and G.

```
definition "from_sSet_ar = (λ(u,F,G) ∈ sSet_ar.
  ((λn::nat. set_func (u {0..n})),(from_sSet_ob F),(from_sSet_ob G)))"
```

Finally, we can define `from_sSet` and prove that it is really a functor between the categories `sSet` and `osSet`:

```
definition "from_sSet = ⦇om = from_sSet_ob, am = from_sSet_ar⦈"

lemma from_sSet_functor:
  "Functor from_sSet : sSet ⟶ osSet"
```

Theorem 15 presents some examples of products of simplicial sets that are simplicial sets. Both results are stated in Isabelle/HOL as follows:

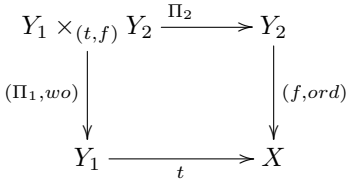
```
sublocale simplicial_set_prod:
  simplicial_set "(λn. (Y n × X n)) "
                      "(λi n x. (∂y i n (fst x), ∂x i n (snd x))) "
                      "(λi n x. (sy i n (fst x), sx i n (snd x)))"

sublocale Y1_times_Y2_tf: simplicial_set
          "(λn. {(y1,y2). y1 ∈ Y1 n ∧ y2 ∈ Y2 n ∧ t n y1 = f n y2})"
          "(λi n x. (∂y1 i n (fst x), ∂y2 i n (snd x)))"
          "(λi n x. (sy1 i n (fst x), sy2 i n (snd x)))"
```

The last **sublocale** is essential for the pullback on objects presented in Definition 31. Essentially, we must prove that $Y_1 \times_{(t,f)} Y_2 \xrightarrow{\Pi_1} Y_1$ is a morphism to complete such a diagram:

```
lemma Y1_times_Y2_tf_Y1_morphism_fst:
  shows "Y1_times_Y2_tf_Y1.morphism (λn. λx∈{(y1,y2). y1 ∈ Y1 n ∧ y2
∈ Y2 n ∧ t n y1 = f n y2}. fst x)"
```

Indeed, we can prove $Y_1 \times_{(t,f)} Y_2 \xrightarrow{(\Pi_1, wo)} Y_2$ is a well-ordered morphism which is somehow more complicated. We want to prove that the following

diagram commutes:



where t is a morphism and (f, ord) is a well-ordered morphism. We have to define the function wo which is the one that given an element x in Y_{1_n} returns a well-order in the fiber $(Y_1 \times_{(t,f)} Y_2)_x := {\Pi_1}^{-1}(x) \subseteq (Y_1 \times_{(t,f)} Y_2)_n$. We have named this function as `X_Y1_Y2_tf_g` and defined it in Isabelle/HOL as follows:

```
definition "X_Y1_Y2_tf_g
  = (λn y1. {((a,b),(c,d)). a=y1 ∧ c=y1 ∧ y1 ∈ Y1 n ∧ b ∈ Y2 n ∧ d ∈
Y2 n ∧ t n y1 = f n b ∧ t n y1 = f n d ∧ (b,d) ∈ ord n (t n y1)})"
```

The first thing to do is to prove that it returns a well-order:

```
lemma Well_order_X_Y1_Y2_tf_g:
  assumes x: "x ∈ Y1 n"
  shows "Well_order (X_Y1_Y2_tf_g n x)"
```

Once that is done, we prove

$$Y_1 \times_{(t,f)} Y_2 \xrightarrow{(\Pi_1, wo)} Y_1$$

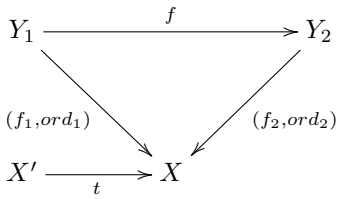
to be a well-ordered morphism (where wo denotes our function `X_Y1_Y2_tf_g`):

```
lemma Y1_times_Y2_tf_Y1_well_ordered_morphism_fst:
shows "X_Y1_Y2_tf.Y1_times_Y2_tf_Y1.well_ordered_morphism
      ((λn. λx∈{(y1,y2). y1 ∈ Y1 n ∧ y2 ∈ Y2 n ∧ t n y1 = f n y2}.
fst x), X_Y1_Y2_tf_g)"
```

Finally, we can define the pullback completely:

```
definition "pullback_set = (λn. {(y1,y2). y1 ∈ Y1 n ∧ y2 ∈ Y2 n ∧ t n
y1 = f n y2})"
definition "pullback_∂ = (λi n x. (∂y1 i n (fst x), ∂y2 i n (snd x)))"
definition "pullback_s = (λi n x. (sy1 i n (fst x), sy2 i n (snd x)))"
definition "pullback = ((λn. λx ∈ pullback_set n. fst x)::(nat ⇒ 'b ×
'c ⇒ 'b), X_Y1_Y2_tf_g)"

corollary well_ordered_morphism_pullback:
  shows "X_Y1_Y2_tf.Y1_times_Y2_tf_Y1.well_ordered_morphism pullback"
```

To prove the pullback on morphisms, we have defined a **locale** where four simplicial sets are fixed (Y_1, Y_2, X' and X), $t : X' \to X$ is a morphism, and f is an isomorphism, following the diagram:



We have proven that $X' \times_{(t,f_1)} Y_1 \xrightarrow{(\Pi_1, f)} X' \times_{(t,f_2)} Y_2$ is an isomorphism. The statement in Isabelle/HOL looks as follows:

```
lemma iso_between_morphism_pullback:
shows "X'_X'Y1_X'Y2.iso_between_morphism (X_X'_Y1_t_f1.pullback)
(X_X'_Y2_t_f2.pullback) (λn. λx∈X_X'_Y1_t_f1.pullback_set n. (fst x,
f n (snd x)))"
```

Finally, the Yoneda embedding is a functor between Δ and $sSet$. Again, we define the functor in two blocks: objects and arrows. But previously we define a couple of auxiliary functions:

Definition 35. *Let m and n be in Ob_Δ. Then, Δ_n^m is the set of morphisms between m and n.*

Definition 36. *A function F is defined as:*

- *Given an object n in Δ, F returns an object in $sSet$: a function that given an object m in Δ^{op} returns the set of morphisms between m and n, that is Δ_n^m.*
- *Given an object n in Δ and a morphism in Δ^{op} (a function $f : m_2 \to m_1$), then F returns the arrow belonging to Set which is a function that given another function $g : m_1 \to n$, then it returns the composition $g \circ f : m_2 \to n$.*

They have been implemented in Isabelle/HOL as follows:

```
definition "Δ = (λn ∈ ob delta_cat. λm ∈ ob delta_cat. {(f,card n -
1,card m - 1)|f. (f,card n - 1,card m - 1) ∈ delta_ar})"
definition "F = (λn ∈ ob delta_cat.
    ⦇om= (λm ∈ ob delta.op_cat. Δ m n),
     am= (λf ∈ ar delta.op_cat.
         let m1=delta_cod f; m2 = delta_dom f (*f ∈ Δ m2 m1*)
         in
            ⦇set_dom = Δ m1 n,
             set_func = (λg ∈ Δ m1 n. (delta_comp g f)),
             set_cod = Δ m2 n⦈)⦈)"
```

We can prove that $(F\, n)$ is a functor between Δ^{op} and Set:

```
lemma Functor_F: "Functor F {0..n} : delta.op_cat ⟶ (set_cat UNIV)"
```

Taking advantage of the previous auxiliary definitions, we can implement the Yoneda embedding. The Yoneda embedding is a functor $y : \Delta \to sSet$ which is defined as follows:

- Given an object in Δ, returns an object in $sSet$ (a functor). That is, given $n \in Ob_{\Delta}$ it returns $F\ n$.
- Given an arrow in Δ, $f : n \to m$, then y has to return an arrow in $sSet$ (a natural transformation between two functors: that is, a function that given an object in Δ^{op} returns an arrow in Set). Fixed an element k belonging to the objects in Δ^{op}, then such a natural transformation will be a function that given a g in $F\ n\ k$ (which is equal to Δ^k_n, the set of functions from k to n), returns the composition $f \circ g : k \to m$. The functors are $F\ n$ and $F\ m$.

