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 have enriched the proof with an additional formalization of its *computational* meaning; to this purpose, we chose 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 had to introduce to get code execution from the original representation, and also on the performance of the code obtained. We also present an alternative verified type refinement of vectors and matrices 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 Linear Algebra algorithms in HOL theorem 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* wrt 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 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. Interestingly, 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. Nevertheless, and to the best of our knowledge, this idea has not been explored before, and requires additional features that we present here. 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 divided 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.

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 a greater number of tools (code generation, automatic proof procedures) are available. We do not aim at presenting 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 notation is particular of the code generation setting, 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* for such 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 ($x :: \alpha ::$ *field*) imposes that the type variable α poses the structure and properties stated in the *field* type class, and can be later replaced exclusively by types which are instances of such 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 greatly inspired by the impressive work of J. Harrison in HOL-Light [10], which includes proofs of intricate theorems (such as the Stone-Weierstrass theorem) and that has also been successfully used as a basis for the Flyspeck project [11], aiming at a formally verified 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 α :: *field*) taking into account that the HOL type system lacks 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 β :: *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' indexes 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 performed 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).

In the HMA library already appear some definitions 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 wrt 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). 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 neither in the Isabelle HMA library, nor in the HOL-Light one, is the executability of the previous data types and operations. As we will see in Section 4, the *finite* type class falls short for enabling executability of some operations over vectors and matrices, 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 at showing that the HMA

library provides a framework where algorithms over matrices can be formalized, executed and coupled with its 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 also admits iterative refinements, as long as their equivalence is always proved. A detailed explanation of the methodology can be 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 thought for simplicity. The second one uses immutable arrays (represented in the Isabelle type *iarray*) and presents a remarkable performance when generated to SML.

2 The Rank Nullity theorem of Linear Algebra

The Rank Nullity theorem is a well-known result in Linear Algebra; it states that the dimension of a finite-dimensional vector space V is equal to the sum of the dimensions of the range of f and its kernel, being f any linear form from V to a vector space W. 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 Isabelle statement of the result is as follows:

```
theorem rank_nullity_theorem:
  assumes "linear (f::(α::{euclidean_space}) => (β::{real_vector}))"
  shows "DIM (α) = 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 such 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 referred to the carrier set of that type.

There is one remarkable result that we didn't found in any textbook, but that proved crucial in the formalization of the theorem. Its Isabelle statement reads as follows:

```
lemma inj_on_extended:
  assumes lf: "linear f"  and f: "finite C"
  and ind_C: "independent C"  and C_eq: "C = B ∪ W"
  and disj_set: "B ∩ W = {}" and span_B: "{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 $\{f e_{m+1} \dots f e_n\}$ (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 direct; we make use of a result in the HMA library (labeled as *matrix-works*) which states that, given a 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 wrt the Isabelle statement of the Rank Nullity theorem; where the source and target vector spaces were, respectively, an Euclidean space and a real vector space (of any dimension), they are now replaced by a $real\hat{}n\hat{}m$ matrix, *i.e.*, the vector spaces $real\hat{}n$ and $real\hat{}m$.

```
lemma rank_nullity_theorem_matrices:
  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*, aka 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 of the range space. The previous formalization [1] is part of the Isabelle repository; thanks to the infrastructure in the HMA library, it summed up only nine pages of Isabelle sources.

[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.

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 chose the Gauss-Jordan elimination method. The main reason is that this has several different applications; it can be used to solve systems of linear equations (Nipkow [20] verified the Gauss-Jordan elimination algorithm to this aim; the version used in that work is very succinct, but works exclusively for input square matrices with unique solution); Gauss-Jordan elimination also performs quite well in the computation of inverse matrices and can be used in the computation of determinants. The algorithm may not be optimal for any of those problems (indeed, it is not), 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 (obtained from [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).

One way to achieve the collection of elementary row operations that reach 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, thought to ease the formalization. In it, the algorithm is described by means of exclusively *elementary row operations*, E_i 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 indexes of rows and columns are represented by *finite* types; the type of columns indexes needs to be *traversed*, and thus it is restricted to be an instance of the *enum* type class; this type class is part of the Isabelle library, and represents types which carrier set is *explicit*.

[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 algorithmic and its mathematical meaning are often presented as different subjects.

The algorithm is defined by the following simple steps; let A be a matrix; let l be an index which stores the row where the pivot (the leading entry) should be placed in the column j (the initial value of l is 0); let $A\,i$ denote the row i of the matrix; let us start from column $j = 0$:

1. check that column j has a nonzero element from row l onwards; if it hasn't, skip to step 1 with A, index l and column $j+1$; if it has (let us name it $a_{i,j}$) skip to step 2;
2. the element $a_{i,j}$ is *pivoted* to the position $a_{l,j}$, by switching rows i and l;
3. $A\,l$ is multiplied by the inverse of the pivoted element, now $a_{l,j}$;
4. the rest of the rows (for instance, k) are applied the elementary row operation $A\,k - (a_{k,j} * A\,l)$;
5. skip to step 1, with the computed matrix, index $l+1$ and column $j+1$.

The algorithm is terminating, since the columns' indexes are elements of a *finite* (and *enumerable*) type. Additionally, it satisfies different properties. For instance, 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 premises in the types indexing columns (and rows); it must have some notion of order, since the proofs will be performed by induction over columns' indexes; we made 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. In particular, a representation of numeral types in the Isabelle library (represented by the *bit0* and *bit1* type constructors over finite types) which we will use later for representing concrete matrices of a given dimension is instance of this type class.

Additionally, the algorithm performs exclusively elementary row operations. 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 invertible_matrix_mult_left_rank':
  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 be also reused for different algorithms in numerical Linear Algebra. 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 types indexing rows and columns can be code generated; we present in Section 4 the details of such process.

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), which underlying *concrete* type is 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 their universe, and some arithmetical possibilities.

The *finite* type class is enough to generate code for 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 equal to the type of rows in B), that a row consists exclusively of zeros (traversing its indexing type), algorithms traversing the universe of the rows or columns indexing types, and also operations that involve logical operators ($\forall$, $\exists$) or the Hilbert's ϵ operator, such as "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 thought for working 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 $\mathit{coset}\,\emptyset$ 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 instance of the type classes *finite*, *enum* and *mod-type* (indeed, *mod-type* can be proved to be a subclass of *enum*, but we preferred to keep them both since they served in our work for different purposes). 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*). 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 $\{0, 1, 2, 3, 4 :: \mathit{int}\}$. The integers as underlying type allow to reuse (with adequate modifications) integer division in the resulting finite types. As part of our development, we had to prove that the *num1*, *bit0* and *bit1* type constructors were instances of the *enum* type class.

