JFP **25**, e9, 21 pages, 2015.
doi:10.1017/S0956796815000155

Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm

JESÚS ARANSAY and JOSE DIVASÓN

Departamento de Matemáticas y Computación, Universidad de La Rioja, c/ Luis de Ulloa s/n, La Rioja, 26004, Spain
(*e-mail:* http://www.unirioja.es/cu/jearansa; http://www.unirioja.es/cu/jodivaso)

Abstract

In this paper, we present a formalisation in a proof assistant, Isabelle/HOL, of a *naive* version of the Gauss-Jordan algorithm, with explicit proofs of some of its applications; and, additionally, a process to obtain versions of this algorithm in two different functional languages (SML and Haskell) by means of code generation techniques from the verified algorithm. The aim of this research is not to compete with specialised numerical implementations of Gauss-like algorithms, but to show that formal proofs in this area can be used to generate usable functional programs. The obtained programs show compelling performance in comparison to some other verified and functional versions, and accomplish some challenging tasks, such as the computation of determinants of matrices of *big* integers and the computation of the homology of matrices representing digital images.

1 Introduction

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

Fortunately, after years of continuous work, theorem proving tools have reduced this well-known gap, and the technology they offer is being used to implement and formalise state of the art algorithms and generate programs to, usually, functional languages, with remarkable performance (see, for instance, Esparza *et al.* (2013), Sternagel (2013)). Code generation is a well-known and admitted technique in the field of Formal Methods. Avigad and Harrison in a recent survey about formally verified Mathematics (Avigad & Harrison, 2014), enumerate three different strategies to verify "mathematical results obtained through extensive computations"; the third

one is presented as "to describe the algorithm within the language of the proof checker, then extract code and run it independently".

In this work, following the third strategy quoted above, we present an experiment to formalise a version of the Gauss-Jordan algorithm (Strang, 2009) over matrices in the theorem prover Isabelle/HOL (Nipkow *et al.*, 2002). The algorithm computes the reduced row echelon form of a matrix, which is then proved to be applicable to solve standard problems in Linear Algebra, such as computing the rank of matrices, computing determinants and inverses, solving systems of linear equations, and computing bases of fundamental subspaces of matrices. These verified algorithms are later code-generated to the functional languages SML (Milner *et al.*, 1997) and Haskell (2014). The algorithm that we implement is neither specialised, nor obtained from a Computer Algebra system, but is just a simple version of the Gauss-Jordan algorithm designed to ease its formalisation. Nevertheless, the utility of our work is threefold.

First, it sets a framework in which formalisation of Linear Algebra results, implementation of algorithms, and code generation to functional programs is achieved with an affordable effort, by using well-established tools. Second, it shows that the performance of the generated code is enough (even if it is not comparable to specialised programs in Computer Algebra) to obtain interesting computations (such as determinants with big integers that disclosed a bug in Mathematica® (Durán *et al.*, 2014) and relevant properties of digital images). Finally, it shows that the ties between matrix algorithmics and Linear Algebra can be established thanks to the Isabelle/HOL Multivariate Analysis Library (HMA, 2014) infrastructure (a property that is harder to find in Computer Algebra systems).

The paper will be divided as follows: in Section 2, we introduce the Isabelle/HOL theorem prover and the infrastructure in such a system that is used in our work; we distinguish among the parts which are already in the system, and the new components that are product of our own work. In Section 3, we present a version of the Gauss-Jordan algorithm over fields, as well as the different applications of it that we have formalised in Isabelle/HOL. In Section 4, we present the code generation process from the formalised Isabelle algorithm to the running versions in SML and Haskell. In Section 5, we introduce some case studies in which the generated algorithms show their usefulness, and some relevant related work. The source files of the development are available at Aransay & Divasón (2014c); they have been developed under the Isabelle 2013-2 version. The previous website also includes the SML and Haskell code generated from the Isabelle specifications, an implementation of the algorithm in C++, which performance is compared to ours, and some input matrices that have been used for profiling and benchmarking. Finally, in Section 6, we draw some conclusions and possible research lines that follow from our work.

