Under consideration for publication in Formal Aspects of Computing

Formalisation of the Computation of the Echelon Form of a Matrix in Isabelle/HOL

Jesús Aransay and Jose Divasón

Abstract. In this contribution we present a formalised algorithm in the Isabelle/HOL proof assistant to compute echelon forms, and, as a consequence, characteristic polynomials of matrices. We have proved its correctness over Bézout domains, but its executability is only guaranteed over Euclidean domains, such as the integer ring and the univariate polynomials over a field. This is possible since the algorithm has been parameterised by a (possibly non-computable) operation that returns the Bézout coefficients of a pair of elements of a ring. The echelon form is also used to compute determinants and inverses of matrices. As a by-product, some algebraic structures have been implemented (principal ideal domains, Bézout domains, etc.). In order to improve performance, the algorithm has been refined to immutable arrays inside of Isabelle and code can be generated to functional languages as well.

Keywords: Theorem proving; Isabelle/HOL; Linear Algebra; Verified Code Generation

1. Introduction

A classical mathematical problem is the transformation of a matrix over a ring into a canonical form (*i.e.*, a unique matrix that is *equivalent* in some sense to the original matrix, and that provides a explicit way to distinguish matrices that are not *equivalent*), 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, including S. Leon [Leo14], 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, including A. Storjohann [Sto00], 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: as L. Hogben shows, its existence can be *proved* over matrices whose coefficients belong to a Bézout domain [Hog06].

Correspondence and offprint requests to: Jesús Aransay and Jose Divasón. Departamento de Matemáticas y Computación, C/ Luis de Ulloa 2, Edificio Juan Luis Vives, Universidad de La Rioja, 26004 Logroño, La Rioja, Spain.
E-mails: jesus-maria.aransay@unirioja.es and jose.divasonm@unirioja.es

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 an echelon form is that, as a by-product, the characteristic polynomials can be easily obtained (even though there exist other more efficient ways of computing characteristic polynomials). Characteristic polynomials play a key role in Linear Algebra: the roots of the characteristic polynomial of a matrix are its *eigenvalues* and from them the *eigenvectors* can be obtained. Among the numerous applications of these concepts are biometrics [Fuk13] and the mathematical foundations of quantum mechanics [VN55]. The characteristic polynomial is also a powerful tool to simplify computations where matrices are involved.

In this work, we present a formalisation in Isabelle/HOL of an algorithm to obtain the echelon form of a given matrix. (The full development can be found at [DA15a].) 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; from its definition, 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 the Bézout coefficients in a Bézout domain, the algorithm would become computable in that structure as well. These transformations into echelon forms allow one to compute 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 work 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, parameterised by the 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 that we developed for the formalisation and refinement of the Gauss-Jordan algorithm [DA14]. This work shows its usefulness.

The paper is divided as follows. Section 2 presents some previous mathematical concepts that are important in our development as well as some other related formalisations. In Section 3 we show the algebraic structures we have implemented in Isabelle and the hierarchy of the involved classes. In Section 4 we explain both the formalisation of the algorithm and its relationship with the infrastructure presented in [DA14]. Section 5 presents the formalisation 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. Section 6 shows some related work presented in the literature. Finally, conclusions and possible further work are presented in Section 7. The Isabelle code presented in the paper is freely available from the Isabelle Archive of Formal Proofs [DA15a]. Let us note that Isabelle libraries are continuously evolving. We present the Isabelle code in the time this paper is written, that is, after the Isabelle 2016 official release. Modifications on the library are being carried out (particularly, many definitions that we introduce in Section 3 are being incorporated to the Isabelle library by M. Eberl and F. Haftmann), and some results, instances and structures presented here could be changed in the Isabelle repository version during their process of adoption as part of the standard library for the next Isabelle release.

2. Preliminaries

2.1. Linear Algebra and Echelon Forms

To begin with, we are going to introduce the main mathematical concepts which our development is based on. 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. We use PID (principal ideal domain) to mean a PIR which has no zero divisors. Confusingly, some authors (e.g. M. Newman [New72]) use PIR to refer to what we call PID. 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 [Sto00]. In the sequel, we assume that F is a field and R a commutative ring with a unit.

This work presents the formalisation of an algorithm which transforms a matrix into its echelon form, hence the most important concepts related to such a transformation must be defined. There are three types of elementary row (column) operations over a matrix $A \in M_{m\times n}(R)$:

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

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

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

Definition 2. 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 is upper triangular, so it is straightforward to compute its determinant. Furthermore, the *reduced row echelon form* is another useful matrix canonical form, since it is the output of the Gauss-Jordan algorithm.

Definition 3. A matrix $A \in M_{m\times n}(R)$ 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. It is a well-known result that over more general rings than fields it could not be possible 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:

- The Hermite normal form. It is the natural generalisation of the reduced row echelon form for PIDs, although it is normally studied only in the case of integer matrices. One of its primary uses is to solve systems of linear Diophantine equations over PIDs.
- The Smith normal form. It is useful in topology for computing the homology of a simplicial complex.
- The minimal echelon form.
- The Howell form.

A. Storjohann [Sto00] gives a detailed account of these canonical forms and presents algorithms to compute them.

On another note, there are three important concepts that must be introduced: eigenvalues, eigenvectors and the characteristic polynomial of a matrix.

Definition 4. A scalar λ is an *eigenvalue* for a matrix $A \in M_{n\times n}(R)$ if there exists a nonzero column vector x for which

$$Ax = \lambda x$$

In this case, x is called an *eigenvector* for A associated with λ.

Definition 5. The *characteristic polynomial* of $A \in M_{n\times n}(R)$ is the polynomial defined as the determinant of the polynomial matrix $tI - A$ (where t denotes the polynomial variable, and I the identity matrix), that is, $\det(tI - A)$.

