Formalization and execution of Linear Algebra: from theorems to algorithms

J. Aransay and J. Divasón

Departamento de Matemáticas y Computación, Universidad de La Rioja,
Edif. Luis Vives, c. Luis de Ulloa s/n. 26004. Spain
{jesus-maria.aransay,jose.divasonm}@unirioja.es

Abstract. In this work we present a formalization of the *Rank Nullity* theorem of Linear Algebra in Isabelle/HOL. The formalization is of interest because of various reasons. First, it has been carried out based on the representation of mathematical structures proposed in the HOL Multivariate Analysis library of Isabelle/HOL (which is part of the standard distribution of the proof assistant). Hence, our proof shows the adequacy of such an infrastructure for the formalization of Linear Algebra. Moreover, we enrich the proof with an additional formalization of its *computational* meaning; to this purpose, we choose to implement the Gauss-Jordan elimination algorithm for matrices over fields, prove it correct, and then apply the Isabelle code generation facility that permits to *execute* the formalized algorithm. For the algorithm to be code generated, we use again the implementation of matrices available in the HOL Multivariate Analysis library, and enrich it with some necessary features. We report on the precise modifications that we introduce to get code execution from the original representation, and on the performance of the code obtained. We present an alternative verified type refinement of vectors that outperforms the original version. This refinement performs well enough as to be applied to the computation of the rank of some biomedical digital images. Our work proves itself as a suitable basis for the formalization of numerical Linear Algebra in HOL provers that can be successfully applied for computations of real case studies.

Keywords: Linear Algebra, Verification, Code generation

Introduction

In standard mathematical practice, formalization of results and execution of algorithms are usually (and unfortunately) rather separate concerns. Computer Algebra systems (CAS) are commonly seen as *black boxes* in which one has to trust, despite some well-known major errors in their computations, and mathematical proofs are more commonly carried out by mathematicians with *pencil & paper*, and sometimes *formalized* with the help of a proving assistant. Nevertheless, some of the features of each of these tasks (formalization and computation) are considered as a burden for the other one; computation demands optimized

versions of algorithms, and very usually *ad hoc* representations of mathematical structures, and formalization demands more intricate concepts and definitions in which proofs have to rely on.

In this paper, we present a case study in which we aim at developing a formalization in Linear Algebra in which computations are still posible. From an existing library in the Isabelle/HOL distribution (HOL Multivariate Analysis [15], *HMA* in the sequel), which has been fruitfully applied in the formalization of major mathematical results (both in this system and also in HOL-Light, that shares a similar representation), we formalize a mathematical result, known as the "Rank Nullity theorem".

The result is of interest by itself in Linear Algebra (some textbooks name it the *Fundamental theorem of Linear Algebra*) but it is even more interesting if we consider that each linear form between *finite dimensional* vector spaces can be represented by means of a *matrix* with respect to 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 behavior of the linear form, but change the underlying bases; the number of *non zero rows* of such a matrix is equal to the rank of the (original) linear form; the number of zero rows is the dimension of its *kernel*.

The best-known algorithm for the computation of the rref of a matrix is the Gauss-Jordan elimination method. We have implemented the algorithm over the representation of matrices in the HMA library; this representation was introduced by J. Harrison in HOL-Light and successfully applied in the formalization 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. *A priori*, finite enumerable types have nice computational features, since mathematical and logical operations (traversing, epsilon operator, universal or existential quantifiers) over them can be executed. We present here some additional features, relying in previous works, that enable these possibilities in Isabelle/HOL. In this work, we link the original statement of the Rank Nullity theorem together with the Gauss-Jordan elimination algorithm, and can use both tools to produce *certified* computations of the rank and kernel of linear forms.

As we will illustrate with some examples, the performance of the algorithm is rather poor, mainly because of the data structure used to represent matrices; the executable algorithm cannot be used for real applications, but only for tests (for instance, it could be used for experimental testing or as a *reference* algorithm for more optimized versions of it). Therefore, we introduce a data type refinement that allows us to obtain a version of the algorithm performing nicely in matrices of a considerable size (but still far from specialized Computer Algebra libraries).

The paper is structured as follows; in Section 1 we describe the Isabelle features in which our development is based on. In Section 2 we present the Rank Nullity theorem, as well as its Isabelle formalization. In Section 3 we introduce the notion of rref and the formalization of the Gauss-Jordan algorithm. In Section 4 we present the choices and setup of the Isabelle code generation tool

that enable to execute operations and algorithms. In Section 5 we bring together the previous ingredients and present the generated SML code from the original algorithm. Additionally, we present a refinement that enabled us to improve the performance of the certified algorithm. In Section 6 we draw some conclusions and present related works, as well as possible future research lines. The source files of the development are available from [2]; they have been developed under the Isabelle 2013 version. The previous web site also includes the SML code generated from the Isabelle specifications, and also the input matrices that have been used in the benchmarks presented in Section 5.