2 Isabelle

2.1 Isabelle/HOL

Isabelle (Paulson, 1990) is a generic theorem prover which has been instantiated to support different object-logics, from which higher-order logic (or briefly, *HOL*

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

(Nipkow *et al.*, 2002)) is the one that offers a greatest number of facilities to the user, some of which will be relevant to our work (such as code generation, see Section 2.3). The HOL type system is based on non-empty types, function types ($\Rightarrow$) and type constructors of different arities ($\_\ list$, $\_ \times \_$) that can be applied to already existing types (*nat, bool*) and type variables (α, β). Types can be also introduced by enumeration (*bool*) or by induction, as lists (by means of the *datatype* command). Additionally, new types can be also defined as non-empty subsets of already existing types (α) by means of the *typedef* command; the command takes a set defined by comprehension over a given type $\{x :: \alpha \mid P\ x\}$, and defines a new type σ.

Isabelle 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 HOL Multivariate Analysis Library has a type class *field* representing the algebraic structure. Concrete types (*real*, *rat*) can be proven to be *instances* of a given type class (*field* in our example). Type classes are also used to impose additional restrictions over type variables; for instance, the expression ($x :: \alpha :: field$) imposes the constraint that the type variable α possesses the structure and properties stated in the *field* type class, and can be later replaced exclusively by types which are instances of that type class.

2.2 HOL multivariate analysis

The HOL Multivariate Analysis (or *HMA* for short) Library (HMA, 2014) is a set of Isabelle theories which contains theoretical results in mathematical fields such as Analysis, Topology and Linear Algebra. They are based on the impressive work of Harrison in HOL Light (Harrison, 2013), which includes proofs of intricate theorems (such as the Stone-Weierstrass theorem) and has been used as a basis for appealing projects such as the formalisation of the proof of the Kepler conjecture by Hales (Hales & Ferguson, 2011). 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 $\alpha :: field$).

The idea (first presented by Harrison (2005)) is to represent n-dimensional vectors (type *vec*) over α by means of *functions* from a finite type variable $\beta :: finite$ to α, where $card(\beta) = n$ (the cardinal of a type can be interpreted as an abuse of notation; it really stands for the cardinal of the universe set of such a type). For proving purposes, this type definition is usually sufficient to support the generic structure $\mathbb{F}^n$.

The Isabelle *vec* 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 (additional restrictions over α and β are added only when required for formalisation purposes):

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

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

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 from the formalisation point of view, as already pointed out by Harrison. For instance, the type system can be used to guarantee that operations on vectors (such as addition and multiplication) are only performed over vectors of equal dimension, i.e., vectors whose indexing types are exactly the same (this would not be the case if we were to use, for instance, lists as vectors). Moreover, the functional flavour of operations and properties over vectors is kept (for instance, vector addition can be defined in a pointwise manner).

The representation of matrices is then derived in a natural way based on the representation of vectors by iterating the previous construction (matrices over a type α will be terms of type $\alpha\hat{}m\hat{}n$, where m and n stand for finite type variables).

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

Our work also enhances the HMA Library by *generalising* some of the definitions and mathematical results presented there over type classes representing *real vector spaces* and *Euclidean spaces* (such as for instance matrices, determinants and associated lemmas) to more general type classes and locales representing *vector spaces* and *finite-dimensional vector spaces* over a generic field $\mathbb{F}$ (of any characteristic). This limitation had been pointed out in some previous developments over this Library (see, for instance, Avigad *et al.* (2014)). This will enable us to carry out computations over any field, such as $\mathbb{F}_p$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$. This generalisation is thoroughly described in Aransay & Divasón (2015). It has involved the introduction of new structures (locales and type classes), the reproduction of some previous definitions and proofs in those structures, and also in some particular cases the formalisation of different proofs for results that hold both in a generic field $\mathbb{F}$ and in $\mathbb{R}$, but whose previous proofs in $\mathbb{R}$ made use of specific properties of such a structure. We present in Sections 4 and 5 further details involving computations over $\mathbb{Z}_2$, $\mathbb{Q}$ and $\mathbb{R}$ matrices.

2.3 Code generation

Another interesting feature of Isabelle/HOL is its code generation facility (Haftmann, 2013). Its starting point are specifications (in the form of the different kinds of definitions supported by the system) whose properties can be stated and proved, and (formalised) rewriting rules that express properties from the original specifications. From the previous *code equations*, a *shallow embedding* from Isabelle/HOL to an abstract intermediate functional language (Mini-Haskell) is performed. Finally, trivial transformations to the functional languages SML, Haskell, Scala and OCaml are performed. The expressiveness of HOL (such as for instance universal and existential quantifiers and the Hilbert's ϵ operator) is not that of functional programming languages, and therefore one must restrict herself to use Isabelle

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

"executable" specifications, if she aims at generating code from them (or prove *code equations* that refine non-executable specifications to executable ones).