The Isabelle library already provides basic arithmetic functions for the numeral types, with definitions of addition, substraction, multiplication and division. Note that, for these operations to be defined generally for every cardinal, the cardinal 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 cardinal is *computable*; we proved that the previous numeral types were instances of such class, therefore 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 to allow execution of the vectors indexing types. Nevertheless, as we mentioned in our presentation of the *vec* data type, this 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 in the index)*. We were aware of its not very encouraging performance, but the closeness between both representations greatly simplifies the process of *formalizing* the refinement; 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 refinement of 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. 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). 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. As dealing with matrices as functions can become rather cumbersome, we also defined 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 of 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 lengthy (various minutes).

The following refinement was thought for improving performance. The Isabelle type *iarray* (the type itself is just a wrapper of lists) is code generated to the SML *Vector* structure; the SML structure allows constant time for access operations, improving, a priori, an implementation by lists. The code equations that perform the data type and operations conversions 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 were 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 was implemented for matrices with entries over a *field*; in our execution experiments we carried out computations over the Isabelle types *real*, *rat* (for $\mathbb{Q}$) and *bit* (an implementation of $\mathbb{Z}/2\mathbb{Z}$); 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 (Gauss-Jordan algorithm in *Mathematica*® becomes rather lengthy at sizes over 500×500). Applying profiling techniques, we discovered that most of the computing time was used not in matrix operations but in the ones related to quotients operations (normalising quotients and the like[3]). The latter serialisation 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 *rat* type is also serialised to quotients of *IntInf.int* pairs; performance is equal to the one obtained for the first serialisation of type *real*. Finally, we created our custom serialisation of type *bit*; the constants $0 :: bit$ and $1 :: bit$ are mapped to 0 and 1 in *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.

With this last serialisation and Poly/ML 5.5 we got to apply Gauss-Jordan elimination, and compute the rank, of matrices of dimensions up to 2048×2048 size; computing time grows linearly on the number of matrix entries, and therefore RAM memory becomes the only limitation. For instance, we were able

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

to compute the rank of the binary matrix representing the following digital image (Fig. 1), captured with a *confocal* microscope from a neuronal culture. The rank of such matrices permits us to compute 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 made "by hand" by biologists). See [13] for details about this technique. Additional benchmarks

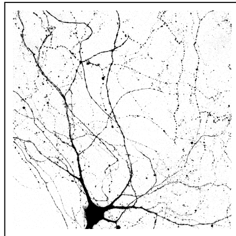

Fig. 1. Image (2048×2048 px.) of a neuron captured with a confocal microscope.

and extensive details on the previous and some other tests can be found in a separate report [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 easily 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 could aim at 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 it is well known [12, Sect. 4], the extensive use of dependent types' features in the

representation of algebraic structures and matrices, which gives as a product the simplicity of the 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.

The previous reflection supports our claim that finite functions as a working type for executing matrices are not a good choice; 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 matrix, are provided.

6.2 Further work and Conclusions

We do not aim at presenting the previous development as a *canonical* approach to the the task of bringing together mathematical formalization and execution, but at showing that proving 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 a really 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.

Different research lines stay ahead of us. 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 artifact), 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 of a given 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 almost 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 on how to to get the right setup for the code generation of sets, and also in our understanding of the type classes computing cardinality of types. Florian Haftmann helped us with the serialisation of the *real* Isabelle type to the SML *Real* structure. Johannes Hölzl assisted us in polishing our formalization of the Rank Nullity theorem. Julio Rubio suggested the use of *profiling* techniques to detect potential weaknesses in the execution experiments and commented on earlier versions of the 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, *Technical Report*, available at `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. Proceedings of CAV 2013 (Computer Aided Verification). Accepted for publication. Preprint available from `http://www21.in.tum.de/~nipkow/pubs/cav13.pdf`
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 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). Accepted for publication. Preprint available at `http://isabelle.in.tum.de/~haftmann/pdf/data_refinement_haftmann_kuncar_krauss_nipkow.pdf`.
9. J. Harrison. A HOL Theory of Euclidean Space. TPHOLs 2005. pp. 114 – 129.
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 certi fied computation of homology groups for digital images. CTIC 2012. pp. 49 – 57. LNCS, 2012.
14. J. Heras, M. Poza, M. Dénès and L. Rideau. Incidence Simplicial Matrices Formalized in Coq/SSReflect. ICM 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. ITP 2013. LNCS, 2013.
17. P. Lammich and T. Tuerk. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. ITP 2012, pp. 166–182. LNCS, 2012.
18. P. Lammich. Automatic Data Refinement, Interactive Theorem Proving (ITP 2013). Accepted for publication. Springer. Preprint available at `https://www21.in.tum.de/~lammich/pub/autoref.pdf`.
19. A. Lochbihler. Light-weight containers for Isabelle: efficient, extensible, nestable, Interactive Theorem Proving (ITP 2013). Accepted for publication. Springer. Preprint available at `http://pp.info.uni-karlsruhe.de/uploads/publikationen/lochbihler13itp.pdf`.
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/`.