1 Isabelle/HOL

Isabelle [21] is a generic interactive proving assistant, on top of which different logics can be implemented; the most explored of these variety of logics is higher-order logic (or *HOL*), and it is also the one where the greatest number of tools (code generation, automatic proof procedures) are available. We do not aim to present here the fundamentals of Isabelle/HOL, just to introduce the main features that are used in our work.

The HOL type system is rather simple; it is based on non-empty types, function types ($\Rightarrow$) and type constructors κ that can be applied to already existing types (*nat, bool*) or type variables (α, β). 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 by means of the *typedef* command; the command takes a set defined by comprehension over a given type $\{x :: \alpha.\, P\, x\}$, and defines a new type σ. We will refer to this new type as *abstract*, and to the underlying one as *concrete* (this terminology is particular to the context of code generation, where the abstract type cannot be directly code generated, whereas the concrete one, under precise assumptions, can be; see [8] for details).

Isabelle also introduces type classes in a similar fashion to Haskell; a type class is defined by a collection of operators (over a single type variable) and premises over them. For instance, the HMA library has a type class *field* representing the algebraic structure. Concrete types (*real*, *rat*) can be proved to be *instances* of a given type class (*field* in our example). Type classes can be also used to impose additional restrictions over type variables; for instance, the expression $(x :: \alpha :: \mathit{field})$ imposes the constraint that the type variable α possess 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.

1.1 HOL Multivariate Analysis library

The HOL Multivariate Analysis library is a set of Isabelle theories which contains a wide range of results in different mathematical fields such as Analysis, Topology or Linear Algebra. They are based on the work of J. Harrison in HOL-Light [10],

which includes proofs of intricate theorems (such as the Stone-Weierstrass theorem) and has been successfully used as a basis for the Flyspeck project [11], aiming at formally verifying the proof of the Kepler conjecture by T. Hales. Among the fundamentals of the library, one of the keys is the representation of n-dimensional vectors over a given type ($\mathbb{F}^n$, where $\mathbb{F}$ stands for a generic field, or in Isabelle jargon a type variable $\alpha :: \mathit{field}$) taking into account that the HOL type system lacks the expressivity of dependent types. A detailed explanation can be found in [9, Section 2]. The idea is to represent vectors over α by means of *functions* from a finite type variable $\beta :: \mathit{finite}$ to α; for proving purposes, this type definition is usually sufficient; if we need to introduce vectors of a *concrete* dimension n, β can be replaced by a (finite) type of such cardinality (we present in Section 4 a possible representation of such types).

The Isabelle type definition is as follows; 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:

```
typedef (α,β) vec = UNIV :: ((β::finite) ⇒ α) set
 morphisms vec-nth vec-lambda ..
```

The previous type also admits in Isabelle the shorter notation $\alpha\hat{}\beta$. The idea of using underlying finite types for vectors indices has great advantages, as already pointed out by Harrison, from the formalization point of view. For instance, the type system enforces that operations on vectors (such as addition or multiplication) are only performed over vectors of equal dimension, *i.e.*, vectors which 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 flavor 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 one of vectors by iterating the previous construction (matrices over a type α will be terms of type $\alpha\hat{}m\hat{}n$, where m and n stand for finite type variables).

The HMA library already contains operations and properties of matrices defined in this way (multiplication, invertible matrices, the relationship between linear forms and matrices, determinants). Nevertheless, we missed some other standard results in Linear Algebra, that we had to introduce, such as the notion of coordinates with respect to a particular (not the canonical one) basis, the influence of changes of bases over a given matrix, or the elementary row (and column) operations over matrices (exchanging rows, multiplying a row by a constant and adding to a row another one multiplied by a constant). These elementary operations also give place to the notion of *elementary matrices*; indeed, these are the invertible matrices; each elementary matrix represents a change of bases.

Another subject that has not been explored in the Isabelle HMA library, or in HOL-Light, is the possibility to execute the previous data types and operations. As we will see in Section 4, the *finite* type class does not enable some operators

over vectors and matrices to be executed, and some additional type classes have to be used.