One weakness of this methodology is the different semantics among the source Isabelle constructs and their functional languages counterparts; this gap can be narrowed to a minimum, since the tool is based on a *partial correctness* principle. This means that whenever an expression v is evaluated to some term t, $t = v$ is derivable in the equational semantics of the intermediate language, see Haftmann & Nipkow (2010) for further details. Then, from the intermediate language, the code generation process can proceed to the functional languages by means of the aforementioned trivial transformations, or, in a broadly accepted way of working (Esparza *et al*., 2013; Haftmann *et al*., 2013; Avigad & Harrison, 2014), *ad-hoc* serialisations to types and operations in the functional languages library can be performed. These serialisations need to be *trusted*, and, therefore, they are kept as simple as possible (in Section 4.1 we explicitly introduce these transformations).

3 The Gauss-Jordan algorithm and its applications

In a previous work (Aransay & Divasón, 2014b), we formalised the Rank Nullity Theorem of Linear Algebra. In our formalisation, it is established that, given V a finite-dimensional vector space over $\mathbb{R}$, W a vector space over $\mathbb{R}$, and $\tau \in L(V, W)$ (a linear map between V and W), $\dim(\ker(\tau)) + \dim(\text{im } (\tau)) = \dim(V)$ or, in a different notation, $\text{null } (\tau) + \text{rk } (\tau) = \dim(V)$. We closely followed the proof in Gockenbach (2010), as we follow here its notation. In our new development (Aransay & Divasón, 2014c), the previous result has been generalised replacing $\mathbb{R}$ by a generic field $\mathbb{F}$ of any characteristic. Unfortunately, having formalised the previous result does not provide us with an algorithm that computes the dimension of the image and kernel sets of a given linear map.

As it has been formalised in Aransay & Divasón (2014c), every linear map between finite-dimensional vector spaces over a field $\mathbb{F}$ is equivalent to a matrix $\mathbb{F}^{m\times n}$ (once a pair of bases have been fixed in both $\mathbb{F}^n$ and $\mathbb{F}^m$), and, therefore, we can reduce the computation of the dimensions of the range (or rank) and the kernel (or nullity) of a linear map to the computation of the *reduced row echelon form* (Roman, 2008) (or *rref*) of a matrix; the number of nonzero rows of such a matrix provides its rank, and the number of zero rows its nullity. The Gauss-Jordan algorithm computes the *rref* of a matrix.

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

Algorithm 1 traverses the columns of the input matrix, finding in each column k an element in the ith row (the first nonzero element in a row greater than the index l); if such an element (the *pivot*) exists, rows i and l are interchanged (if the matrix has maximum rank, l will be equal to the column index, otherwise it will be smaller), and the lth row is multiplied by the inverse of the pivoted element; this row is then

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

Algorithm 1 Gauss-Jordan elimination algorithm

```
1: Input: A, a matrix in 𝔽^{m×n};
2: Output: The rref of the matrix A
3: l ← 0;                                   ▷ l is the index where the pivot is placed
4: for k ← 0 to (ncols A) − 1 do
5:     if nonzero l (col k A) then          ▷ Check that col. k has a pivot over pos. l
6:         i ← index-nonzero l (col k A)    ▷ Let i be the such entry
7:         A ← interchange-rows A i l       ▷ Rows i and l are interchanged
8:         A l ← mult-row A l (1/A l k)     ▷ Row l is multiplied by (1/A l k)
9:         for t ← 0 to (nrows A) − 1 do
10:            if t ≠ l then                ▷ Row t is added row l times (−A t k)
11:                A t ← row-add A t l (− A t k)
12:            end if
13:        end for
14:        l ← l + 1
15:    end if
16: end for
```

used to perform row operations to reduce all remaining coefficients in column k to 0. If a column does not contain a pivot, the algorithm processes the next column. The algorithm performs exclusively *elementary row operations.*

In our Isabelle specification, rows and columns are assigned finite enumerable (in the sense that they admit an extensional definition) types, over which matrices are represented as functions. We have replaced the operations in the previous algorithm that produce side effects (for instance, *interchange-rows*, *mult-row*, *row-add*) by Isabelle *functions* whose outputs are matrices. We have replaced the first (and outer) *for* loop ranging along the columns indexes in the original algorithm by means of a *fold* operation of a list with the desired indexes. The second (and inner) *for* loop, whose range is the rows of a given column, is replaced by means of a *lambda expression* over the rows' type (indeed, each row is itself a function over this finite type).