Such concepts play an important role on mathematics due to their significant applications and are related with the Cayley-Hamilton theorem:

Theorem 1. The Cayley-Hamilton theorem for matrices. Every square matrix over a commutative ring satisfies its own characteristic polynomial.

S. Adelsberger *et al.* have formalised the Cayley-Hamilton theorem in Isabelle/HOL [AHP14]. Our development will rely on this development to reuse the definition and some properties of characteristic polynomials.

2.2. Isabelle, type classes and Gauss-Jordan algorithm

The importance and use of theorem provers are increasing nowadays, both in the formalisation of mathematical results and in the verification of algorithms. Isabelle is one of the most used and well-known proof assistants, on top of which different logics are implemented; the most explored of them is higher-order logic (or HOL), and it is also the one where the greatest number of tools (code generation, automatic proof procedures) and developments are available. It has been successfully used, for instance, in the proof of the Kepler conjecture by T. Hales *et al.* [HAB+15] (the largest formal proof completed to date) and in the formal verification of seL4, an operating-system kernel, by G. Klein *et al.* [KEH+09]. A simple but precise way to describe Isabelle/HOL is as functional programming plus logic. Some of the notation in the sequel will exploit this functional flavour; for instance, by $f\,a\,b$ we will refer to the function f applied to arguments a and b (or, even in a more functional jargon, the function $f\,a$ applied to the argument b).

The HOL Multivariate Analysis Library (or, *HMA* for short) is a set of Isabelle/HOL theories that has been successfully used in concrete developments in Analysis, Topology and Linear Algebra. It contains (in the current Isabelle release) more than 4,000 lemmas and 250 definitions and is based on the impressive work of J. Harrison in HOL Light [Har13]. One of the keys of the library is the representation of matrices; this representation was successfully applied in the formalisation of mathematics in various theorem provers, because of its succinctness and its taking advantage of the underlying type system; vectors are represented as functions over an underlying finite type; matrices as vectors of vectors. Our development is based on this library.

Isabelle/HOL provides axiomatic type classes [Haf16b], which allow organising polymorphic specifications. Essentially, they combine an operational aspect (in the manner of Haskell) with a logical aspect, both managed uniformly. A type class C specifies assumptions $P_1, \ldots, P_k$ for fixed constants and operations $c_1, \ldots, c_m$ and may be based on other type classes $A_1, ..., A_n$. The command *class* declares type classes in Isabelle/HOL. Just one type variable α is allowed to occur in the type class specification. It is said that a type β is an *instance* of the type class C if it provides definitions for the respective constants and respects the required assumptions. For example, the Isabelle type *int* which represents the integer numbers is an instance of the *semigroup* class. The command *subclass* establishes relationships (inclusions) among classes, allowing one to inherit facts and definitions. Code can also be generated from type classes to functional languages in a Haskell-like manner. In this work, we make use of type classes to represent algebraic structures and their relationships.

In [DA14] we presented an infrastructure to formalise, execute and refine algorithms of Linear Algebra in Isabelle/HOL. The development is based on *HMA* and goes together with a case study: the formalisation of the Gauss-Jordan algorithm. The Gauss-Jordan algorithm transforms a matrix (whose coefficients belong to a field) into another matrix in *reduced row echelon form* by means of *elementary operations*. We formalised the elementary operations over matrices and its properties, which can be reused for proving the correctness

of other Linear Algebra algorithms. We also set up Isabelle to generate code from the matrix representation presented in *HMA*. Additionally, a refinement to immutable arrays was carried out in such a way that the algorithm obtained a remarkable performance, even compared with an equivalent imperative version of the algorithm (see [AD15a]).

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 solutions of systems of linear equations (in every case: unique solution, multiple solutions and no solution). Verified code of these computations is generated to both Standard ML and Haskell.

It is worth remarking that the Gauss-Jordan algorithm is applied to matrices whose elements belong to a field (the generalisation from $\mathbb{R}$ to fields was presented in [AD15b]). 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.

2.3. Matrices over rings

Despite the fact that matrices over fields have been more studied, matrices over rings possess remarkable applications. Two well-known examples of this kind of matrices are integer matrices and polynomial matrices (also denoted as $\mathbb{Z}$ and $F[x]$-matrices respectively).

Integer matrices are widely used in graph theory [BW04] and combinatorics [LL00]. 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 [Zil12] and factor analysis in statistics [Chi06]) and in other varied fields, such as biometrics [Fuk13], quantum mechanics [VN55] and solid mechanics [Bat03]. They are also used in the PageRank computation [LM11] (the algorithm used by Google Search to rank websites in their search engine results).

3. Algebraic structures, formalisation and hierarchy

In this section, we recall mathematical definitions of the main algebraic structures involved in the development as well as their formalisation that we have carried out in Isabelle. Throughout the paper, by *gcd* we mean greatest common divisor.

Figure 1 shows the hierarchy of the main Isabelle type classes involved in the development. The arrows express strict inclusions (all of them have been proved 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 1 shows the main ones. The structures that we have had to introduce are presented in bold.

The algebraic structures presented in this section and their properties have been formalised in the file *Rings2.thy* of [DA15a]. Let us start with the mathematical concept of *GCD ring* and *GCD domain*.

Definition 6. 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*.

As it is shown below, our Isabelle implementation does not fix a *gcd* operation. The existence of a *gcd* for each pair of elements a and b is just assumed. Similarly, when required, the existence of the Bézout coefficients for any two elements a and b is simply assumed, but nothing is made explicit about how to compute them.