Finally, another aspect that has not been explored in the HMA library is numerical Linear Algebra. There is no implementation of common algorithms such as Gaussian elimination or diagonalization. We aim to show that the HMA library provides a framework where algorithms over matrices can be formalized, executed and coupled with their mathematical meaning.

1.2 Code generation

Isabelle/HOL offers a facility to generate code from specifications of data types, type classes and definitions over them, as long as these elements have an executable representation in the target languages (SML, Haskell, OCaml or Scala). The code generator is part of the trusted kernel of Isabelle [7].

As we explained before, 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 to 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 the 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 formalizations 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 is found in [7]; an interesting case study in [5].

In Section 5 we present two different refinements of the *vec* Isabelle type; the first one uses functions over finite domains, and is designed for simplicity. The second one uses immutable arrays (represented in the Isabelle type *iarray*) and presents a remarkable performance improvement when generated to SML.

2 The Rank Nullity theorem of Linear Algebra

The Rank Nullity theorem is a well-known result in Linear Algebra; the following formulation has been obtained from [22, Theorem 2.8].

Theorem 1 (The rank plus nullity theorem). *Let* $\tau \in L(V, W)$.

$$\dim(\ker(\tau)) + \dim(im\,(\tau)) = \dim(V)$$

or, in other notation,

$$rk\,(\tau) + null\,(\tau) = \dim(V)$$

In the previous statement, $L(V, W)$ denotes the set of linear forms between two given vector spaces V and W. It is worth noting that V must be a finite-dimensional vector space. Several textbooks impose the additional restriction of W being also finite-dimensional, but this restriction (as can be observed in the Isabelle formalization) is only needed in the version of the theorem for matrices representing linear forms (otherwise, we would have a matrix with an infinite number of columns representing the linear form). The following formalization [1] is part of the Isabelle repository; thanks to the infrastructure in the HMA library, it comprises a total of 380 lines of Isabelle code. 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 [16], 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 subset 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 formalization. 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 form 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 form 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).[1]

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 (labeled as *matrix-works*) which states that, given any linear form f, $f(x :: real\hat{}n)$ 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,

[1] In our opinion, this result is a typical example of a property that is unavoidable in a formalized proof, but usually skipped in paper & pencil proofs.