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

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

```
Gauss-Jordan-in-pos A l k =
 (let
   i = (LEAST n. A $ n $ k ≠ 0 ∧ n ≥ l);
```

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

```
  interchange-A = (interchange-rows A i l);
  A' = mult-row interchange-A l (1/interchange-A $ l $ k)
   in
 vec-lambda (λt. if t = l then A' $ l
    else (row-add A' t l (-(interchange-A $ t $ k)))$ t))
```

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

```
Gauss-Jordan-column (i', A) k' =
 (let
  i = from-nat i';
  k = from-nat k'
 in
  if (∀m≥i. A $ m $ k = 0) ∨ (i = nrows A) then (i, A)
  else (i+1, (Gauss-Jordan-in-pos A i k)))
```

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

```
Gauss-Jordan-upt A k = snd (foldl Gauss-Jordan-column (0,A) [0..<Suc k])

Gauss-Jordan A = Gauss-Jordan-upt A ((ncols A) - 1)
```

The algorithm admits several variants, both to speed up its performance and also to avoid numerical stability issues with floating point numbers (Gockenbach (2010), Chap. 9), but in order to reduce the complexity of its formalisation we chose the above presentation.

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

- Computation of the inverse of a matrix, by "storing" the elementary row operations over the identity matrix.
- Determinants, taking into account that some of the elementary row operations can introduce multiplicative constants.
- Computation of bases and dimensions of the null (defined as $\{x \in \mathbb{F}^m \mid A \cdot x = 0\}$), left null ($\{x \in \mathbb{F}^n \mid x^T \cdot A = 0\}$), column ($\{A \cdot x \mid x \in \mathbb{F}^m\}$) and row ($\{A^T \cdot x \mid x \in \mathbb{F}^n\}$) subspaces (also known as fundamental subspaces) of a matrix.
- Solution of systems of linear equations ($\{x \in \mathbb{F}^m \mid A \cdot x = b\}$), both consistent (with unique or multiple solutions) and inconsistent ones.

The formalisation of the Gauss-Jordan algorithm, together with numerous previous results in Linear Algebra, and the different applications that are presented above, summed up *ca.* 12 000 lines of code (see Table 4); the proofs check that the defined objects (determinant, inverse matrix, solution of the linear system, fundamental

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

subspaces) are preserved (or modified in a certain way) after each algorithm step (and more concretely, after each elementary row operation). By using product types, we store an initial value for the desired object, and the input matrix. In the case of determinants, the initial pair is $(1, A)$.

```
Gauss-Jordan-upt-k-det-P A k =
 (let step = foldl Gauss-Jordan-column-k-det-P (1,0,A) [0..<Suc k]
   in (fst step, snd (snd step))))

Gauss-Jordan-det-P A =
 Gauss-Jordan-upt-k-det-P A ((ncols A) - 1)
```

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

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

```
lemma det-Gauss-Jordan-det-P:
 fixes A::'a::field^'n^'n::mod-type
 defines comp == Gauss-Jordan-det-P A
 shows (fst comp) * det A = det (snd comp)
```

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

```
definition basis-null-space A =
  {row i (P-Gauss-Jordan (transpose A))  |i. to-nat i ≥ rank A}
definition basis-row-space A =
  {row i (Gauss-Jordan A)  |i. row i (Gauss-Jordan A) ≠ 0}
definition basis-col-space A = {row i (Gauss-Jordan (transpose A))
  |i. row i (Gauss-Jordan (transpose A)) ≠ 0}
```

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

```
definition basis-left-null-space A =
 {row i (P-Gauss-Jordan A) | i. to-nat i ⩾ rank A}
```

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

In order to consider inconsistent systems suitably, we have represented the solutions as elements of the Isabelle option type (whose elements are of the form *(Some x)* and *None*); the set of solutions of a system will be presented as a singular point (whenever the system has solution), and the corresponding vectors forming a basis of the null space (or the empty set). As an additional result, we have formalised in Isabelle that every solution to a given system is of the previous form. Previous formalisations of the solution of systems of linear equations through the Gauss-Jordan algorithm (see, for instance, Nipkow (2011)) and most Computer Algebra systems compute exclusively single solutions (even more, for exclusively compatible systems with equal number of equations and unknowns).

In Section 1, we pointed out that the HMA library permitted us to keep the tie between Linear Algebra and algorithmics. The crucial result to this aim consists in establishing and formalising a link between *linear maps* and *matrices*.