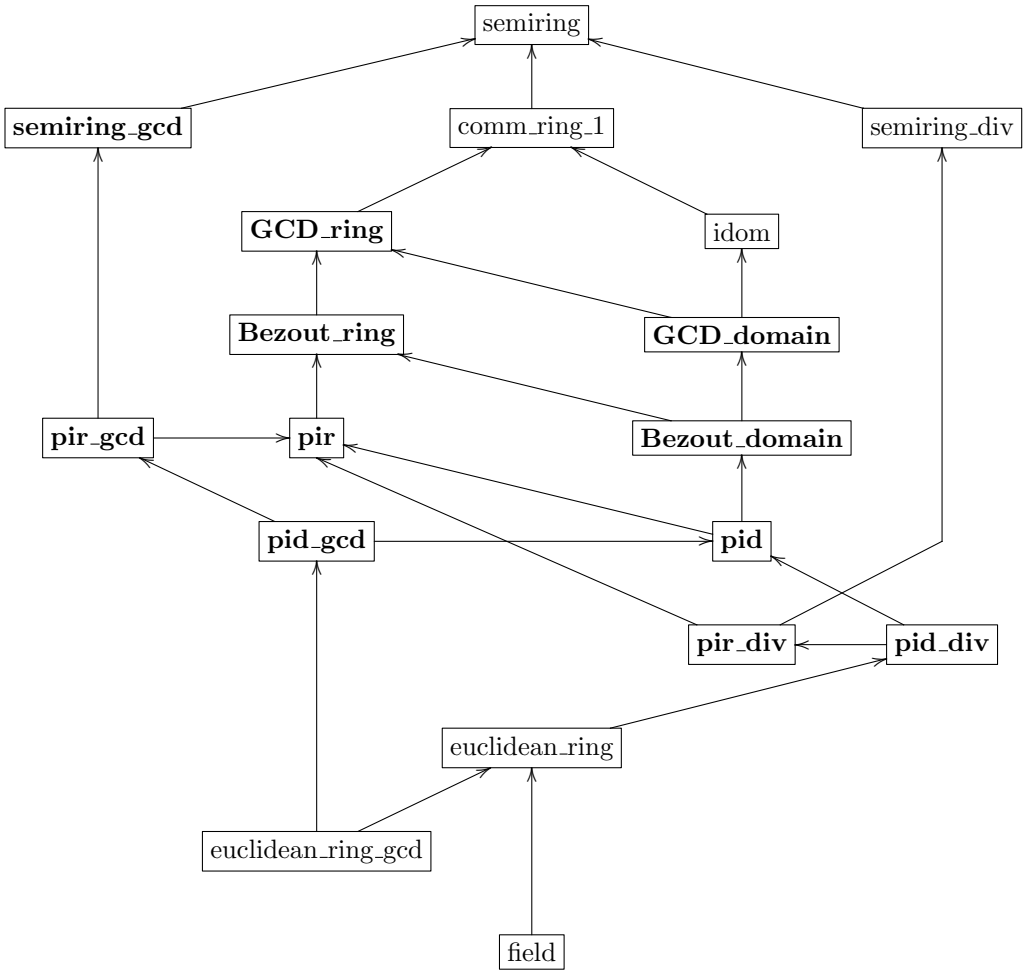

Fig. 1. Hierarchy of the main classes appearing in the development (arrows express inclusions; transitive ones are left out). The structures that we have had to introduce are presented in bold.

Thanks to this design decision, we can represent non-constructive structures (in the sense that no witnesses could be obtained for the operations). Moreover, structures where both the greatest common divisor and the Bézout coefficients are non-unique are also considered.

```
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 7. A *Bézout ring* R is a ring where the Bézout 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
```

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

Definition 8. 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 9. 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, we have proved some important lemmas (maybe not crucial for our development, but indeed for Ring Theory) over this structure. For instance the *ascending chain condition*, which is fundamental for proving that any PID is a unique factorization domain.

Theorem 2. 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

$$I_1 \subset I_2 \subset \cdots$$

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.

```
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)"

end
```

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 proof 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 greatest common divisor, completing the proof.

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

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

Definition 10. 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 $f(r) < f(b)$.

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. It was developed by M. Eberl [Ebe15] as part of his formalisation of a decision procedure for univariate polynomials. In his terminology, he uses ring to refer to an integral domain.

```
class euclidean_semiring = semiring_div + normalization_semidom +
  fixes euclidean_size :: "'a ⇒ nat"
  assumes size_0 [simp]: "euclidean_size 0 = 0"
  assumes mod_size_less:
    "b ≠ 0 ⟹ euclidean_size (a mod b) < euclidean_size b"
  assumes size_mult_mono:
    "b ≠ 0 ⟹ euclidean_size a ≤ euclidean_size (a * b)"

class euclidean_ring = euclidean_semiring + idom
```

Note that one additional operation is fixed by M. Eberl: the `normalisation_factor`, included in the class `normalisation_semidom`. This operation returns *a unit* such that dividing any element of the ring by its normalisation factor yields the same result for all elements in the same association class, effectively normalising the element.[1] For instance, for integers, a normalisation factor is the sign (dividing two associated integers by their respective signs we obtain the same result). For polynomials, a normalisation factor is the leading coefficient (two associated polynomials divided by their leading coefficients yield the same monic polynomial). The Isabelle definition uses `euclidean_size` to represent the Euclidean norm introduced in Definition 10.

Both the integers ($\mathbb{Z}$) and the univariate polynomials over a field ($F[x]$) form Euclidean domains. In

[1] Two elements $a, b \in R$ are said to be associates if $a = ub$, where u is a unit.

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 greatest common divisor of any two elements. Furthermore, this algorithm can always be used to obtain the Bézout coefficients. This constructive operation is presented in Isabelle in the *euclidean_ring* class with the name of *gcd_euclid*.