The corresponding Isabelle/HOL implementation is:

```
definition  "yoneda_embedding =
    (|om = F,
     am = (λf∈ar delta_cat.
           let n=delta_dom f; m=delta_cod f
           in
             ((λk∈ ob delta.op_cat.
               (|set_dom = (F n) ∘ k,
                set_func=(λg ∈ (F n) ∘ k. delta_comp f g),
                set_cod=(F m) ∘ k|), (F n),(F m))))|)"
```

We have always had to work with triples, since the arrows in both `Set` and `sSet` are defined in that way. The final statement looks like this (its complete proof is available from the file *Yoneda_Embedding.thy* of the development [48]):

```
lemma Functor_Yoneda_Embedding:
  "Functor yoneda_embedding : delta_cat ⟶ sSet"
```

6.4 The Simplicial Model

In this section we show the first definitions and results about representability of fibrations, presented in [96]. In that paper the notions of *regular cardinal* and *α-small morphisms* are also considered, in order to get that some collections of fibrations are sets and not *proper* classes. But in our *typed* environment these cautions are unnecessary.

Proposition 1. *Given two well-ordered sets, there is at most one isomorphism between them. Given two well-ordered morphisms over a common base, there is at most one isomorphism between them.*

Definition 37. *Given a simplicial set X we define $\mathbf{W}(X)$ to be the set of isomorphism classes of well-ordered morphisms $f : Y \to X$. Given a morphism $t : X' \to X$ we define $\mathbf{W}(t) : \mathbf{W}(X) \to \mathbf{W}(X')$ by $\mathbf{W}(t) = t^*$ (the pullback functor). This provides a functor $\mathbf{W} : osSet^{op} \to Set$.*

This definition is the first milestone of the simplicial model:

Definition 38. *Define the simplicial set W by*

$$W := \mathbf{W} \circ y^{op} : \Delta^{op} \to Set$$

where y denotes the Yoneda embedding $y : \Delta \to sSet$.

6.5 Formalising the Simplicial Model

Now it is time to prove in Isabelle/HOL Proposition 1. The result consists of two statements. The first one can be proven thanks to the existing properties presented in the library:

```
lemma well_order_sets_iso_unique:
assumes "well_order_on A r" and "well_order_on B r'"
and "iso r r' f" and "iso r r' g " and "a ∈ A"
shows "f a = g a"
by (metis assms embed_embedS_iso embed_unique well_order_on_Field)
```

For the second one, we have proven something even more general. First of all, we have defined what we have called *weak isomorphism between morphisms*:

```
definition "iso_between_morphism_weak f1 f2 g
  = (Y1_X.well_ordered_morphism f1
    ∧ Y2_X.well_ordered_morphism f2 (*Both f1 and f2 are a well ordered
morphism*)
    ∧ Y1_Y2.morphism g (*Morphism between simplicial set*)
    ∧ (∀n::nat. ∀y ∈ Y1 n. (fst(f2) n) (g n y) = (fst(f1) n) y)
(*Commute*)
    ∧ (∀n::nat. ∀x ∈ X n. iso (snd f1 n x) (snd f2 n x) (g n))
(*Isomorphism between fibers that respects the well-order on them*)
    )"
```

The main difference between the above definition and the definition of isomorphism of well-ordered morphisms, which is presented in Section 6.3 and called there *iso_between_morphism*, is that here we have not required *g* to be an isomorphism between simplicial sets, but just a morphism. Of course, *iso_between_morphism* implies the weak version:

```
lemma iso_imp_iso_weak:
  assumes "iso_between_morphism f1 f2 g"
  shows "iso_between_morphism_weak f1 f2 g"
```

Now, we can prove the second part of Proposition 1 in a more general way, using the definition presented above:

```
lemma iso_between_morphism_weak_unique:
  assumes g: "iso_between_morphism_weak (f1,ord1) (f2,ord2) g"
  and f: "iso_between_morphism_weak (f1,ord1) (f2,ord2) f"
  shows "g = f"
```

Finally, the desired result is obtained from the previous lemmas as a corollary:

```
corollary iso_between_morphism_unique:
  assumes g: "iso_between_morphism (f1,ord1) (f2,ord2) g"
  and f: "iso_between_morphism (f1,ord1) (f2,ord2) f"
  shows "g = f"
```

Up to now, we have proven [96, Proposition 3]. Now, we want to implement

and prove the properties shown in Definition 37.

We have to start by defining $\mathbf{W}(X)$ as the set of isomorphism classes of well-ordered morphisms $f : Y \to X$. We will do it in three steps:

1. Let X be a simplicial set, we define $W'(X)$ as the set of well-ordered morphisms $f : Y \to X$.
2. We define the following equivalence relation rel: two well-ordered morphisms $f_1 : Y_1 \to X$ and $f_2 : Y_2 \to X$ are related if there exists an isomorphism of well-ordered morphisms between them.
3. $\mathbf{W}(X)$ will be the quotient set $W'(X)/rel$.

When we define $W'(X)$ in Isabelle, we will have the same problem as when we worked with the *Set*-category: the elements of the set must be of the same type. Mathematically $f : Y_1 \to X$ and $g : Y_2 \to X$ where Y_1`::'a`, Y_2`::'b`, X`::'c` would belong to the same set $W'(X)$, but in Isabelle/HOL that is not possible, since once we say that $f : Y_1 \to X \in W'(X)$, then each element of that set $W'(X)$ must be of type `'a ⇒ 'c`. [1]

We have implemented $W'(X)$ and the equivalence relation in Isabelle/HOL as follows:

```
definition "W' = {((f,ord),(Y,∂y,sy)). class.simplicial_set Y ∂y sy
  ∧ two_simplicial_sets.well_ordered_morphism Y ∂y sy X ∂ s (f,ord)}"

definition "rel = {(((f1,ord1),(Y1,∂y1,sy1)),((f2,ord2),(Y2,∂y2,sy2))).
  class.simplicial_set Y1 ∂y1 sy1 ∧
  class.simplicial_set Y2 ∂y2 sy2 ∧ (∃g.
  three_simplicial_sets.iso_between_morphism X ∂ s Y1 ∂y1 sy1 Y2 ∂y2 sy2
(f1,ord1) (f2,ord2) g)}"
```

Both definitions have been done inside a **locale** where a simplicial set X is fixed. This is the reason why the simplicial set X is not an argument for W. However, when working with W involving different simplicial sets or just outside the **locale** where W is defined, we will have to make use of interpretations. For instance, outside such a **locale** we will have to write `X.W` and `Y.W` in Isabelle/HOL to refer to $W(X)$ and $W(Y)$ respectively, once we have proven `X` and `Y` to be simplicial sets. Same occurs with rel. In addition, let us remark that rel is just a set that consists of related pairs. Here, the simplicial sets Y_1 and Y_2 do have to be of the same type.

Now we demonstrate that `rel` is an equivalence relation (symmetric, reflexive, transitive). Again, we prove the properties as general as possible. For example, the following statement is the transitivity property of the relation rel:

```
lemma trans_extended:
assumes f: "X_Y1_Y2.iso_between_morphism (f1,ord1) (f2,ord2) f"
and g: "X_Y2_Y3.iso_between_morphism (f2,ord2) (f3,ord3) g"
shows "∃h. X_Y1_Y3.iso_between_morphism (f1,ord1) (f3,ord3) h"
```

[1] We have simplified the notation to make more understandable the situation. Y_1 cannot be just of type $'a$ because of the definition of simplicial set. The real type of a simplicial set is more cumbersome to manage: Y_1`::(nat ⇒ 'a set) × (nat ⇒ nat ⇒ 'a ⇒ 'a) × (nat ⇒ nat ⇒ 'a ⇒ 'a))`.