The following result states that applying a linear map f to an element x of its source vector space is equal to multiply the associated matrix to f by the vector x (the matrix represents the linear map with respect to some previously fixed bases). The generalisation of this result to fields was formalised as a part of this development, inspired by the result over *real vector spaces*, already available in HMA. This result is available in the file *Generalizations.thy* in Aransay & Divasón (2014c)).

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

Then, in file *Linear-Maps.thy*, several results have been proven based on this one, that relate linear maps and properties of their associated matrices. For instance, the following result proves that a linear map is bijective if and only if its associated matrix has full column rank.

```
lemma linear-bij-rank-eq-ncols:
  fixes f::'a::field^'n::mod-type ⇒ 'a^'n
  assumes linear (op *s) (op *s) f
  shows bij f ⟷ rank (matrix f) = ncols (matrix f)
```

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

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

4 Code generation to functional languages

The presented Isabelle formalisation of the Gauss-Jordan algorithm can be directly executed *inside* of Isabelle (by rewriting specifications and code equations) with some setup modifications that we presented in a previous work (see Section 4 in Aransay & Divasón (2014a)) obtaining, thus, *formalised computations*. Every operation performed in Isabelle is itself executable, since we are dealing with enumerable types for representing matrices columns and rows. For instance, the Isabelle function *Gauss-Jordan-in-pos* presented previously, makes use of the *LEAST* operator (based itself on the Hilbert's ϵ operator and not executable in general), and is specified in our particular setting taking advantage of the fact that its underlying type is enumerable, and, therefore, can be executed to select a pivot.

Unfortunately, the performance obtained makes the algorithm unusable in practice, except for testing purposes. Matrices represented as functions over finite domains are reportedly impractical. More concretely, there are two sources of inefficiency in the results obtained. First, Isabelle is not designed as a programming language, and execution inside of the system offers poor performance. In Section 4.1, we present a solution to *translate* our specifications to functional programming languages. Second, the data structures (functions) that helped us to prove the correctness of the Gauss-Jordan algorithm and its applications are optimal for formalisation, but not for execution. Section 4.2 describes a *verified refinement* between the type used for representing matrices in our formalisation (*vec* and its iterated construction) and *immutable arrays*, a common data structure in functional programming.

4.1 Code generation and serialisations

Since the Isabelle code is not suitable for computing purposes, the original Isabelle specifications are *translated* to a programming (functional) language, as introduced in Section 2.3. Our choices (from the available languages in the standard Isabelle code generation setup) were SML (since the SML Standard Library includes a *Vector* type representing immutable arrays) and Haskell (for a similar reason, with the Haskell *IArray* class type and its corresponding instance *IArray.Array*, and also because there is a built-in *Rational* type representing arbitrary precision rational numbers).

Additionally, we make use of *serialisations*, a process to map Isabelle types and operations to the corresponding ones in the target languages. Serialisations are a common practice in the code generation process (see, Haftmann (2013) for some introductory examples); otherwise, the source types and operations would be *generated* from scratch in the target languages, and the obtained code would be less readable and efficient (for instance, the *nat* type would be generated to an *ad-hoc* type with 0 and *Suc* as constructors, instead of using the built-in representation of

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

Table 1. *Type serialisations*

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

integers in the target language). The advantage of applying serialisations to the target languages suitably is stressed by an empirical result; profiling the computations carried out over matrices of rational numbers in SML, we detected that the greatest amount of time was spent in reducing fractions (operations *gcd* and *divmod*). Serialising these Isabelle operations to the corresponding built-in Poly/ML (2014) and MLton (2014) functions (which are not part of the SML Standard Library, but particular to each compiler), decreased by a factor of 20 the computing times.

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

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

The complete set of Isabelle serialisations that we used is shown in Table 1. The Isabelle types *rat*, *real* and *bit* represent respectively $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{Z}_2$. The SML type *IntInf.int* represents arbitrarily large integers. It is worth noting that the Isabelle type *real* can be also serialised to the types used for *rat* in SML and Haskell, preserving arbitrary precision and avoiding numerical stability issues. Types presented in bold face identify serialisations that were introduced by us as part of this work. We also contributed some improvements to the Isabelle Library in the serialisation to the SML type *Vector.vector*.

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

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

We also explored the serialisation of Isabelle type *real* to double-precision floating point formats (*Double* in Haskell, *Real.real* in SML) in the target languages, but the computations performed present the expected *round-off errors*.

The *bit* type admits multiple serialisations, ranging from boolean values to subsets of the integers (with either fixed or arbitrary size). Experimental results showed us that the best performing option was to serialise *bit* and its operations to the corresponding boolean types in the target languages.