M. 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

subclass (in euclidean_semiring_gcd) semiring_gcd
```

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*, etc.) and possibly non-constructive structures (where it is just assumed the existence of the operations, for instance *pir*, *Bezout_domain*, etc.).

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 [Jac12, Rom07, FS01]. The following chain of strict inclusions is satisfied (as it is said in this section, all of them have been proved in Isabelle):

$$\text{Field} \subset \text{Euclidean ring} \subset \text{Principal ideal ring} \subset \text{Bézout ring} \subset \text{GCD ring}$$

4. Parametric algorithms and proofs

In Section 1 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 known to exist, 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 greatest common divisor 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. The operator `SOME` is used in Isabelle/HOL to describe Hilbert's choice operator (also known as ϵ); thanks to its axiomatic definition, a witness for purely existential predicates can be chosen. If an operation to obtain the Bézout coefficients is defined by means of the `SOME` operator, 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 (over constructive Bézout domains neither) because it is not possible to prove 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 `gcd` normalised over such a class:

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

Then, now one could prove that: `gcd_bezout_ring_norm = gcd_eucl`. This would allow us to execute the `gcd` function, but with the 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 we can prove the correctness of the algorithm over Bézout domains since in such structures there always exists a possibly non-constructive operation which satisfies such a predicate. In addition, we will always be able to execute it over Euclidean domains, since we can prove that there exists a computable `bezout` operation which satisfies the properties.

An interesting discussion and a solution on the topic of *executing* Russell's definite description operator ι (under favourable premises) in Isabelle/HOL by A. Lochbihler and L. Bulwahn is available [LB11]. For a given predicate P, they compute the set of every x such that $P\,x$; if the set is a singleton, ι returns such value; otherwise it throws an exception. In our particular case, solutions in Bézout domains could be non-computable. Therefore, we chose to use an underspecified parameter `bezout` in that setting, that is replaced, in the particular case of Euclidean domains, by the computable *gcd*.

The properties that a `bezout` operation must satisfy are fixed by means of the predicate `is_bezout_ext`. Let us first introduce the definition of the *Bézout coefficients*.

Definition 11. Given a ring R, and $a, b \in R$, the elements p, q such that

1. $pa + qb = d$
2. d is a greatest common divisor of a and b

are the *Bézout coefficients* of a and b. Because of the definition of *gcd*, there also exist elements $u, b \in R$ such that $du = -b$ and $dv = a$.

The coefficients u and v are the elements that will be used later to define the *Bézout matrices* (see Definition 12) that are used to compute the echelon form. It is relevant to note that they cannot be directly defined as $u = -b/d$ and $v = a/d$ because in abstract structures, such as Bézout domains, we do not have an explicit division operation (if we do so, we would have to work over a `bezout_domain_div` class instead of using the more general `bezout_domain` one).

The Isabelle definition of Bézout coefficients (in the form of a predicate, and considering the five coefficients presented in Definition 11) follows:

```
definition is_bezout_ext :: "('a ⇒ '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)"
```

In the following lemma we prove that there exists a (non-computable) function satisfying such a predicate over a *Bézout ring* (see Definition 7). Do note that the Bézout coefficients of any pair (a, b) are known to exist in this structure, but they might not be uniquely determined, neither computable. Therefore, in the lemma, when we introduce the witness function `bezout_ext`, we have to use the Isabelle operator `SOME` to choose the Bézout coefficients:

```
context bezout_ring
begin

lemma exists_bezout_ext: "∃ bezout_ext. is_bezout_ext bezout_ext"
proof -
  def bezout_ext ≡ "λa b. (SOME (p,q,u,v,d). p * a + q * b = d
    ∧ (d dvd a) ∧ (d dvd b) ∧ (∀d'. d' dvd a ∧ d' dvd b ⟶ d' dvd d) ∧ d * u = -b ∧ d * v = a)"
```

Finally, when we move into Euclidean domains, we can define a computable operation (do note that `SOME` is no longer used in the definition of the witness) which satisfies the predicate `is_bezout_ext`.

```
context euclidean_ring
begin

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

lemma is_bezout_ext_euclid_ext2: "is_bezout_ext euclid_ext2"

end
```

Thanks to the lemma presented above, we know that there exists a constructive `bezout` operation over Euclidean domains. So 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 prove 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 predicate `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).

   ```
   lemma echelon_form_of_invertible:
     fixes A::"'a::{bezout_domain}^'cols::{mod_type}^'rows::{mod_type}"
     assumes ib: "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` is defined over Euclidean domains, is computable and satisfies the predicate `is_bezout_ext`, we will have a computable algorithm and the `corollary` stating that the algorithm produces a matrix in echelon form contains *no premises* (no **assumes** clause in the Isabelle statement). Unconditional statements play a key role in the Isabelle code generation process (Section 4.3).

   ```
   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)"
```

4.1. An algorithm computing the echelon form of a matrix, parametrically

In this section, we present the parameterised 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 an operation *bezout* that must satisfy the Isabelle predicate `is_bezout_ext`, a column k and a tuple (A, i), where A is the matrix, and i the position of the column k where the pivot should be placed; the output is another tuple (A', i'), where A' is the matrix A with the elements of column k equal to zero below i, and i' is the position where the next pivot should be placed in column $k+1$. 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) 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, $(A, i + 1)$ 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 as A'. Hence, $(A', i + 1)$ 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.

Definition 12. Given a matrix A and two of its coefficients $A_{a,j}$, $A_{b,j}$ (both in the same column), the (elementary) *Bézout matrix* associated to them is:

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

where (p, q, u, v, d) denote the Bézout coefficients of $A_{a,j}$ and $A_{b,j}$ such that $pA_{a,j} + qA_{b,j} = d$, $du = -A_{b,j}$ and $dv = A_{a,j}$. The coefficients not explicitly shown in the matrix are assumed to be 0, except for the ones in the diagonal, which are 1.