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^α^β
  shows DIM (real^α) = dim (null_space A) + dim (col_space A)
```

This statement is used to compute the dimensions of the rank and kernel of linear forms 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.

3 The Gauss-Jordan elimination method

There are several ways of computing the dimension of the range (and consequently of the kernel) of a linear form. In our development we choose the Gauss-Jordan elimination method. The main reason is that it has several different applications. For instance, it can be used to solve systems of linear equations; Nipkow [20] has proved that the Gauss-Jordan elimination algorithm is correct in this respect; the algorithm used in that work is very succinct, but works exclusively for input square matrices with unique solution, *i.e.*, whose rank is equal to their dimension. Nipkow proves that the algorithm is *complete* (under suitable circumstances, it generates a solution) and *correct* (it generates a vector which is a solution of the linear system). The algorithm we are formalizing differs from Nipkow's since we need an algorithm capable of dealing with non-square matrices whose rank can be smaller or equal than their number of rows. We also prove a different property of the algorithm than the one he proves; namely, that the rank of the input matrix is preserved through the algorithm steps (in other words, that the rank is an *invariant* of linear forms). Another difference is implicit here; in the HMA setting, linear forms and matrices are proved to be equivalent, whereas Nipkow uses an *ad hoc* matrix data type. In this sense, our implementation of Gauss-Jordan elimination would admit further applications, such as the computation of inverse matrices (or inverses of linear forms) and the computation of determinants. The algorithm is not optimal for any of those problems, but algorithmic refinements could be used in later stages to reach better performing algorithms for each of the previous tasks, once the mathematical properties of the original algorithm are stated and proved.

The Gauss-Jordan algorithm is based on the computation of the *reduced row echelon form* of (probably non-square) matrices. The *rref* of a matrix is defined as follows (see [22]):

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

4. Any column that contains a leading entry has 0's in all other positions.

The previous definition of rref is valid for non-square matrices. Interestingly, the rref (R) of a matrix A can be obtained by performing exclusively *row operations*, in such a way that $R = E_1 \dots E_k A$, where E_i denote elementary matrices; since elementary operations (and elementary matrices) preserve the rank of a matrix, computing the rank of A can be reduced to computing the rank of R (its number of nonzero rows). The code in the following formalization is available from [2] in files *Elementary_ Operations* and *Gauss_ Jordan*.

One way to compute the successive elementary row operations that produce the rref of a matrix is through the Gauss-Jordan elimination algorithm[2]; versions of the algorithm abound in the literature; however, we preferred to introduce our own version, designed to ease the formalization. In it, the algorithm is described by means of exclusively *elementary row operations* E_i (namely *interchange_rows*, *mult_row* and *add_row*), so that the rank of a matrix A is preserved because of the previous formula $R = E_1 \dots E_k A$. Additionally, the algorithm exploits the underlying (finite) representation of matrices, where both the indices of rows and columns are represented by *finite* types; both the types of columns and rows indices need to be *traversed*, and thus are restricted to be instances of the *enum* type class; this type class is part of the Isabelle library, and represents types for which the carrier set is *explicit*.

Algorithm 1 Gauss-Jordan elimination algorithm

Data: A is the input matrix;
$l \leftarrow 0$; ▷ l is the index where the pivot is to be placed after each iteration;
for $k \leftarrow 0, (ncols\ A) - 1$ **do**
  ▷ Check that there is a nonzero entry over index l in column k;
  if $nonzero\,l\,(col\,k\,A)$ **then**
    $i \leftarrow index\_nonzero\,l\,(col\,k\,A)$ ▷ Let i be the index of the first nonzero entry;
    $A \leftarrow interchange\_rows\,A\,i\,l$ ▷ Rows i and l are interchanged;
    $A\,l \leftarrow mult\_row\,A\,l\,(1/A\,l\,k)$ ▷ Row l is multiplied by $(1/A\,l\,k)$;
    for $t \leftarrow 0, (nrows\ A) - 1$ **do**
      if $t \neq l$ **then**
        $A\,t \leftarrow add\_row\,A\,t\,l\,(-A\,t\,k)$ ▷ Row t is added row l times $(-A\,t\,k)$;
      end if
    end for
    $l \leftarrow l + 1$
  end if
end for

[2] A somehow surprising point is that this algorithm is not even mentioned in [22], even if a detailed description of elementary operations over matrices, rref or invertible matrices is presented; this underscores our claim that algorithms and its mathematical meaning are often presented as different subjects.

The algorithm satisfies the following properties. When applied from column 0 up to column k, the first $k+1$ columns will be in rref. Note that implicitly we are imposing additional constraints on the types indexing columns (and rows); they must be inductive, since the proofs will be performed by induction over columns' indices; 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. Therefore, 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*, as can be noted in the Isabelle *type definition* of the algorithm:

```
definition Gauss_Jordan::'α::{inverse, uminus,
    semiring_1}^columns::{mod_type}^rows::{mod_type} => 'α^columns^rows
 where ...
```

In the previous algorithm definition we exclusively included the type classes required to *state* the algorithm; in the later proof of the algorithm, we have to restrict α 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 crucial result in the formalization of the algorithm preserving the rank of matrices is that elementary operations (*i.e.*, invertible matrices) applied to a matrix preserve its rank:

```
lemma fixes A::real^'n^'m and P::real^'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 forms are preserved by elementary operations (only the underlying bases change). Note that the previous machinery is not particular to our formalization, but could also be reused for different algorithms in numerical Linear Algebra. Additionally, we formalized a result stating that the previous algorithm produces a rref.

Moreover, the presented version of the algorithm is *executable*, as long as the rows and columns types can be generated to some type in the target languages; we present in Section 4 the details of that extraction.

4 Code generation from finite types

Up to now, we have used in our development an *abstract* data type *vec* (and its iterated construction 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.

The *finite* type class is enough to generate code from some abstract data structures, such as *finite sets*, which are later mapped in the target programming language (for instance, SML) to data structures such as lists or red black trees (see [19] for details and benchmarks). Our case study is a bit more demanding, since the indexing types of vectors and matrices 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".

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 types, whereas in an *enumerable* type this 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 these kind of calculations (only for enumerable types), which are *ad-hoc* for our case study, can be found in the file *Code_Set* of our development [2].