4.2 Data type refinements

Some data types present better properties for specification and formalisation purposes. For instance, specifying an algorithm over sets is easier than doing so over lists. However, the latter data type is better suited for execution tests. Following this idea, the poor performance presented by functions representing matrices can be solved by means of a *data refinement* to a better performing data structure.

Data refinement (Haftmann *et al*., 2013) offers the possibility to replace an abstract data type in an algorithm by a concrete type; in particular, our intention is to replace the *vec* type representing vectors before applying code generation. In our development, we have used the Isabelle type *iarray* as the target type of our refinement. Accordingly, we define functions *vec-to-iarray* that convert elements of type *vec* to elements of type *iarray* (an operation *matrix-to-iarray* will convert iterated vectors to iterated iarrays, the natural representation for matrices in this setting).

```
definition vec-to-iarray::α^col::{mod-type} ⇒ α iarray
 where vec-to-iarray v = IArray.of-fun (λi. v$(from-nat i)) (CARD(col))
```

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

```
lemma [code-unfold]:
 fixes A::α::{field}^col::{mod-type}^row::{mod-type}
 shows matrix-to-iarray (Gauss-Jordan A) =
  Gauss-Jordan-iarrays (matrix-to-iarray A)
```

The code equation certifies that it is correct to replace the function *Gauss-Jordan* (defined over abstract matrices, or elements of type *vec*) by the function *Gauss-Jordan-iarrays*. As it can be observed, the code equation does not include premises; this is a requirement from the Isabelle code generator. The label *code-unfold*

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

instructs the code generation tool to record the lemma as a rewriting rule, replacing occurrences of the left-hand side in the execution and code generation processes by the right-hand side. From a broader perspective, the function *matrix-to-iarray* has to be proved to be a homomorphism between the original and the refined type.

The proving effort (*ca.* 2 800 code lines, see Table 4) to prove the type refinement is significantly smaller than the formalisation itself (*ca.* 12 200). It must be noticed that reproducing the formalisation over *iarrays* would presumably take more than the aforementioned 12 200 lines, since the type *vec* has particular properties that ease and simplify the proofs. At least two features of this refinement from *vec* to *iarray* can be identified that have an influence in the modest amount of work devoted to it. First, the close relationship between the original data type *vec* and the new data type *iarray*, since both of them share a similar interface that allows to easily *identify* operations between them. Second, the fact that we preserved the original algorithm design, replacing operations over the *vec* type by equivalent ones over the *iarray* type. Nevertheless, the Isabelle code generator leaves the door open to algorithmic refinements (obviously, the bigger the differences between the *abstract* and the *concrete* algorithms, the greater the proving effort that has to be invested to fill such a gap). This effort pays off in terms of reusability, since the lemmas proving the equivalence between abstract and concrete operations for elementary row (or column) operations could be reused in future developments (implementations of different algorithms over matrices).

```
lemma [code-unfold]:
fixes A::α::{semiring-1}^col::{mod-type}^row::{mod-type}
shows matrix-to-iarray (interchange-rows A i j) =
  interchange-rows-iarray (matrix-to-iarray A) (to-nat i)(to-nat j)
```

Mapping some other operations on the type *vec* to the type *iarray* may involve programming decisions. For instance, the *LEAST* operator, which in its Isabelle original definition merely has a logical specification (a description of the properties that the least element satisfying a predicate possesses), and has not an *executable* specification, needs to be proved equivalent to a user provided operation on the data type *iarray*. The definition below shows an operation on the type *iarray*, which is itself based on the operation *find* of lists. The Isabelle corollary presented below proves that this operation successfully finds the *least* (with respect to the indexing order) nonzero element in a matrix column j over a row i (if such an element exists, as expressed in the *assumes* clause).

```
definition least-nonzero-position-index v i =
  the (List.find (λx. v!!x ≠ 0) [i..<(IArray.length v)])
```

```
corollary
 fixes A::α::zero^col::mod-type^row::mod-type and j::col
 defines colj ≡ vec-to-iarray (column j A)
 assumes ¬(vector-allzero-from-index (to-nat i, colj))
 shows least-nonzero-position-index (colj) (to-nat i) =
  to-nat (LEAST n. A$n$j ≠ 0 ∧ i ≤ n )
```

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

5 The generated programs and related work

5.1 The generated programs

The Isabelle code generation facility is now applied to generate the SML and Haskell code from the Gauss-Jordan algorithm specification. Note that in this process both the serialisations and data type refinements presented in Section 4 are used. The automatically generated code sums up 2,300 lines in SML and 1,300 in Haskell (see Table 4). The SML and Haskell sources are available from Aransay & Divasón (2014c). They closely resemble the definitions presented in Section 3 of the Gauss-Jordan algorithm.