Their implementation in Isabelle follows (the binder χ is used to define elements of type `'a^'rows` as functions over the type `'rows`, and in this particular case is used iteratively to define a matrix, an element of type `'a^'cols^'rows` in terms of variables `x` and `y` representing respectively the rows and the columns of the matrix; it is natural to think of χ as the λ binder but for defining vectors; then, the symbol `$` is used for access operations over the matrix indexes):

```
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 = (χ x y.
```

```
    (let
      (p, q, u, v, d) = bezout (A $ a $ j) (A $ b $ j)
     in
       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

The Bézout matrices have good properties, such as being *invertible* and having *determinant* equal to 1.

Moreover, the elementary Bézout matrix of two coefficients $A_{a,j}$ and $A_{b,j}$ satisfies the following relevant property (when multiplied to the left times A):

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

Now, we can iteratively 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 the following 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 `bezout_iterate` in a 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 bezout A' k =
    (let (A, i) = A'
     in if (∀m≥from_nat i. A $ m $ from_nat k = 0) ∨ (i = nrows A) then (A, i) else
        if (∀m>from_nat i. A $ m $ from_nat k = 0) then (A, i + 1) 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))"
```

The previous operation is the one which carries out the four main steps presented at the beginning of this section. In Section 5.1, when we perform the refinements in order to improve performance, we will introduce some binders to avoid some repeated computations that appear in the previous definition (such as, for instance, `from_nat k`, `from_nat i` and `(∀m>from_nat i. A $ m $ from_nat k = 0)`). Since now we are concerned with formalisation, we postpone any attempt of optimisation.

Interestingly, we have reused in the definition `echelon_form_of_column_k` some of the operations presented in the Gauss-Jordan development, such as `interchange_rows` (the elementary operations were defined over matrices over rings in such a development in order to reuse them). Folding the operation 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 bezout) (A,0) [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` (the operation that returns the elements (p, q, u, v, d)) as an additional parameter. Thus, parameterising the algorithm with different `bezout` operations, different echelon forms of a matrix could be obtained. Moreover, this idea let us use the same algorithm definition for both Bézout domains and Euclidean domains.

4.2. 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 already presented in Section 2.1. As an auxiliary predicate, we introduce the definition of *echelon form up to the column k*.

Definition 13. A matrix $A \in M_{m\times n}(R)$ is said to be in *row echelon form up to column k* if:

1. For every zero (up to column k) row i, there is no row j, with $i < j$, such that it is a nonzero row up to column k .
2. For any two consecutive nonzero (up to column k) rows i and j, with $i < j$, the leading entry of j is to the right of the leading entry of i (in other words, its index is greater).

The Isabelle definitions of `is_zero_row_upt_k` (which is being reused from our previous formalisation of Gauss-Jordan [DA14]) and `echelon_form_upt_k` follow:

```
definition is_zero_row_upt_k :: "'rows ⇒ nat ⇒'a::{zero}^'columns::{mod_type}^'rows ⇒ bool"
  where "is_zero_row_upt_k i k A = (∀j::'columns. (to_nat j) < k ⟶ A $ i $ j = 0)"

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))))"
```

Then the predicate `echelon_form` (see Definition 3) will just be *echelon form up to the last column*:

```
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 [DA15a]. Just one remark: remember that our approach includes 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"
```

The final theorem of the formalisation of the algorithm is the following one:

Theorem 3. For every matrix $A \in M_{m\times n}(R)$ there exists a matrix E such that:

1. E is in *echelon form*.
2. There exists an invertible matrix P such that $PA = E$.

Its Isabelle statement follows:

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

Do note that our Isabelle implementation uses the function `echelon_form_of` to define the matrix E.

Thanks to the infrastructure developed in the Gauss-Jordan development [DA14], we have been able to reuse many definitions and properties, saving effort. Even so, the complete proof of the correctness of the algorithm has taken about 3,000 lines.

4.3. 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 can 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. In order to get executable specifications, one must restrict herself to a subset of Isabelle/HOL (for instance, the Hilbert choice operator ϵ and the Russell definite description operator ι are not, in general, executable). The execution then consists in successive equation rewritings; the left hand side of definitions and lemmas is replaced by their right hand side. The lemmas that can be used for code generation are exclusively unconditional ones (the ones that do not contain premises). For instance, lemma `det_echelon_form_of_det_setprod` in Section 5 cannot be used in code generation, but lemma `inverse_matrix_code_rings` (presented also in Section 5) can be (in fact, it is used, as the label `code_unfold` indicates).

The particular setup modifications that we have performed are quite related to the ones that we presented in our previous work about the Gauss-Jordan algorithm and its applications (see [AD14, Sect. 4]). Essentially, we are carrying out the natural data type refinement: from the abstract and non-executable datatype `vec` to its executable implementation as functions over finite types. The code equations are established by means of the `code abstract` and `code_unfold` labels [Haf16a].

The execution is carried out inside of Isabelle by means of the **value** command. We can specify the evaluation to be performed inside of Isabelle (by means of a `simp` suffix), and therefore the evaluation will be performed using exclusively the Isabelle trusted kernel (and the obtained evaluation can be labeled as a `lemma` in the system). This kind of evaluation is known to be quite inefficient. Alternatively, the specification can be translated to a functional language (our languages of choice are Standard ML and Haskell; in previous works [AD14, AD15a] we already detected the Standard ML interpreter Poly/ML [Pol] to be faster than the Haskell GHC compiler, at least for our particular case study), by means of the code generation machinery [Haf16a]. This technique is orders of magnitude faster than Isabelle evaluation, but still quite slow when using functions to represent vectors.