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 α bit0 = {0...<2 * CARD(α:: finite)}
typedef α bit1 = {0...<1 + 2 * CARD(α:: 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 (substraction 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* type class.

```
instantiation bit0 :: enum begin
definition (enum::α bit0 list)=map (Abs_bit0'∘ int) (upt 0 (CARD α 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, substraction, 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 class, enabling the computation of their cardinals (see file *Numeral_Type_Addenda* in [2] for the complete proofs).

5 Bringing it all back home: formalization and execution

In the previous section we have presented a setup that permits code generation of the vectors indexing types and their operations. Nevertheless, as we mentioned in Section 1.1, *vec* is itself an *abstract* type which 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, an executable version of the algorithm can be achieved, capable of computing the rref of 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 [23]).

In order to achieve the first refinement (from abstract matrices to *functions*), the type morphisms between the type *vec* and its counterpart (functions) have to be labeled precisely in the code generator setup.

lemma *[code_abstype]: vec_lambda (vec_nth v) = (v::α^β::finite)*

Additionally, every operation over the abstract data type has to be *mapped* to an operation over the concrete data type (and their behavioral 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. 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.(mfi * m'ij)) UNIV)*
lemma *[code abstract]: vec_nth (mat_mult_row m m' f) =*
*vec_lambda (λj. setsum (λi.(mfi * m'ij)) UNIV)*
lemma *[code abstract]: vec_nth (m ** m') = mat_mult_row m m'*

As long as our algorithm is based on (abstract) operations which are mapped to 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 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; 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 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; the evaluation can be also performed *in* Isabelle (and therefore the code generator would not intervene):