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

- The rank (computed, for instance, as the number of nonzero rows of the *rref*) of a $\mathbb{Z}_2$ matrix, permits the computation of the number of connected components of a digital image; in Neurobiology, this technique can be used to compute the number of synapses in a neuron (see Heras (2012) for details). With our programs, the computations can be carried out on images at least of $2\,560 \times 2\,560$ px. size (which are conventional ones in real life experiments). The running time to obtain these computations was *ca.* 160 s.
- Varona *et al.* (Durán *et al.*, 2014) detected that Mathematica®, at least in versions 8.0, 9.0 and 9.0.1, was erroneously computing determinants of matrices of big integers, even for matrices of small dimensions (in their work they present an example of a matrix of dimension 14×14). The same computation, performed twice, can produce different results. The bug was reported to the Mathematica® support service. The error might be originated in the use of some arithmetic operations modulus large primes (apparently, the prime numbers could be either not large enough or not as many as required). With our verified program, the computation takes *ca.* 0.64 s (in MLton; Haskell and Poly/ML performance are also practically the same over inputs of such sizes), and the result obtained is the same as in Sage (which requires 0.0 s) and Maple™. Our algorithm relies on the arbitrarily large integers as implemented in each one of the functional languages. The computing time in Mathematica® (of the wrong result) sums up 2.68 s.

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

5.2 Related work

The formalisation of Abstract Algebra has been of recurrent interest for the Theorem Proving community, and it would be difficult to provide here a complete survey

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

on the subject; even so, the formalisation of Linear Algebra algorithms has not received such an attention from the community. This is somehow surprising, since new algorithms are being continuously implemented in Computer Algebra systems in a search for better performance.

Nevertheless, it is worth mentioning the CoqEAL (standing for Coq Effective Algebra Library) effort (Dénès *et al.*, 2012). This work is devoted to develop a set of libraries and commodities over which algorithms over matrices can be implemented, proved correct, refined, and finally executed. We try to compare our work with theirs in each of these terms.

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

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

Refinements in CoqEAL can be performed both on the algorithms and on the data structures (Dénès *et al.*, 2013) (as we also do in our development); data type refinements in CoqEAL are made in a *parametricity* fashion, since algorithms are implemented and proved over unspecified operations (by means of *type classes*) that are later replaced with computational or proof oriented types. Then, some parametricity lemmas state the equivalence between computational and proof oriented operations. These lemmas sometimes introduce additional parameters (such as the size of matrices) that could be unnecessary in some of the representations. In practice, the data type refinements in CoqEAL are reduced to using either functions or lists for representing vectors (and then matrices iteratively). Our approach is more flexible since it is based in proving the equivalence (modulus type morphisms) of the operations in the computational and proof oriented types (Haftmann *et al.*, 2013). In Isabelle, there is no special requirement over the proof type and the computational type, as long as a suitable morphism exists between both, and operations that are proven equivalent in both types. Type classes are thus avoided, enhancing the flexibility of the refinements to be carried out.

Regarding *execution*, there is a crucial difference between CoqEAL's approach and ours. In the CoqEAL framework computations are carried out over the Coq virtual machine (and thus "inside" the Coq system, avoiding code generation), a bundle of Coq libraries to optimise computational times. Our approach is thought for code generation, and executions are completed in Haskell or SML (they can be also performed "inside" of Isabelle, but execution times are considerably slower). This

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

provides us with a remarkable edge in performance. For instance, the computation of the rank of a $\mathbb{Z}_2$ matrix of size $1\,000 \times 1\,000$ consumes 32 s. in Poly/ML but 121 s. in Coq (following the figures in Dénès (2013)), even if their algorithm exclusively computes the rank of the matrix, without any additional information, whereas ours produces the *rref* of the matrix. One potential source of inefficiency in CoqEAL performance could be the use of *lists*, instead of arrays, to encode vectors and matrices.

A different tempting approach would be to verify an *imperative* version of the algorithm producing the *rref* of a matrix. The implementation of imperative programs and (mutable) arrays in Isabelle/HOL has been studied in Bulwahn *et al.* (2008), through the use of suitable monads, giving place to a library called *Imperative HOL*. When applying code generation, mutable arrays are refined to the SML *Array.array* type, which is also mutable (whereas immutable arrays were refined in our work to the SML type *Vector.vector*). The authors present two different case studies; in one of them the performance gain is a 30% with respect to a purely functional implementation using balanced trees. The use of monads introduces inherent intricacies both in the definitions and also in the consequent formalisations.