In the previous statement, the types of the simplicial sets are different (Y_1`::'a`, Y_2`::'b`, and Y_3`::'c`), but the following corollary fixes the simplicial sets to be of the same type `'a` since we have "fixed" the set `rel`:

```
lemma trans_rel: "trans rel"
```

Same occurs with the symmetry: let `f` be of type `'a ⇒ 'b` and `g` of type `'c ⇒ 'd` (again we have simplified the notation), and `(f,g) ∈ rel`. If `rel` is symmetric, then `(g,f) ∈ rel` and since `rel` is a set and both `(f,g) ∈ rel` and `(g,f) ∈ rel` then the types of `f` and `g` must be equal. The reflexive property always requires the simplicial sets to be of the same type. Now, we can prove the relation to be an equivalence relation over $W(X)$:

```
lemma equiv_W'_rel:
  shows "equiv W' rel"
```

Finally, we can define the quotient set:

```
definition "W = W' // rel"
```

Up to now, we have shown how we have carried out the formalisation of the first line of Definition 37. We have to define **W** for morphisms, that is, the pullback functor. Let $t : X' \to X$ be a morphism. We have defined t^* in Isabelle/HOL as follows (we have called it `t'` since t^* is not an allowed notation in Isabelle):

```
definition "t_aux A =
  (let ((f,ord), (Y, ∂y, sy)) = A
    in (three_simplicial_sets_t_wo_f.pullback X' Y t f ord,
        (three_simplicial_sets_t_wo_f.pullback_set X' Y t f,
         three_simplicial_sets_t_wo_f.pullback_∂ ∂x' ∂y,
         three_simplicial_sets_t_wo_f.pullback_s sx' sy
        ))
  )"

definition "t' S = {b. ∃a∈S. (b, t_aux a) ∈ simplicial_set.rel X' ∂x'
sx'}"
```

Now, we have to prove that t^* respects the equivalence relation. That is, if

$$((f_1, ord_1), (Y_1, \partial_{y_1}, s_{y_1})) \sim ((f_2, ord_2), (Y_2, \partial_{y_2}, s_{y_2}))$$

then

$$t^*((f_1, ord_1), (Y_1, \partial_{y_1}, s_{y_1})) \sim t^*((f_2, ord_2), (Y_2, \partial_{y_2}, s_{y_2}))$$

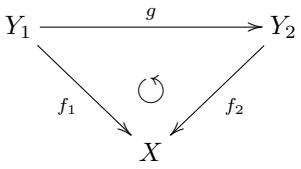
The statement of this result is presented in Isabelle/HOL as follows:

```
lemma pullback_preserves_rel:
  "((X_X'_Y1_t_f1.pullback,(X_X'_Y1_t_f1.pullback_set,
X_X'_Y1_t_f1.pullback_∂,X_X'_Y1_t_f1.pullback_s))
  ,(X_X'_Y2_t_f2.pullback, (X_X'_Y2_t_f2.pullback_set,
X_X'_Y2_t_f2.pullback_∂,X_X'_Y2_t_f2.pullback_s))) ∈ X'.rel"
```

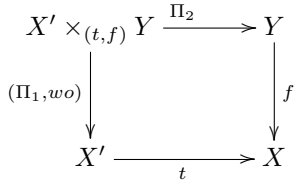
We must show that $\mathbf{W} : osSet^{op} \to Set$ is a functor in Isabelle/HOL. Among other things, we have to prove that $\mathbf{W}(t) : \mathbf{W}(X) \to \mathbf{W}(X')$ is an arrow in *a Set*-category implemented in Isabelle/HOL.

The mathematical proof of $\mathbf{W}(t)$ being an arrow from $\mathbf{W}(X)$ to $\mathbf{W}(X')$ involves the use of the pullback on morphisms. In Isabelle/HOL, this property can be stated and proven, but only if $\mathbf{W}(X)$ and $\mathbf{W}(X')$ have different types, which is a problem since $\mathbf{W}(t)$ must be an arrow in the *Set*-category. Let us explain it. Again we simplify the types to make more understandable the point.

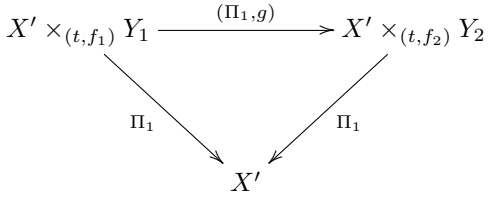
Let X and X' be two simplicial sets of types `'a` and `'b` respectively. Let `'c ⇒ 'a` be the type of the elements that belong to the set $\mathbf{W}(X)$. In order to show explicitly the types of the sets, we denote it as $\mathbf{W}_{'c \Rightarrow 'a}(X)$. Analogously, we will denote $rel_{'c \Rightarrow 'a}(X)$ the corresponding equivalence relation. Let $t : X' \to X$ be a morphism. Let $f_1 : Y_1 \to X$ and $f_2 : Y_2 \to X$ (Y_1 and Y_2 are of type `'c`) be two well-ordered morphisms that are related $(f_1, f_2) \in rel_{'c \Rightarrow 'a}(X)$. This means that there exists an isomorphism $g : Y_1 \to Y_2$ between the well-ordered morphisms f_1 and f_2.



Let $f : Y \to X$ be a well-ordered morphism. We know that $\mathbf{W}(t)(f) = t^*(f)$ returns the well-ordered morphism $X' \times_{(t,f)} Y \xrightarrow{(\Pi_1, wo)} X'$ that makes the following diagram commutative:



Thus, $t^*(f_1)$ and $t^*(f_2)$ are related in the following way:



That is, there exists an isomorphism (Π_1, g) of well-ordered morphisms. The simplicial sets $X' \times_{(t,f_1)} Y_1$ and $X' \times_{(t,f_2)} Y_2$ are of type `'b × 'c`:

$$X' \times_{(t,f_1)} Y_1 \xrightarrow{(\Pi_1, g)} X' \times_{(t,f_2)} Y_2$$

This implies that $(t^*(f_1), t^*(f_2)) \in rel_{('b \times 'c) \Rightarrow 'b}(X')$. Thus, we can prove $\mathbf{W}(t) : \mathbf{W}_{'c \Rightarrow 'a}(X) \to \mathbf{W}_{('b \times 'c) \Rightarrow 'b}(X')$. But as we see, $\mathbf{W}_{'c \Rightarrow 'a}(X)$ and $\mathbf{W}_{('b \times 'c) \Rightarrow 'b}(X')$ must be of different types.

As we have said, **W** must be a functor in Isabelle/HOL and that demands $\mathbf{W}(t)$ to be an arrow in the *Set*-category. This would mean that $\mathbf{W}_{'c \Rightarrow 'a}(X)$ and $\mathbf{W}_{('b \times 'c) \Rightarrow 'b}(X')$ must be of unifiable types, which is not possible. Thus, we cannot prove $\mathbf{W} : osSet^{op} \to Set$ to be a functor in Isabelle/HOL.

The next step would be to define in Isabelle/HOL the functor W as a composition of three functors. Let us note that we would have to include the function $from\_sSet$ because y^{op} is a functor between Δ^{op} and $sSet^{op}$, but **W** is a functor between $osSet^{op}$ and Set (in the article [96] $sSet$ is used to refer both $sSet$ and $osSet$ since the categories are isomorphic). Thus, the composition would be the following one:

$$W := \mathbf{W} \circ from\_sSet^{op} \circ y^{op} : \Delta^{op} \to Set$$

where y denotes the Yoneda embedding $y : \Delta \to sSet$.

However, we cannot do it since we have not been able to prove $\mathbf{W} : osSet^{op} \to Set$ as a functor in Isabelle/HOL. More concretely, we cannot prove that $\mathbf{W}(t)$ is an arrow according to the definition of *Set*-category which we are using in Isabelle/HOL.

6.5.1 Porting the Development to Isabelle/HOLZF

The previous section shows that the existing Isabelle/HOL library about Category Theory does not allow us to go further in our development. Unfortunately, to develop a new library about Category Theory in Isabelle/HOL is not a workaround since the mathematical concept of general *Set*-category seems difficult to formalise within Isabelle/HOL.

As it was presented in Section 6.1, there exist other logics which have been implemented on top of Isabelle, although they have barely been used.