```
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] dim (null_space (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))
```

The previous computations have been carried out with matrices represented as functions. They are almost instantaneous, but the computation of the algorithm over matrices of size 10×10 is already very slow (several minutes).

The second aforementioned refinement was designed for improving performance. The original Isabelle abstract type *vec* is mapped to the Isabelle type *iarray* (the type itself is just a wrapper of lists), which is then mapped in the code generation process to the SML *Vector* structure; the SML structure requires constant time for access operations, improving, a priori, an implementation by lists. The code equations that perform the data type and operations conversions (from type *vec* to type *iarray*) can be found in file *Matrix_To_IArray* in [2]. As in our previous example, the data type refinement demands labeling the morphisms between the abstract type (*vec*) and the concrete one (*iarray*), and introducing operations on *iarrays* that are proven equivalent to the original abstract ones. These proofs are almost straightforward, since the *iarray* and *vec* representations share a *functional* flavor (in the way of accessing elements) that can be exploited in proofs.

Our Gauss-Jordan algorithm is implemented for matrices with entries over a *field*; in our execution experiments we carry out computations over the Isabelle types *real*, *rat* (for $\mathbb{Q}$) and *bit* (an implementation of the field $\mathbb{Z}/2\mathbb{Z}$); the Isabelle type *real* admits serialisations to an SML *ad hoc* type (quotients of SML *IntInf.int* elements) and also to the SML *Real.real* type. The former offers arbitrary precision, but on a standard machine, using the optimizer compiler MLton, only (randomly generated) matrices up to 100×100 size can be computed in a reasonable time (as a matter of comparison, Gauss-Jordan algorithm in *Mathematica*® over matrices of real numbers with arbitrary precision

becomes rather slow at sizes over 500×500). Table 1 shows the times used by the SML implementation Poly/ML and the optimizer compiler MLton to process and execute the Gauss-Jordan elimination algorithm generated from the Isabelle verified specification over (randomly generated) matrices whose inputs are quotients of *IntInf.int* elements. The following experiments have been carried out in a computer with an Intel Core i3-370M Processor (2 cores of 2.4 GHz) with 4GB of RAM and Ubuntu GNU/Linux 11.10. The SML code and the benchmark matrices are available from [2].

Rational matrices				
	Poly/ML		**MLton**	
Size (n)	**Processing Time (seconds)**	**Execution Time (seconds)**	**Processing Time (seconds)**	**Execution Time (seconds)**
10	0.0	0.0	0.2	0.0
20	0.0	0.2	0.3	0.0
40	0.1	3.7	0.9	1.5
60	0.2	22.7	1.9	9.6
80	0.5	77.0	3.5	32.7
100	0.7	200.9	6.0	84.1

Table 1. Elapsed time (in seconds) to process random $\mathbb{Q}^{n \times n}$ matrices (with elements between -10 and 10) and computing their rrefs using the Gauss-Jordan algorithm with Poly/ML 5.5 and MLton 20100608.

Applying profiling techniques, we detected that most of the computing time is used not in matrix operations but in the ones related to integer quotients operations (normalising quotients, computing the lcm of denominators, and the like[3]). The latter serialisation (to the SML *Real.real* type) is produced only for computing purposes, since it is inconsistent and suffers from numerical stability problems, but allows us to apply Gauss-Jordan elimination to (randomly generated) matrices up to size 700×700. The performance tests are presented in Table 2. The processing and execution times in Poly/ML follow a linear pattern with respect to the number of elements in the matrix (n^2).

The *rat* type is also serialised to quotients of *IntInf.int* pairs; the performance tests are therefore equal to the ones obtained for the first serialisation of type *real* and presented in Table 1.

Finally, we define our custom serialisation of type *bit* to SML; the Isabelle constants $0 :: bit$ and $1 :: bit$ are mapped in SML to 0 and 1 of type *IntInf.int*; operations over *bit* to arithmetic operations modulo 2 in *IntInf.int*. This serialisation proved empirically to perform better than other options such as the SML type *Bool*, or using *IntInf.int* with exhaustive definitions of the operations. The benchmarks of this serialisation are presented in Table 3.

[3] Both MLton and Poly/ML make use of the GMP `http://gmplib.org/` set of libraries for arithmetic.

Real matrices				
	Poly/ML		MLton	
Size (n)	Processing Time (seconds)	Execution Time (seconds)	Processing Time (seconds)	Execution Time (seconds)
10	0.0	0.0	0.8	0.0
20	0.0	0.0	2.5	0.0
40	0.1	0.0	13.8	0.0
60	0.2	0.0	56.9	0.0
80	0.3	0.0	164.3	0.0
100	0.6	0.2	361.6	0.1
200	3.7	0.7	9145.4	0.5
400	20.3	5.9	-	-
600	65.8	20.5	-	-
700	98.6	44.4	-	-

Table 2. Elapsed time (in seconds) to process random $\mathbb{R}^{n\times n}$ matrices (with elements between -10 and 10) and computing their rrefs using the Gauss-Jordan algorithm with Poly/ML 5.2 and MLton 20100608.

With this last serialisation and Poly/ML 5.5 we get to apply Gauss-Jordan elimination, and compute the rank, of matrices of dimensions up to 2560×2560; computing time grows linearly on the number of matrix entries (as seen in Table 3), and thus RAM memory becomes the only practical limitation. For instance, we are able to compute the rank of the binary matrix representing the following digital image (Fig. 1), captured with a *confocal* microscope from a neuronal culture. It is worth noting that processing and computing times over matrices obtained from digital images are smaller than the ones obtained over randomly generated matrices, since the first ones usually contain patterns which reduce the number of computations performed during the diagonalization.

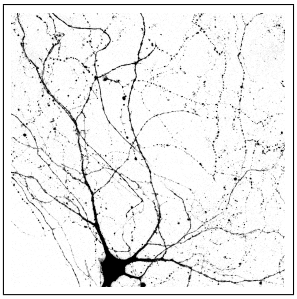