For instance, the following command computes (generating code to Standard ML) 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 development, 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 [DA15a]. 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 was not completed in 90 minutes.[2] We consider it fair to say that this representation and evaluation technique are valid for testing purposes. To improve the performance, we have refined the algorithm to a more efficient matrix representation based on immutable arrays and exported code to functional languages, reusing the infrastructure developed in the formalisation of the Gauss-Jordan algorithm. This is presented in Section 5.1.

4.4. 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 development of ours: the formalisation of the Gauss-Jordan algorithm and its applications (see Section 2.2). The main difference is that the Gauss-Jordan algorithm 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 already defined and their properties proved over general rings in the Gauss-Jordan development, 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 this fact has been proved 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 proofs 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 3 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 follow the same strategy for defining the algorithm and induction is applied over the columns. The elementary operations and its properties have been reused, in addition to the setup of the Isabelle code generator and the infrastructure for refining to immutable arrays.

5. 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.

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

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. 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 proved that b_m will 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 [AHP14], 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 [AHP14], 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
```

Note that the last output is `None`, since its corresponding input matrix was not invertible. `[: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 [AHP14], such as minors, adjugates, cofactor matrix, the evaluation of

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

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 [DA15a].

5.1. Code refinement

As we have said in Section 4.3, 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 Standard ML and Haskell, following the approach and intensively reusing the infrastructure presented in the Gauss-Jordan development [AD14]. Immutable arrays are polymorphic vectors, immutable sequences with constant-time access. They were successfully used in such a development to enhance performance, allowing us to apply that verified Gauss-Jordan algorithm in interesting real case studies [AD15a].

The previous algorithm `echelon_form_of` has to be redefined over immutable arrays. Additionally, we apply (and prove) some optimisations that also help to improve performance (for instance, binders are used to identify some computations that have to be performed various times, so that they are only performed once). After that, we have had to prove 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 (`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 the paper and every application (determinants, inverses, characteristic polynomial) has been refined to immutable arrays. 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 [Haf16a] for some introductory examples). In our case, we have made use of the serialisations presented in the Gauss-Jordan formalisation (such as *Vector.vector* and *IArray.array* to encode immutable arrays in Standard ML and Haskell respectively; and the type for representing integer numbers in the target languages). Moreover, 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 [Pol] and MLton [MLt] functions (which are not part of the SML Basis Library, but particular to each compiler), increases notably the performance. To serialise in Standard ML the `div` and `mod` integer operations we considered two alternatives:

1. Serialise the Isabelle `div` and `mod` integer operations to the corresponding ones (*IntInf.div* and *IntInf.mod*).
2. Serialise both of them to the operation *IntInf.divmod*, which returns the pair (*i IntInf.div j*, *i IntInf.mod j*). The SML Basis Library says that this is likely to be more efficient than computing both components separately.

In our development, the benchmarks showed that the best option was the second one. But in other formalisations the best option may be the first one, depending on the need of computing `div` and `mod` or just one of them. Hence:

```
constant "divmod_integer :: integer => _ => _" ⇀ (SML) "(IntInf.divMod ((_),(_)))"
```

The generated Standard ML code has about 2,400 lines (it is noteworthy that a substantial amount of this code is devoted to the definitions of *dictionaries*, the Standard ML mechanism to resemble Isabelle type classes [Haf16a]). We have said in Section 4.3 that the computation of the echelon form of a 8×8 integer matrix was not completed in more than 90 minutes using the matrix representation based on functions. Thanks to the refinements and the serialisations presented in this section, when code is exported to Standard ML, this computation takes 0.001 seconds of CPU time (the echelon form is computed in a similar time). If we perform a similar computation over a random 20×20 integer matrix using immutable arrays, the CPU time consumed 2.172 seconds.[4] More examples of execution of our algorithm are shown in

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

the file *Examples_Echelon_Form_IArrays.thy*. This computation time can significantly vary depending on the magnitude of the matrix coefficients, and also on the one of the intermediary *gcd* computations.

6. Related work

There are several formalisations of Linear Algebra in most proof systems, above all focusing the point on vector spaces properties. For instance, P. Rudnicki *et al.* [RST01] present a formalisation of commutative algebra in the Mizar system. In HOL Light, an impressive library [Har13] of theorems about n-dimensional Euclidean spaces was developed by J. Harrison. Nevertheless, algorithmic aspects have not been explored in these systems. M. Eberl [Ebe15] has completed a decision procedure for univariate real polynomials in Isabelle/HOL. Some of his work on Euclidean rings has been useful to us (indeed, he is leading an ongoing work to improve the representation of division and *gcd* structures in the Isabelle library, in the vein of some of the suggestions and ideas introduced in Section 3, that will benefit from both his and our developments). Apart from that, the goal of his work is focused on polynomials over the real numbers.

There is a formalisation of matrix algebra using arrays in ACL2 by R. Gamboa *et al.* [GCV03], but it is focused on vectors and matrices over numerical types, without considering other types of rings (such as polynomials). They have implemented the computation of determinants and inverses, by means of elementary row operations, but these computations are not formalised.

Probably the closest work to ours is the one by G. Cano *et al.* [CCD+16]. It is a formalisation of Linear Algebra over elementary divisor rings in Coq. They also present a formalisation of the Smith normal form. The algorithm they have implemented performs similar transformations to the ones we have presented in this paper. The main difference between Cano's 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 Bézout domains. The executability of the algorithm will depend on the existence of an executable `bezout` operation. Therefore, in Euclidean domains, execution is guaranteed, thanks to the `euclid_ext2` operation; in Bézout domains executability will depend on the existence of an executable operation. 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 by R. Thiemann and A. Yamada [TY15]. The algorithm that they formalise 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 by T. Coquand *et al.* [TAV12]. The Sasaki-Murao algorithm is specially designed to compute the determinant of square matrices over a commutative ring. In [TAV12, Sect. 4], 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, even if the algorithm is specially designed for that computation. When they generate code to Haskell that determinant is computed in 0.273 seconds. As we mentioned in Section 5.1, the echelon form algorithm takes 2.172 seconds in a similar computation. Our algorithm is not specialised in the computation of determinants (indeed, it works over non-square matrices), but instead it computes the echelon form, from which more information than the determinant, such as related canonical forms, can be obtained. A substantial difference between the work in Coq and ours is that they use lists, instead of arrays, to encode vectors and matrices. Furthermore, the Cayley-Hamilton theorem is also formalised in this system [OB10].

It is also worth mentioning the CoqEAL (standing for Coq Effective Algebra Library) effort [CDM13, DMS12]; the project, led by G. Gonthier, is devoted to develop a set of libraries and commodities over which algorithms over matrices can be implemented, proved correct, refined to list of lists, and finally executed, in a similar way to our approach in Isabelle. The Sasaki-Murao algorithm and the rank of matrices over fields [DMS12] are based on it.

A recent formalisation of decision procedures for univariate polynomial computation by A. Narkawicz *et al.* in PVS has been published [NMnD15]. It is based on Sturm's and Tarski's theorems and it is useful, for instance, in the computation of the roots of the characteristic polynomial of a matrix (the eigenvalues).

Narkawicz *et al.* also had to formalise some properties of matrices and the Gauss-Jordan algorithm. As it has already been mentioned, the Gauss-Jordan algorithm and its applications have been formalised in Isabelle by us [DA14]. The Gaussian elimination has been formalised by M. Denes in Coq [Dén13]. Another way to approximate eigenvalues is by means of the QR algorithm. A previous step is the QR decomposition, which was also formalised and refined in Isabelle by us [DA15b] and can be executed with symbolic computations.

7. Conclusions and further 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 proved over Bézout domains and its executability is guaranteed over constructive structures, such as Euclidean domains. In order to do that we have parameterised the functions of the algorithm by the operation *bezout*. The algorithm is proved correct for any choice for *bezout* operation. By instantiating *bezout* by a computable operation, the echelon form becomes computable. Therefore, the correctness of the algorithm is proved over non-constructive algebraic structures, and then the algorithm 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 [DA14] to more abstract rings. Such a Gauss-Jordan formalisation has intensively been reused: the infrastructure developed there (elementary operations, code generator setup, refinement statements, matrix properties, etc.) has shown to be very useful. One sign of it is that the whole development of the echelon form took about 8,000 lines of Isabelle code, whereas the Gauss-Jordan formalisation needed about 15,000 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, etc.) and their properties (ideals, subgroups, relationships among them) have been formalised, enhancing the Isabelle library of rings using type classes.

Isabelle/HOL has a relevant amount of features, some of which have been crucial to our work. Following the order in which they were used in our work, *type classes* [Haf16b] are the first one that eased our development. The hierarchy introduced in Figure 1 was fully performed by means of type classes. The possibility to define structures and prove them *subclasses* of existing ones, automatically importing every result in the context of the first class greatly simplified the formalisation task. Moreover, type classes are naturally connected to code generation (specially in Haskell, but also in Standard ML thanks to dictionaries), easily obtaining data structures in the chosen functional language representing the Isabelle type classes. From the *logical perspective*, Hilbert's choice operator allowed us a direct representation of the *gcd* and *bezout* operations in Bézout and GCD domains. The way to connect these operations to the existing one in Euclidean domains was discussed in Section 4. Then, *code generation* [Haf16a] enables the direct translation of Isabelle executable specifications to functional languages.