One of them is HOLZF, for which there exists a library about Category Theory, which is explained thoroughly in [98]. This library includes the definition of category, locally small category, functors, natural transformations, and a proof of Yoneda's lemma. The *Set*-category is defined in such a library as follows:

```
definition
  SET' :: "(ZF, ZF) Category" where
  "SET' ≡ (|
      Category.Obj = {x . True} ,
      Category.Mor = {f . isZFfun f} ,
      Category.Dom = ZFfunDom ,
      Category.Cod = ZFfunCod ,
      Category.Id = λx. ZFfun x x (λx . x) ,
      Category.Comp = ZFfunComp
  |)"

definition "SET ≡ MakeCat SET'"
```

Let us note that an extensive use of HOLZF is made, since now the objects and arrows of the category of sets are both of type `ZF` which is the type that supports ZF set theory in HOL. In contrast with the existing implementation of the *Set*-category in Isabelle/HOL that we are using, this one models better the mathematical concept of *Set*-category: there exists just one and it is not

dependent on the types. Furthermore, a function $f : \{1, 2, 3\} \to \{True, False\}$ such that $f(1) = f(2) = True$ and $f(3) = False$ (as the one we presented in Section 6.3) does belong to this Isabelle/HOLZF implementation of the *Set*-category.

It seems to be a good approach to experiment with this library of Isabelle/HOLZF in order to know if the problem presented in Section 6.5 could be avoidable. Strongly reusing the infrastructure developed in Isabelle/HOL and thanks to the existing relation between HOL and HOLZF, we have successfully ported to Isabelle/HOLZF several of the results presented in previous sections. In order to do that, we have transformed almost each occurrence of the types `'a set` and `'a` to `ZF` and adapted the code to the characteristics of *ZF*, such as the use of the operators $|\subseteq|$ and $|\in|$ instead of $\subseteq$ and $\in$. This formalisation in Isabelle/HOLZF is presented in [49] and it includes the main definitions (simplicial sets, well-ordered morphisms, isomorphism between well-ordered morphisms). We have also fully ported Proposition 1, which is the first result of Voevodsky's simplicial model, and Theorem 15, whose statement is presented below:

```
sublocale simplicial_set_prod:
  simplicial_set "(λn. (Y n) |×| (X n)) "
                 "(λi n x. Opair (∂y i n (Fst x)) (∂x i n (Snd x)))"
                 "(λi n x. Opair (sy i n (Fst x)) (sx i n (Snd x)))"
proof
  ...
qed


sublocale Y1_times_Y2_tf: simplicial_set
          "(λn. Sep (Y1 n |×| Y2 n) (λx. t n (Fst x) = f n (Snd x)))"
          "(λi n x. Opair (∂y1 i n (Fst x)) (∂y2 i n (Snd x)))"
          "(λi n x. Opair (sy1 i n (Fst x)) (sy2 i n (Snd x)))"
proof
  ...
qed
```

The latter result is specially important: we have now proven that given two simplicial sets (Y, ∂_Y, s_Y) and (X, ∂_X, s_X) where the graded sets Y and X are of type `nat ⇒ ZF`, then the corresponding graded set $(Y \times X)$ is again of type `nat ⇒ ZF`, in contrast to our development in Isabelle/HOL, where Y `:: nat ⇒ 'a set`, X `:: nat ⇒ 'b set` and hence $(Y \times X)$ `:: nat ⇒ ('a × 'b) set`. This seems to mean that the $\mathbf{W}(t)$ could be defined and proved as an arrow of the Set category, avoiding the limitation presented in Section 6.5.

Let us also remark that when defining the simplicial set $(Y_1 \times_{(t,f)} Y_2, \partial_{Y_1 \times_{(t,f)} Y_2}, s_{Y_1 \times_{(t,f)} Y_2})$ in Isabelle/HOLZF we have had to make use of the separation operator `Sep`, which corresponds to the ZF *separation axiom*.

6.6 Conclusions

In this chapter, we report on an experiment aimed at exploring how much of Voevodsky's simplicial model [96] could be formalised in Isabelle/HOL. Once the experiment done the conclusion is: very few. In addition, once the main limitation is understood, one realises that big parts of the formalisation were unnecessary to enlighten the nature of the problem. Nevertheless, we consider that the Isabelle/HOL code written, which takes up *ca.* 5000 lines, is interesting and deserves to be disseminated, as it includes, in particular, a formalisation of simplicial sets for the first time (up to our knowledge) in Isabelle/HOL.

The way in which the obstruction to the formalisation occurred can be considered a consequence of the methodology we followed: keep us inside HOL and reuse code as much as possible. In particular, we collided with the definition of category in the library [121], and even more concretely with its definition of *Set*-category. Roughly speaking, the problem is that in a category of arrows, one morphism and a pullback of it have non-unifiable types, and then they cannot be considered objects of the same category in Isabelle/HOL.

From a conceptual perspective, the problem lies on the decision of defining the category of sets in Isabelle/HOL by means of the type constructor `set`, as it is clear that it does not reflect the essence of Zermelo-Fraenkel sets. But the issue of representing a more expressive category of sets in Isabelle/HOL seems far from trivial. The limitation of using the type constructor `set` was already remarked in [98], where a definition of the *Set*-category is introduced based on the Isabelle/HOLZF framework. In Isabelle/HOLZF [120], HOL is extended with Zermelo-Fraenkel axioms. Some preliminary experiments show that porting our development to Isabelle/HOLZF could avoid the obstruction encountered in Isabelle/HOL. In any case, to determine both if the found limitation could be overcome in Isabelle/HOL and if Voevodsky's model is representable in Isabelle/HOLZF is still to be elucidated.

Chapter 7

Conclusions and Future Work

7.1 Results

In a nutshell, this thesis covers the formalisation of theorems and algorithms in Linear Algebra together with their well-known applications. Roughly speaking, we have presented the formalisation of:

- **Theorems:** the Fundamental Theorem of Linear Algebra and relationships between linear maps and matrices.
- **Algorithms:** Gauss-Jordan, QR decomposition, echelon form, and Hermite normal form.

The way to tackle it has been as follows: we have formalised a general framework where we have achieved the execution of the matrix representation presented in the HOL Multivariate Analysis library (something that had not been explored before, either in Isabelle or in HOL Light). This framework eases the definition and formalisation of Linear Algebra algorithms using the matrix representation of the HOL Multivariate Analysis library. Algorithms can be executed within Isabelle. Besides, the infrastructure developed also allows us to refine algorithms to a more efficient matrix representation based on immutable arrays and to generate from them code to functional languages (SML and Haskell). Furthermore, the ties between matrix algorithmics and linear maps are also formalised (this link is not usually present in Computer Algebra systems). This infrastructure is not exclusive for our development and it can be reused for other Linear Algebra developments [8].

We have also serialised some of the Isabelle datatypes (reals, rationals, ...) to the corresponding native structures in the target functional languages in order to improve the performance of the generated code. We present some experiments which show that the use of immutable arrays do not pose a limitation compared to an imperative implementation. Additionally, many theorems and structures presented in the HOL Multivariate Analysis library have been generalised from concrete types (real matrices, real vector spaces, ...) to more abstract structures (matrices over fields, vector spaces over an arbitrary field, ...). By means

of the previous framework and generalisations, we have formalised four well-known Linear Algebra algorithms: the Gauss-Jordan algorithm, QR decomposition, echelon form and Hermite normal form. The first two ones involve the use of matrices over fields, the latter two ones involve matrices over Bézout domains. As far as we know, most of them had never been formalised in Isabelle or even in any theorem prover before. The straightforward applications of the algorithms have been formalised as well: computation of ranks of matrices, inverses, characteristic polynomials, bases of the fundamental subspaces, orthogonal bases, ...

In fact, thanks to these algorithms, two of the "central problems of Linear Algebra" (a terminology used by Strang in [142]) can be formally computed. We have explored both the precision (when working with reals serialised as floats in the target languages) and performance of the algorithms. Although the obtained performance is not comparable to unverified Computer Algebra systems, empirical experiments showed that the exported verified code can be used in real-world applications. For instance, our exported code from the formalisation of the Gauss-Jordan algorithm can be used to compute the homology of matrices representing digital images (a process that is useful in Neurobiology, where 2500×2500 $\mathbb{Z}_2$ matrices are involved).