Fig. 1. Image (2048×2048 px.) of a neuron captured with a confocal microscope.

The rank of matrices with entries in $\mathbb{Z}_2$ permits to know the number of connected components (and can be successfully applied to the computation of the number of synapses in a neuron, automating a cumbersome task previously

$\mathbb{Z}_2$ matrices				
	Poly/ML		MLton	
Size (n)	Processing Time (seconds)	Execution Time (seconds)	Processing Time (seconds)	Execution Time (seconds)
50	0.0	0.0	0.8	0.0
100	0.3	0.0	4.0	0.1
200	1.0	0.3	54.6	0.6
400	4.6	2.9	809.2	5.2
600	10.6	9.8	-	-
800	19.8	24.1	-	-
1000	31.8	45.1	-	-
1200	53.7	79.7	-	-
1400	65.6	143.0	-	-
1600	107.0	200.5	-	-

Table 3. Elapsed time (in seconds) to process randomly generated $(\mathbb{Z}_2)^{n\times n}$ matrices and computing their corresponding rrefs using the Gauss-Jordan algorithm with Poly/ML 5.5 and MLton 20100608.

made "by hand" by biologists) in the original image. See [13] for details about this technique. Additional benchmarks and extensive details on the previous and some other tests are presented in [3].

6 Related work and Further work

6.1 Related work

From the different theorem provers available in the HOL family, the ones with a better mathematical library are HOL-Light and Isabelle; this can be checked by reading through their libraries, and corroborated by informal but informative rankings such as [24]; our work here relies on the foundations that both systems share and has reused successfully the mathematical machinery that has been developed there; nevertheless, and to the best of our knowledge, both of them lack of implementations of numerical Linear Algebra; moreover, we do not know of any attempt of execution of the definitions available in that libraries. From our point of view, our work is a starting point to fill a gap between formalization and execution that aims to a greater use of these already powerful libraries.

Some other theorem provers have also formalized the computation of the rank of linear forms; for instance, the SSReflect library of Coq contains the most extensive effort to formalize finite-dimensional Linear Algebra concepts, aiming at providing a suitable library for the implementation of the classification of finite simple groups. The whole library is based upon finite-dimensional structures, and Coq itself is a constructive setting in which proofs and algorithms are intertwined, so that one would (erroneously) expect that an implementation of Gauss-Jordan elimination over matrices should be executable; as is well

known [12, Sect. 4], the extensive use of dependent types features in the representation of algebraic structures and matrices, which allows for relatively simple proofs, comes at a cost: these definitions have been locked to avoid the heavy computations that they would demand, since they may not finish in a reasonable amount of time. In an effort to offer executability of some of the concepts in the SSReflect library, a new library CoqEAL [4] has been carried out in which, by means of types and algorithms refinements, computable versions of, for instance, the rank of a matrix, are provided.

6.2 Further work and Conclusions

We do not aim to present this development as a *canonical* approach to the the task of bringing together mathematical formalization and execution, but to show that proof assistants are mature enough to enable the simultaneous development of both fields with some technical effort (that once carried out, can be later reused in different settings). Additionally, one of the fields in which the Isabelle/HOL tool is more actively growing at the moment is data types and algorithms refinements, with the ambitious goal of reducing the gap between *software formalization* and *working software*.

The case study we have presented in this paper can be considered from at least two different points of view. First, as an experiment in Linear Algebra formalization, for which the HMA library has shown to be an adequate framework. With some technical effort in the code generation process, we have been capable of formalizing and executing the same "abstract" algorithm; in addition to this, we have developed tools (definitions and proofs over row and column elementary operations) that are applicable in the formalization of numerical Linear Algebra. Second, as an effort to get competitive results from a computational point of view; we have successfully applied some refinement techniques already available in Isabelle, obtaining formalized programs that can be executed over matrices of a remarkable size.

There are several directions we plan to take this work. Even if the performance of the Gauss-Jordan formalized algorithm is quite satisfying, some refinements could be thought of to reduce the number of operations that it performs; the algorithm could be implemented using *block matrices* that recursively decrease their size after each iteration of the algorithm. This would reduce the number of operations performed; on the other hand, it could demand the use of *dependent types* or *subtypes* to define submatrices (or some similar construct), falling short of the HOL type system.

Some other improvements of the algorithm are presented in the literature; for instance, instead of pivoting the first nonzero element over a given index of a column, the maximum element of the same column can be pivoted ("partial pivoting"), or even the maximum element in the whole *submatrix* ("total pivoting"); these strategies are experimentally known to improve the performance of the algorithm and specially its numerical stability. Instead of improving the performance of the Gauss-Jordan elimination algorithm, an *ad hoc* algorithm

computing the rank of matrices could be implemented, and *linked* by a standard refinement technique with rank computation by Gauss-Jordan elimination.