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 Hermite normal form and the Smith normal form would be very interesting. This is feasible thanks to both the infrastructure already developed in the formalisation of the Gauss-Jordan algorithm and the Ring Theory presented in this contribution. In fact, the Hermite normal form can simply be obtained from the echelon form reducing the elements above the pivots. In addition, the computation of eigenvalues and eigenvectors from the characteristic polynomial would be desirable. On a different note, our refinement from the Isabelle `vec` type to *immutable arrays* did not benefit from the Isabelle machinery by B. Huffman and O. Kunčar [HK13] to *lift and transfer* specifications and proofs between data types. When we completed our previous work [AD15a], this facility was still "work in progress", but if it supports nested types, it should be now useful to ease the link between the different representations of matrices in the abstract setting and the one defined in our refinement.

Our algorithm to compute the echelon form (and hence, the characteristic polynomial) of a matrix relies on a function to compute the 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 [BvH82] and approximate [Pan01], would be interesting.

Acknowledgements

The authors would like to thank the anonymous referees because of their valuable ideas. Particularly, their suggestions helped us to improve the readability of the Isabelle code presented, and also the related work section. This work has been supported by the research grant FPI-UR-12, from Universidad de La Rioja and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

References

[AD14] J. Aransay and J. Divasón. Formalization and execution of Linear Algebra: from theorems to algorithms. In G. Gupta and R. Peña, editor, *PostProceedings of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013*, volume 8901 of *LNCS*, pages 01 – 19. Springer, 2014.

[AD15a] 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(e9):21, 2015. doi: 10.1017/S0956796815000155.

[AD15b] J. Aransay and J. Divasón. Generalizing a Mathematical Analysis library in Isabelle/HOL. In K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the Seventh NASA Formal Methods Symposium: NFM 2015*, 2015.

[AHP14] S. Adelsberger, S. Hetzl, and F. Pollak. The Cayley-Hamilton Theorem. *Archive of Formal Proofs*, 2014. `http://isa-afp.org/entries/Cayley_Hamilton.shtml`, Formal proof development.

[Bat03] K. J. Bathe. *Computational Fluid and Solid Mechanics*. Elsevier Science, 2003.

[BvH82] A. Borodin, J. von zur Gathen, and J. E. Hopcroft. Fast Parallel Matrix and GCD Computations. *Information and Control*, 52(3):241–256, 1982.

[BW04] L.W. Beineke and R.J. Wilson. *Topics in Algebraic Graph Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2004.

[CCD+16] G. Cano, C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Formalized Linear Algebra over Elementary Divisor Rings in Coq. *Logical Methods in Computer Science*, 2016.

[CDM13] C. Cohen, M. Dénès, and A. Mörtberg. Refinements for Free! In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs: CPP 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2013.

[Chi06] D. Child. *The Essentials of Factor Analysis*. Bloomsbury Academic, 2006.

[DA14] J. Divasón and J. Aransay. Gauss-Jordan Algorithm and Its Applications. *Archive of Formal Proofs*, 2014. `http://isa-afp.org/entries/Gauss_Jordan.shtml`, Formal proof development.

[DA15a] J. Divasón and J. Aransay. Echelon Form. *Archive of Formal Proofs*, 2015. `http://isa-afp.org/entries/Echelon_Form.shtml`, Formal proof development. Updated version available from the AFP repository version: `http://www.isa-afp.org/devel-entries/Echelon_Form.shtml`.

[DA15b] J. Divasón and J. Aransay. *QR* Decomposition. *Archive of Formal Proofs*, 2015. `http://isa-afp.org/entries/QR_Decomposition.shtml`, Formal proof development.

[Dén13] M. Dénès. *Formal study of efficient algorithms in Linear Algebra*. PhD Thesis, Université Nice Sophia Antipolis, 2013.

[DMS12] M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to Computational Algebra in COQ. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving: ITP 2012*, volume 7406 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[Ebe15] M. Eberl. A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 75–83, New York, NY, USA, 2015.

[FS01] L. Fuchs and L. Salce. *Modules Over Non-Noetherian Domains*. Mathematical surveys and monographs. American Mathematical Society, 2001.

[Fuk13] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Computer science and scientific computing. Elsevier Science, 2013.

[GCV03] R. Gamboa, J. Cowles, and J. Van Baalen. Using ACL2 Arrays to Formalise Matrix Algebra. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[HAB+15] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[Haf16a] F. Haftmann. Code generation from Isabelle/HOL theories. Tutorial documentation, 2016. `http://isabelle.in.tum.de/dist/Isabelle2016/doc/codegen.pdf`.

[Haf16b] F. Haftmann. Haskell-style type classes with Isabelle/Isar. Tutorial documentation, 2016. `http://isabelle.in.tum.de/dist/Isabelle2016/doc/classes.pdf`.

[Har13] J. Harrison. The HOL Light Theory of Euclidean Space. *Journal of Automated Reasoning*, 50(2):173 – 190, 2013.

[HK13] B. Huffman and O. Kunčar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs: CPP 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013.

[Hog06] J. Hogben. *Handbook of Linear Algebra*. (Discrete Mathematics and Its Applications). Chapman & Hall/CRC, 1 edition, 2006.

[Jac12] N. Jacobson. *Basic Algebra I: Second Edition*. Dover Books on Mathematics. Dover Publications, 2012.

[KEH+09] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[LB11] A. Lochbihler and L. Bulwahn. Animating the Formalised Semantics of a Java-like Language. In van Eekelen M., H. Geuvers, J. Schmalz, and F. Wiedijk, editors, *Interactive Theorem Proving (ITP 2011)*, volume 6898 of *Lecture Notes in Computer Science*, pages 216 – 232. Springer, 2011.

[Leo14] S. J. Leon. *Linear Algebra with Applications*. Featured Titles for Linear Algebra (Introductory) Series. Pearson Education, 2014.

[LL00] B. Liu and H.J. Lai. *Matrices in Combinatorics and Graph Theory*. Network Theory and Applications. Springer, 2000.

[LM11] A.N. Langville and C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2011.

[MLt] The MLton website. `http://mlton.org/`.

[New72] M. Newman. *Integral matrices*. Pure and Applied Mathematics. Elsevier Science, 1972.

[NMnD15] A. Narkawicz, C. Muñoz, and A. Dutle. Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's Theorems. *Journal of Automated Reasoning*, 54(4):285 – 326, 2015.

[OB10] S. Ould Biha. *Mathematical components for groups theory*. PhD Thesis, Université Nice Sophia Antipolis, 2010.

[Pan01] V. Y. Pan. Computation of approximate polynomial gcds and an extension. *Information and Computation*, 167(2):71–85, 2001.

[Pol] The Poly/ML website. `http://www.polyml.org/`.

[Rom07] S. Roman. *Advanced Linear Algebra*. Graduate Texts in Mathematics. Springer, 2007.

[RST01] P. Rudnicki, C. Schwarzweller, and A. Trybulec. Commutative Algebra in the Mizar System. *Journal of Symbolic Computation*, 32(1/2):143–169, 2001.

[Sto00] A. Storjohann. *Algorithms for Matrix Canonical Forms*. PhD Thesis, Swiss Federal Institute of Technology Zurich, 2000.

[TAV12] Coquand T., Mörtberg A., and Siles V. A formal proof of Sasaki-Murao algorithm. *Journal of Formalized Reasoning*, 5(1):27–36, 2012.

[TY15] R. Thiemann and A. Yamada. Matrices, Jordan Normal Forms, and Spectral Radius Theory. *Archive of Formal Proofs*, August 2015. `http://isa-afp.org/entries/Jordan_Normal_Form.shtml`, Formal proof development.

[VN55] J. Von Neumann. *Mathematical Foundations of Quantum Mechanics*. Investigations in Physics. Princeton University Press, 1955.

[Zil12] D. Zill. *A First Course in Differential Equations with Modeling Applications*. Cengage Learning, 2012.