To sum up, we have developed a set of results and formalised programs in Linear Algebra that correspond to various lessons of an undergraduate text in Mathematics. For instance, many theorems and algorithms presented in this thesis correspond to the first four chapters in the textbook by Strang [142], which sums up about 250 pages. Computer Algebra systems are usually error prone and black boxes which employ tricky performance optimisations. Nowadays, code obtained from formalisations cannot compete against them in terms of performance, but it pays off in reliability and confidence. This work tries to reduce the existing gap between software formalisation and working software, since we have tried to get a reasonable performance as well.

Apart from theorems and algorithms of Linear Algebra, we present a more tentative chapter where a formalisation of simplicial sets for the first time (up to our knowledge) in Isabelle/HOL is explained, together with an experiment for its application to Voevodsky's simplicial model for Homotopy Type Theory. In this experiment, we collided with the definition of *Set*-category in Isabelle/HOL, which prevented us to go further in the development. Nevertheless, we have also presented the possibilities to port it to Isabelle/HOLZF which seems that could avoid the obstruction encountered in Isabelle/HOL.

The formalisations are part of the Archive of Formal proofs so that other users could reuse them (in fact, they have already been useful for other users [106, 148]). The total number of lines of the formalisations presented here is 36075 (see Table A.8). Tables A.1 to A.7 show the files and the number of lines of each development.

7.2 Future Work

Although we have discussed the possible future work for each algorithm in their corresponding chapters, here we suggest general projects that are interesting to work on in the future.

- **Formalisation of other Linear Algebra algorithms:** Linear Algebra is a big topic and there exist some other algorithms that could be formalised following the same approach. For example, the formalisation of the related Smith normal form and the Singular Value Decomposition would be of wide interest because of their applications. In addition, it would be desirable to explore other more efficient representations, such as sparse matrices. To refine the algorithms presented in this thesis to more efficient versions of them would also be useful.

- **Exploration of the ties with other Linear Algebra developments:** During the last year while developing this thesis, another formalisation about Linear Algebra in Isabelle/HOL was published. It was developed by Thieman and Yamada and it includes, among other things, many results about Jordan normal forms, eigenvalues and a formalisation of the Gauss-Jordan elimination. They reused some parts of our work, such as the serialisation to immutable arrays in Haskell. By means of the lifting and transfer package [91] it would be nice to explore if it is possible to transfer their results (both theorems and algorithms) to the library we based our work on (the HOL Multivariate Analysis library). Furthermore, it would be interesting to study the work by Kunčar and Popescu [101] in order to connect both representations bidirectionally.

- **Algebraic numbers:** To study the possibilities of using of algebraic numbers would be desirable. In the last few months, two developments about this topic have been published: the first one by Thiemann and Akihisa [147] and the second one by Li and Paulson [106]. Both of them make use of *oracles* and certifiers to validate the results. These works could be useful to explore the computation of eigenvalues and eigenvectors, and in general for developments which require exact computations in structures different from $\mathbb{Q}[\sqrt{b}]$.

- **Homotopy Type Theory:** the simplicial model for homotopy type theory developed by Voevodsky has become in an influential and important topic, not only for the interactive theorem proving community but for many mathematicians. In Chapter 6 we have presented some results of simplicial sets and an experiment of its application to Voevodsky's simplicial model. Unfortunately, we got stuck on going deeper on the formalisation due to the definition of the Set category in Isabelle/HOL, but some experiments showed that such a limitation could be avoidable in Isabelle/HOLZF. To explore if Isabelle/HOLZF is actually able to formalise such a simplicial model would be of interest by far.

Appendix A

Detailed List of Files and Benchmarks

A.1 Detailed List of Files

In this section, lists with length and file names presented in each development are given. The total amount of lines of this thesis is 36075. The exact number of lines can vary along the time, since AFP is constantly being updated. These lines just correspond to the developments presented in the AFP version for Isabelle 2016. In addition, there are some other parts of our work that were part of the AFP, but now they have already been included as part of the Isabelle library by the Isabelle developers.

Rank-Nullity Theorem	
File name	**Number of lines**
Dim_Formula	334
Dual_Order	73
Fundamental_Subspaces	193
Generalisations	2549
Miscellaneous	537
Mod_Type	557
Total Isabelle code	4243

Table A.1: List of files in the formalisation of the Rank-Nullity theorem [52].

Gauss-Jordan algorithm and its applications	
File name	**Number of lines**
Bases_Of_Fundamental_Subspaces	214
Bases_Of_Fundamental_Subspaces_IArrays	164
Code_Bit	62
Code_Generation_IArrays	115
Code_Generation_IArrays_Haskell	59
Code_Generation_IArrays_SML	127
Code_Matrix	84
Code_Rational	139
Code_Real_Approx_By_Float_Haskell	66
Code_Set	42
Determinants2	502
Determinants_IArrays	246
Elementary_Operations	1030
Examples_Gauss_Jordan_Abstract	164
Examples_Gauss_Jordan_IArrays	211
Gauss_Jordan	2433
Gauss_Jordan_IArrays	402
Gauss_Jordan_PA	457
Gauss_Jordan_PA_IArrays	319
IArray_Addenda	87
IArray_Haskell	118
Inverse	331
Inverse_IArrays	70
Linear_Maps	1127
Matrix_To_IArray	414
Rank	46
Rref	413
System_Of_Equations	640
System_Of_Equations_IArrays	390
Total Isabelle code	10472

Table A.2: List of files in the formalisation of the Gauss-Jordan algorithm [54].

QR **decomposition**	
File name	**Number of lines**
Examples_QR_Abstract_Float	49
Examples_QR_Abstract_Symbolic	99
Examples_QR_IArrays_Float	87
Examples_QR_IArrays_Symbolic	185
Generalizations2	624
Gram_Schmidt	915
Gram_Schmidt_IArrays	465
IArray_Addenda_QR	192
Least_Squares_Approximation	458
Matrix_To_IArray_QR	422
Miscellaneous_QR	445
Projections	197
QR_Decomposition	759
QR_Decomposition_IArrays	111
QR_Efficient	698
Total Isabelle code	5706

Table A.3: List of files in the formalisation of the QR decomposition [51].

Echelon form	
File name	**Number of lines**
Cayley_Hamilton_Compatible [a]	115
Code_Cayley_Hamilton	108
Code_Cayley_Hamilton_IArrays	85
Echelon_Form	2918
Echelon_Form_Det	306
Echelon_Form_Det_IArrays	368
Echelon_Form_IArrays	386
Echelon_Form_Inverse	56
Echelon_Form_Inverse_IArrays	30
Euclidean_Algorithm [b]	2067
Euclidean_Algorithm_Extension	245
Examples_Echelon_Form_Abstract	52
Examples_Echelon_Form_IArrays	150
Rings2	1046
Total Isabelle code	7932

Table A.4: List of files in the formalisation of the Echelon form [50].

[a] Developed by Hölzl
[b] Based on the Eberl's work with slight modifications

Hermite normal form	
File name	**Number of lines**
Hermite	2152
Hermite_IArrays	155
Total Isabelle code	2307

Table A.5: List of files in the formalisation of the Hermite normal form [57].

Experiment about the simplicial model in Isabelle/HOL	
File name	**Number of lines**
Yoneda_Embedding	1368
Simplicial_Model_HOL	2396
OsSet	1129
Total Isabelle code	4893

Table A.6: List of files in the formalisation of the experiment about the simplicial model in Isabelle/HOL [48].

Experiment about the simplicial model Isabelle/HOLZF	
File name	**Number of lines**
Simplicial_Model_HOLZF	522
Total Isabelle code	522

Table A.7: List of files in the formalisation of the experiment about the simplicial model in Isabelle/HOLZF [49].

Total number of lines of the development	
Development name	**Number of lines**
Rank-Nullity Theorem [52]	4243
Gauss-Jordan algorithm and its applications [54]	10472
QR decomposition [51]	5706
Echelon form [50]	7932
Hermite normal form [57]	2307
Experiment about the simplicial model in Isabelle/HOL [48]	4893
Experiment about the simplicial model in Isabelle/HOLZF [49]	522
Total Isabelle code	36075

Table A.8: Total number of lines of the development.