There are further refinement techniques in Isabelle that we would like to explore as a natural continuation to our work. The work in [8] presents an infrastructure for *lifting* definitions from a *concrete* data type to an abstract one, and for *transferring* proofs from the abstract setting to the concrete one. The concept is really close to the one we have proposed in this paper, but at the moment the technology can be applied to Isabelle user defined types (as abstract type) and its underlying concrete types or quotient types. In our setting, it could have been used to lift definitions from *functions* to the type *vec*; it is also used in the code generation process of some of the fields that we used as examples. Another interesting Isabelle tool that we would like to explore is *Autoref* [18]; according to the authors, the tool automatically refines algorithms over abstract concepts to algorithms over concrete implementations; even if our underlying algebraic structures (vectors or matrices) are not completely "abstract", it could be interesting to explore the feasibility of writing down Linear Algebra algorithms in Isabelle in an imperative way (as they are usually presented in textbooks) and rely on the automatic refinement to translate these algorithms to executable ones in a functional programming setting, very much in the spirit of [17]. The previous tools and techniques could be applied to a wide range of Linear Algebra algorithms, some of them rooted in variants of Gauss-Jordan elimination.

Acknowledgements. This work has been supported by projects MTM2009-13842-C02-01 (Ministerio de Educación y Ciencia), FORMATH, nr. 243847, of the FET program within the FP7 of the European Commission, and Universidad de La Rioja, research grant FPI-UR-12.

Andreas Lochbihler provided us with great insight and invaluable ideas in how to get the right setup for code generation of sets, and also in understanding the type classes computing cardinality of types. Florian Haftmann helped us with the serialisation of the *real* Isabelle type to the *Real* SML structure. Johannes Hölzl assisted us in polishing our formalization of the Rank Nullity theorem. Julio Rubio suggested the use of *profiling* techniques to detect weaknesses in the execution experiments and commented on earlier versions of the paper. The authors also wish to thank the referees because of their valuable comments on the first version of this paper.

References

1. J. Aransay and J. Divasón. Rank Nullity Theorem in Linear Algebra, Archive of Formal Proofs. 2013. `http://afp.sourceforge.net/entries/Rank_Nullity_Theorem.shtml`.
2. J. Aransay and J. Divasón. Gauss-Jordan elimination in Isabelle/HOL. 2013. `http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan/`.
3. J. Aransay and J. Divasón. Performance Analysis of a Verified Linear Algebra program in SML, Taller de Programación Funcional (TPF 2013). Accepted for pub-

lication. Preprint available from http://wiki.portal.chalmers.se/cse/uploads/ForMath/pavlap.
4. M. Dénès, A. Mörtberg and V. Siles. A refinement-based approach to computational algebra in COQ. Interactive Theorem Proving (ITP 2012). pp. 83 – 98. LNCS, 2012.
5. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf and J. G. Smaus. A Fully Verified Executable LTL Model Checker. Computer Aided Verification (CAV 2013). pp. 463 – 478. LNCS, 2013.
6. Formath Project: Formalisation of Mathematics. http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath.
7. F. Haftmann and T. Nipkow. Code Generation via Higher-Order Rewrite Systems. Functional and Logic Programming (FLOPS 2010). pp. 103 – 117. LNCS, 2010.
8. F. Haftmann, A. Krauss, O. Kunčar and T. Nipkow. Data Refinement in Isabelle/HOL. Interactive Theorem Proving (ITP 2013). pp. 100 – 115. LNCS, 2013.
9. J. Harrison. A HOL Theory of Euclidean Space. TPHOLs 2005. pp. 114 – 129. LNCS, 2005.
10. J. Harrison. The HOL Light Theory of Euclidean Space. J. Autom. Reasoning, *50* (2). pp. 173 – 190. 2013.
11. T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua and R. Zumkeller. A revision of the Proof of the Kepler Conjecture. Discrete & Computational Geometry, *44* (1). pp. 1 – 34. 2010.
12. J. Heras, T. Coquand, A. Mörtberg and V. Siles, Computing Persistent Homology within Coq/SSReflect. ACM Transactions on Computational Logic. Accepted for publication. Preprint available from http://www.cse.chalmers.se/~mortberg/papers/cphwcs.pdf.
13. J. Heras, M. Dénès, G. Mata, A. Mörtberg, M. Poza and V. Siles. Towards a certified computation of homology groups for digital images. Computational Topology in Image Context (CTIC 2012). pp. 49 – 57. LNCS, 2012.
14. J. Heras, M. Poza, M. Dénès and L. Rideau. Incidence Simplicial Matrices Formalized in Coq/SSReflect. Conference on Intelligent Computer Mathematics (CICM 2011). pp. 30 – 44. LNCS, 2011.
15. J. Hölzl *et al*, HOL Multivariate Analysis, http://isabelle.in.tum.de/dist/library/HOL/HOL-Multivariate_Analysis/index.html, 2013.
16. J. Hölzl, F. Immler and B. Huffman. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. Interactive Theorem Proving (ITP 2013). pp. 279 – 294. LNCS, 2013.
17. P. Lammich and T. Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. Interactive Theorem Proving (ITP 2012). pp. 166–182. LNCS, 2012.
18. P. Lammich. Automatic Data Refinement. Interactive Theorem Proving (ITP 2013). pp. 84 – 99. LNCS, 2013.
19. A. Lochbihler. Light-weight containers for Isabelle: efficient, extensible, nestable. Interactive Theorem Proving (ITP 2013). pp. 116 – 132. LNCS, 2013
20. T. Nipkow. Gauss-Jordan Elimination for Matrices Represented as Functions. Archive of Formal Proofs, 2011. http://afp.sourceforge.net/entries/Gauss-Jordan-Elim-Fun.shtml.
21. T. Nipkow, L. Paulson and M. Wenzel. Isabelle/HOL: A proof assistant for Higher-Order Logic. Springer, 2002.
22. S. Roman. Advanced Linear Algebra (Third Edition). Springer. 2008
23. E. Gasner and J. H. Reppy (eds.) The Standard ML Basis Library, http://www.standardml.org/Basis/.
24. F. Wiedijk. Formalizing 100 Theorems. http://www.cs.ru.nl/~freek/100/.