A.2 Benchmarks

We present here the performance tests that we have carried out for our programs obtained from the formalisation of the Gauss-Jordan algorithm and the QR decomposition [51] (both of them were introduced in Chapter 4). We present the benchmarks for the refined versions to immutable arrays and making use of the serialisations explained through the thesis. The same randomly generated

matrices have been used across the different systems and they can be obtained from [56]. Let us note that the underlying field of the matrix coefficients notably affects the performance and can even affect the complexity bounds. The times presented throughout the tables are expressed in seconds. The benchmarks have been carried out in a laptop with an Intel Core i5-3360M processor, 4 GB of RAM, Poly/ML 5.6, Ubuntu 14.04, GHCi 7.6.3, and Isabelle 2016.

Size (n)	**Poly/ML**	**Haskell**
100	0.04	0.36
200	0.25	2.25
300	0.85	9.09
400	2.01	17.17
500	3.90	32.56
600	6.16	56.39
800	15.96	131.73
1 000	32.08	255.84
1 200	62.33	453.57
1 400	97.16	715.87
1 600	139.70	1097.41
1 800	203.10	1609.72
2 000	284.28	2295.30

Table A.9: Time to compute the *rref* of randomly generated $\mathbb{Z}_2$ matrices.

Size (n)	**Poly/ML**	**Haskell**
10	0.01	0.01
20	0.02	0.03
30	0.07	0.09
40	0.21	0.24
50	0.57	0.57
60	1.16	1.09
70	2.20	2.12
80	3.77	3.53
90	6.22	5.75
100	9.75	9.03

Table A.10: Time to compute the *rref* of randomly generated $\mathbb{Q}$ matrices.

Size (n)	Poly/ML	Haskell
100	0.03	0.38
200	0.25	2.62
300	0.81	8.47
400	1.85	19.51
500	3.51	37.13
600	6.03	64.13
700	9.57	100.59
800	13.99	148.20

Table A.11: Time to compute the *rref* of randomly generated $\mathbb{R}$ matrices.

Size (n)	Poly/ML
100	0.748
120	1.558
140	2.779
160	4.621
180	7.251
200	10.869
220	18.941
240	22.188
260	30.198
280	42.100
300	84.310
320	94.223
340	97.360
360	123.266
380	157.426
400	183.754

Table A.12: Time to compute the QR decomposition of Hilbert matrices over $\mathbb{R}$.

Bibliography

[1] http://www.math4all.in/public_html/linearalgebra/chapter3.4.html.

[2] http://homotopytypetheory.org/.

[3] Archive of Formal Proofs. http://afp.sourceforge.net/.

[4] Immutable Arrays in Haskell. https://downloads.haskell.org/~ghc/6.12.1/docs/html/libraries/array-0.3.0.0/Data-Array-IArray.html.

[5] Matlab documentation. Definition of Hermite Normal Form. http://es.mathworks.com/help/symbolic/hermiteform.html#butzrp_-5.

[6] The Isabelle website. https://isabelle.in.tum.de/.

[7] ARIANE 5 Flight 501 Failure. Report by the Inquiry Board, 1996. https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[8] S. Adelsberger, S. Hetzl, and F. Pollak. The Cayley-Hamilton Theorem. *Archive of Formal Proofs*, 2014. http://afp.sf.net/entries/Cayley_Hamilton.shtml, Formal proof development.

[9] J. Aransay and J. Divasón. Formalizing an abstract algebra textbook in Isabelle/HOL. In Juan Rafael Sendra Pons and Carlos Villarino Cabellos, editors, *Proceedings of the XIII Spanish Meeting on Computer Algebra and Applications (EACA 2012)*, pages 47 – 50, 2012.

[10] J. Aransay and J. Divasón. Formalization and execution of Linear Algebra: from theorems to algorithms. In G. Gupta and R. Peña, editor, *PreProceedings of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013*, pages 49 – 66, 2013.

[11] J. Aransay and J. Divasón. Performance Analysis of a Verified Linear Algebra Program in SML. In L. Fredlund and L. M. Castro, editors, *V Taller de Programación Funcional: TPF 2013*, pages 28 – 35, 2013.

[12] J. Aransay and J. Divasón. Formalization and execution of Linear Algebra: from theorems to algorithms. In G. Gupta and R. Peña, editors, *PostProceedings (Revised Selected Papers) of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013*, volume 8901 of *LNCS*, pages 01 – 19. Springer, 2014.

[13] J. Aransay and J. Divasón. Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. *Journal of Functional Programming*, 25, 2015. doi:10.1017/S0956796815000155.

[14] J. Aransay and J. Divasón. Generalizing a Mathematical Analysis Library in Isabelle/HOL. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 9058 of *LNCS*, pages 415–421. Springer, 2015.

[15] J. Aransay and J. Divasón. Proof Pearl - A formalisation in HOL of the Fundamental Theorem of Linear Algebra and its application to the solution of the least squares problem. Draft paper, 2015. `https://www.unirioja.es/cu/jodivaso/publications/2015/Least_Squares_2015.pdf`.

[16] J. Aransay and J. Divasón. Verified Computer Linear Algebra. In Accepted for presentation at *the XV Spanish Meeting on Computer Algebra and Applications (EACA 2016)*, 2016.

[17] J. Aransay and J. Divasón. Formalisation of the Computation of the Echelon Form of a matrix in Isabelle/HOL. Accepted for publication in *Formal Aspects of Computing*, 2016. `http://www.unirioja.es/cu/jodivaso/publications/2016/echelon_FAC_draft.pdf`.

[18] J. Aransay, J. Divasón, and J. Rubio. Formalising in Isabelle/HOL a simplicial model for Homotopy Type Theory: a naive approach. 2015. `http://www.unirioja.es/cu/jodivaso/publications/2016/hott_draft_2016.pdf`, Draft paper.

[19] J. Aransay-Azofra, J. Divasón, J. Heras, L. Lambán, M. V. Pascual, Á. L. Rubio, and J. Rubio. Obtaining an ACL2 specification from an Isabelle/HOL theory. In Gonzalo A. Aranda-Corral, Jacques Calmet, and Francisco J. Martín-Mateos, editors, *Artificial Intelligence and Symbolic Computation - 12th International Conference, AISC 2014, Seville, Spain, December 11-13, 2014. Proceedings*, volume 8884 of *Lecture Notes in Computer Science*, pages 49–63, 2014.

[20] J. Avigad and J. Harrison. Formally Verified Mathematics. *Communications of the ACM*, 57(4):66 – 75, 2014.

[21] J. Avigad, J. Hölzl, and L. Serafin. A formally verified proof of the Central Limit Theorem. *CoRR*, abs/1405.7012, 2014.

[22] S. Axler. *Linear Algebra Done Right*. Undergraduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1997.

[23] C. Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[24] C. Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.

[25] K.J. Bathe. *Computational Fluid and Solid Mechanics*. Elsevier Science, 2003.

[26] BBC news. Airbus A400M plane crash linked to software fault. `http://www.bbc.com/news/technology-32810273`.

[27] L. W. Beineke and R. J. Wilson. *Topics in Algebraic Graph Theory.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2004.

[28] M. Bezem, T. Coquand, and S. Huber. A Model of Type Theory in Cubical Sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, 2014.

[29] A. Björck. *Numerical methods for least squares problems.* SIAM, 1996.

[30] J. Blanchette, M. Haslbeck, D. Matichuk, and T. Nipkow. Mining the Archive of Formal Proofs. In M. Kerber, editor, *Conference on Intelligent Computer Mathematics (CICM 2015)*, volume 9150, pages 3–17, 2015. Invited paper.

[31] A. Borodin, J. von zur Gathen, and J. E. Hopcroft. Fast Parallel Matrix and GCD Computations. *Information and Control*, 52(3):241–256, 1982.

[32] G. H. Bradley. Algorithms for Hermite and Smith normal matrices and linear diophantine equations. *Mathematics of Computation*, 25(116):897–907, 1971.

[33] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative Functional Programming with Isabelle/HOL. In C. Muñoz O. Mohamed and S. Tahar, editors, *TPHOLs '08: Proceedings of the 21th International Conference on Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2008.

[34] G. Cano, C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Formalized Linear Algebra over Elementary Divisor Rings in Coq. Technical report, 2014.

[35] G. Cantor. Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *Crelle's Journal für Mathematik*, 77:258–263, 1874.

[36] S.T. Chapman and S. Glaz. *Non-Noetherian Commutative Ring Theory.* Mathematics and Its Applications. Springer US, 2013.

[37] D. Child. *The Essentials of Factor Analysis.* Bloomsbury Academic, 2006.

[38] A. Church. A Formulation of the Simple Theory of Types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

[39] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for Free! In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[40] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[41] T. Coquand, A. Mörtberg, and V. Siles. A formal proof of Sasaki-Murao algorithm. *J. Formalized Reasoning*, 5(1):27–36, 2012.

[42] G. Dahlquist and A. Björck. *Numerical methods in scientific computing*. SIAM, 2008.

[43] G. Dahlquist and J. Dongarra. *Numerical Methods in Computer Science*, volume I. SIAM, 2008.

[44] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in COQ. In L. Beringer and A. Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406, pages 83–98. Springer, 2012.

[45] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2015. Version 8.5. `http://coq.inria.fr`.

[46] J. Divasón. Benchmarks between Vec, IArray and Lists in Isabelle/HOL, 2015. `http://www.unirioja.es/cu/jodivaso/Isabelle/Thesis/Vec_vs_IArray_vs_List/Vec_vs_IArray_vs_List.zip`.

[47] J. Divasón. Serialisation to Haskell's UArrays, 2015. `http://www.unirioja.es/cu/jodivaso/Isabelle/Thesis/UArrays/UArray_Haskell.thy`.

[48] J. Divasón. Simplicial model in Isabelle/HOL. Formal Proof Development, 2015. `https://www.unirioja.es/cu/jodivaso/Isabelle/Simplicial_Model_HOL/`.

[49] J. Divasón. Simplicial model in Isabelle/HOLZF. Formal Proof Development, 2015. `http://www.unirioja.es/cu/jodivaso/Isabelle/Simplicial_Model_HOLZF/`.

[50] J. Divasón and J. Aransay. Echelon Form. *Archive of Formal Proofs*, 2015. `http://afp.sf.net/entries/Echelon_Form.shtml`, Formal proof development.

[51] J. Divasón and J. Aransay. QR Decomposition. *Archive of Formal Proofs*, 2015. `http://afp.sf.net/entries/QR_Decomposition.shtml`, Formal proof development.

[52] J. Divasón and J. Aransay. Rank-nullity theorem in linear algebra. *Archive of Formal Proofs*, January 2013. `http://afp.sf.net/entries/Rank_Nullity_Theorem.shtml`, Formal proof development.

[53] J. Divasón and J. Aransay. Rank-nullity theorem in linear algebra. *Archive of Formal Proofs*, January 2013. `http://afp.sourceforge.net/release/afp-Rank_Nullity_Theorem-2013-02-16.tar.gz`, Formal proof development. Old Version with no generalisations.

[54] J. Divasón and J. Aransay. Gauss-Jordan Algorithm and Its Applications. *Archive of Formal Proofs*, September 2014. `http://afp.sf.net/entries/Gauss_Jordan.shtml`, Formal proof development.

[55] J. Divasón and J. Aransay. Gauss-Jordan elimination in Isabelle/HOL. `http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2/`, 2014. Version without generalisations.

[56] J. Divasón and J. Aransay. Gauss-Jordan elimination in Isabelle/HOL. `http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2-Generalized/`, 2014.

[57] J. Divasón and J. Aransay. Hermite normal form. *Archive of Formal Proofs*, July 2015. `http://afp.sf.net/entries/Hermite.shtml`, Formal proof development.

[58] J. Dongarra and F. Sullivan. The top 10 algorithms. *Computer Science Engineering*, 2(1):22–23, 2000.

[59] A. J. Durán, M. Pérez, and J. L. Varona. Misfortunes of a mathematicians' trio using Computer Algebra Systems: Can we trust? *Notices of the AMS*, 61(10):1249 – 1252, 2014.

[60] M. Dénès. *Étude formelle d'algorithmes efficaces en algèbre linéaire*. PhD thesis, INRIA Sophia Antipolis, 2013.

[61] M. Eberl. A decision procedure for univariate real polynomials in Isabelle/HOL. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 75–83. ACM, 2015.

[62] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. G. Smaus. A Fully Verified Executable LTL Model Checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification: CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 463 – 478. Springer, 2013.

[63] J. G. F. Francis. The QR transformation, a unitary analogue to the LR transformation. I, II. *The Computer Journal*, 4:265–271, 332 – 345, 1961.

[64] L. Fuchs and L. Salce. *Modules Over Non-Noetherian Domains*. Mathematical surveys and monographs. American Mathematical Society, 2001.

[65] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Computer science and scientific computing. Elsevier Science, 2013.

[66] R. Garfinkel and G.L. Nemhauser. *Integer programming*. Series in decision and control. Wiley, 1972.

[67] E. Gasner and J. H. Reppy (eds.). The Standard ML Basis Library. `http://www.standardml.org/Basis/`.

[68] M. S. Gockenbach. *Finite-dimensional Linear Algebra*. CRC Press, 2010.

[69] K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, 1992. Translation B. Meltzer.

[70] G. Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[71] G. Gonthier. Point-free, set-free concrete linear algebra. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving: ITP 2011*, volume 6898 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2011.

[72] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In S. Blanzy, C. Paulin-Mohring, and D.Pichardie, editors, *Interactive Theorem Proving: ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 163 – 179. Springer, 2013.

[73] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[74] M. J. C. Gordon. Set Theory, Higher Order Logic or Both? In *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, pages 191–201, 1996.

[75] NASA Langley Formal Methods Groups. What is Formal Methods? `http://shemesh.larc.nasa.gov/fm/fm-what.html`.

[76] J. L. Hafner and K. S. McCurley. A rigorous subexponential algorithm for computation of class groups. *Journal of the American Mathematical Society*, 2(4):837–850, 1989.

[77] F. Haftmann. Haskell-style type classes with Isabelle/Isar. Tutorial documentation, 2015. `https://isabelle.in.tum.de/doc/classes.pdf`.

[78] F. Haftmann. Code generation from Isabelle/HOL theories. `https://isabelle.in.tum.de/dist/Isabelle2016/doc/codegen.pdf`, 2016.

[79] F. Haftmann, A. Krauss, O. Kuncar, and T. Nipkow. Data Refinement in Isabelle/HOL. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving: ITP 2013*, volume 7998 of *Lecture Notes in Computer Science*, pages 100 – 115. Springer, 2013.

[80] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume and N. Kobayashi and G. Vidal, editor, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009 of *Lecture Notes in Computer Science*, pages 103 – 117. Springer, 2010.

[81] T. C. Hales et al. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[82] T. C. Hales and S. P. Ferguson. *The Kepler Conjecture. The Hales-Ferguson Proof.* Springer New York, 2011.

[83] P. R. Halmos. *Naive Set Theory.* Van Nostrand, 1960.

[84] J. Harrison. A HOL Theory of Euclidean Space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 114 – 129. Springer, 2005.

[85] J. Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50(2):173 – 190, 2013.

[86] The Haskell Programming Language. `http://www.haskell.org/`, 2014.

[87] J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza, and V. Siles. Towards a certified computation of homology groups for digital images. In M. Ferri, P. Frosini, C. Landi, A. Cerri, and B. Di Fabio, editors, *Computational Topology in Image Context: CTIC 2012*, volume 7309 of *Lecture Notes in Computer Science*, pages 49 – 57. Springer, 2012.

[88] HOL Multivariate Analysis Library. `http://isabelle.in.tum.de/library/HOL/HOL-Multivariate_Analysis/index.html`, 2014.

[89] L. Hogben. *Handbook of Linear Algebra*. (Discrete Mathematics and Its Applications). Chapman & Hall/CRC, 1 edition, 2006.

[90] J. Hölzl, F. Immler, and B. Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. 7998:279–294, 2013.

[91] B. Huffman and O. Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, pages 131–146, 2013.

[92] H. Hummel. Will Computers Redefine the Roots of Math?, 2015. `https://www.quantamagazine.org/20150519-will-computers-redefine-the-roots-of-math/`.

[93] M. S. Hung and W. O. Rom. An application of the Hermite normal form in integer programming. *Linear Algebra and its Applications*, 140:163 – 179, 1990.

[94] N. Jacobson. *Basic Algebra I: Second Edition*. Dover Books on Mathematics. Dover Publications, 2012.

[95] R. Kannan and A. Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *siam Journal on Computing*, 8(4):499–507, 1979.

[96] C. Kapulkin, P. LeFanu Lumsdaine, and V. Voevodsky. Univalence in Simplicial Sets. *ArXiv e-prints*, March 2012.

[97] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962. Translation in *Physics-Doklady* 7: 595-596, 1963.

[98] A. Katovsky. Category Theory in Isabelle/HOL. Technical report, 2010. `http://apk32.user.srcf.net/Isabelle/Category/Cat.pdf`.

[99] G. Klein et al. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009, Big Sky, Montana, USA*, pages 207–220, 2009.

[100] V.N. Kublanovskaya. Certain algorithms for the solution of the complete problem of eigenvalues. *Soviet Mathematics. Doklady*, 2:17–19, 1961.

[101] O. Kunčar and A. Popescu. From types to sets in isabelle/hol. In *Isabelle Workshop*, 2014.

[102] A.N. Langville and C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2011.

[103] M. Lecat. *Erreurs de mathématiciens des origines à nos jours*. Castaigne, 1935.

[104] S.J. Leon. *Linear Algebra with Applications*. Featured Titles for Linear Algebra (Introductory) Series. Pearson Education, 2014.

[105] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[106] W. Li and L. C. Paulson. A modular, efficient formalisation of real algebraic numbers. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 66–75, 2016.

[107] B. Liu and H.J. Lai. *Matrices in Combinatorics and Graph Theory*. Network Theory and Applications. Springer, 2000.

[108] A. Lochbihler. Light-weight containers for Isabelle: efficient, extensible, nestable. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 116–132. Springer, 2013.

[109] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[110] G. Mackiw. A Note on the Equality of the Column and Row Rank of a Matrix. *Mathematics Magazine*, 68-4:285–286, 1995.

[111] J.P. May. *Simplicial Objects in Algebraic Topology*. Chicago Lectures in Mathematics. University of Chicago Press, 1993.

[112] R. Milner, R. Harper, D. MacQueen, and M. Tofte. *The Definition of Standard ML, revised edition*. MIT Press, 1997.

[113] The MLton website. `http://mlton.org/`, 2015.

[114] J. V. Neumann. *Mathematical Foundations of Quantum Mechanics*. Investigations in physics. Princeton University Press, 1955.

[115] M. Newman. *Integral matrices*. Pure and Applied Mathematics. Elsevier Science, 1972.

[116] T. R. Nicely. Enumeration to 1014 of the twin primes and Brun's constant. *Virginia Journal of Science*, 46(3):195 – 204, 1995.

[117] T. Nipkow. Gauss-Jordan Elimination for Matrices Represented as Functions. *Archive of Formal Proofs*, 2011. `http://afp.sf.net/entries/Gauss-Jordan-Elim-Fun.shtml`, Formal proof development.

[118] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[119] L. Noschinski. Graph theory. *Archive of Formal Proofs*, April 2013. `http://afp.sf.net/entries/Graph_Theory.shtml`, Formal proof development.

[120] S. Obua. Partizan Games in Isabelle/HOLZF. In *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, pages 272–286, 2006.

[121] G. O'Keefe. Towards a Readable Formalisation of Category Theory. *Electr. Notes Theor. Comput. Sci.*, 91:212–228, 2004.

[122] G. O'Keefe. Category Theory to Yoneda's Lemma. *Archive of Formal Proofs*, April 2005. `http://afp.sf.net/entries/Category.shtml`, Formal proof development.

[123] V. Y. Pan. Computation of approximate polynomial gcds and an extension. *Information and Computation*, 167(2):71–85, 2001.

[124] L. C. Paulson. *Logic and Computer Science*, chapter Isabelle: The next 700 theorem provers, pages 361 – 388. Academic Press, 1990.

[125] L. C. Paulson. Set Theory for Verification: I. From Foundations to Functions. *Journal of Automated Reasoning*, 11(3):353–389, 1993.

[126] L. C. Paulson. Set Theory for Verification. II: Induction and Recursion. *Journal of Automated Reasoning*, 15(2):167–215, 1995.

[127] L. C. Paulson. *ML for the Working Programmer (2Nd Ed.)*. Cambridge University Press, New York, NY, USA, 1996.

[128] L. C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *Journal of Automated Reasoning*, 55(1):1–37, 2015.

[129] L. C. Paulson. Isabelle's Logics: FOL and ZF. Technical report, 2015. `https://isabelle.in.tum.de/dist/Isabelle2016/doc/logics-ZF.pdf`.

[130] The Poly/ML website. `http://www.polyml.org/`, 2015.

[131] J. Ramanujam. Beyond unimodular transformations. *The Journal of Supercomputing*, 9(4):365–389, 1995.

[132] G. E. Reeves. What really happened on Mars?, 1997. `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html`.

[133] J. Rehmeyer. Voevodsky's Mathematical Revolution, 2013. `http://blogs.scientificamerican.com/guest-blog/voevodskye28099s-mathematical-revolution/`.

[134] S. Roman. *Advanced Linear Algebra*. Springer, 3rd edition, 2008.

[135] B. Russell. *The Principles of Mathematics*. W. W. Norton and Company, 2 edition, 1903.

[136] A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim, and P. Norvig. Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999.

[137] C. Sternagel. Proof Pearl - A Mechanized Proof of GHC's Mergesort. *Journal of Automated Reasoning*, 51(4):357 – 370, 2013.

[138] C. Sternagel and R. Thiemann. Abstract rewriting. *Archive of Formal Proofs*, June 2010. `http://afp.sf.net/entries/Abstract-Rewriting.shtml`, Formal proof development.

[139] C. Sternagel and R. Thiemann. Executable matrix operations on matrices of arbitrary dimensions. *Archive of Formal Proofs*, June 2010. `http://afp.sf.net/entries/Matrix.shtml`, Formal proof development.

[140] A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2000.

[141] G. Strang. The Fudamental Theorem of Linear Algebra. *The American Mathematical Monthly*, 100(9):848–855, 1993.

[142] G. Strang. *Introduction to Linear Algebra*. 2003.

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

[144] P. Suppes. *Axiomatic Set Theory*. Dover Publications, 1960.

[145] R. Thiemann. Implementing field extensions of the form $\mathbb{Q}[\sqrt{b}]$. *Archive of Formal Proofs*, February 2014. `http://afp.sf.net/entries/Real_Impl.shtml`, Formal proof development.

[146] R. Thiemann and C. Sternagel. Certification of termination proofs using ceta. In *Theorem Proving in Higher Order Logics*, pages 452–468. Springer, 2009.

[147] R. Thiemann and A. Yamada. Algebraic Numbers in Isabelle/HOL. *Archive of Formal Proofs*, December 2015. `http://afp.sf.net/entries/Algebraic_Numbers.shtml`, Formal proof development.

[148] R. Thiemann and A. Yamada. Matrices, Jordan Normal Forms, and Spectral Radius Theory. *Archive of Formal Proofs*, August 2015. `http://afp.sf.net/entries/Jordan_Normal_Form.shtml`, Formal proof development.

[149] V. E. Tourloupis. Hermite normal forms and its cryptographic applications. Master's thesis, University of Wollongong, 2013.

[150] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[151] M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, pages 167–184, 1999.

[152] M. Wenzel. *Isabelle/Isar - a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[153] A.N. Whitehead and B. Russell. *Principia Mathematica*. Principia Mathematica. University Press, 1910.

[154] D. Zill. *A First Course in Differential Equations with Modeling Applications*. Cengage Learning, 2012.