As an experiment, we produced a (non-formalised) *imperative* implementation of the Gauss-Jordan algorithm, inspired by Algorithm 1 and our functional representation (see file *Gauss-Jordan-Imp.thy* in Aransay & Divasón (2014c)). Algorithm 1 involves numerous matrix rows and column traversals that we have implemented by recursion (up to our knowledge, there is not a suitable array traverse operator in Imperative HOL).

We generated SML code from this imperative version, and carried out some performance tests between the *functional* verified version presented in Section 5.1 and the new *imperative* one. Rather surprisingly, the functional and imperative versions performed almost equally (with a little bias of a 10% for each version depending on the compiler used, MLton or Poly/ML). Profiling both algorithms (see Table 2) it seemed clear that the amount of time spent in the functional version of the algorithm in the construction of new immutable arrays (operation *of-fun*) was greater than the amount of time devoted in the imperative version (operation *make.fn*), but this amount of time was compensated by the time spent in the imperative version performing updating and the remaining recursive operations (operations *sub* and *nth.fn* represent both the array access operation).

Table 2 presents the result of profiling in MLton the computation of the *rref* of a 600×600 matrix with inputs in $\mathbb{Z}_2$ (profiling itself adds up an overhead of almost 20% in the computing times).

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

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

Table 2. *Profiling of the "imperative" and verified versions of Gauss-Jordan*

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

Table 3. *C++ and verified versions of the Gauss-Jordan algoithm*

Matrix sizes	**C++ version**	**Verified version**
600×600	01.33 s.	06.16 s.
$1\,000 \times 1\,000$	05.94 s.	32.08 s.
$1\,200 \times 1\,200$	10.28 s.	62.33 s.
$1\,400 \times 1\,400$	16.62 s.	97.16 s.

As an additional test of the performance of the obtained verified code, we implemented the same algorithm as presented in Algorithm 1 in C++ (see file *Gauss-Jordan.cpp* in Aransay & Divasón (2014c)), where vectors are represented by C++ arrays, and array traversals are replaced by *for* loops. We produced some benchmarking to compare our verified functional version, executed with the Poly/ML interpreter, against the C++ version, that is presented in Table 3.

The obtained times with both versions are of the same order of magnitude, and grow cubically with respect to the number of rows of the matrices (it is worth noting that the Gauss-Jordan algorithm is of cubic order). Apparently, the use of *immutable* arrays in our verified code does not pose a drawback, and both algorithms scale similarly.

6 Conclusions and further work

In this work, we have presented a formalisation of the Gauss-Jordan algorithm and its principal applications in Linear Algebra. The algorithm has been proved in an environment (the HMA Library) that allows to keep the mathematical meaning of the results (for instance, the relation between matrices and linear maps) and over a representation of matrices which simplifies the proving effort. Additionally, the algorithm has been refined to a more suitable data type, and generated to SML and Haskell, languages in which we have been capable of executing the algorithm over matrices of remarkable dimensions (in our case study with digital images) and inputs (in matrices with *big* coefficients, that were erroneously computed in

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

Table 4. *List of files in the formalisation and generated code*

File name	Number of lines
Formalisation	
Dual-Order	71
Mod-Type	527
Generalizations	2 519
Miscellaneous	535
Rref	412
Fundamental-Subspaces	263
Code-Set	40
Code-Matrix	82
Elementary-Operations	1 031
Dim-Formula	366
Rank	45
Gauss-Jordan	2 416
Linear-Maps	1 229
Gauss-Jordan-PA	457
Determinants2	502
Inverse	332
Bases-Of-Fundamental-Subspaces	214
System-Of-Equations	621
Code-Bit	62
Examples-Gauss-Jordan-Abstract	149
Subtotal	12 183
Refinement	
IArray-Addenda	98
Matrix-To-IArray	415
Gauss-Jordan-IArrays	401
Gauss-Jordan-PA-IArrays	319
Bases-Of-Fundamental-Subspaces-IArrays	165
System-Of-Equations-IArrays	389
Determinants-IArrays	245
Inverse-IArrays	70
Examples-Gauss-Jordan-IArrays	193
Code-Generation-IArrays	103
Code-Generation-IArrays-SML	121
Code-Real-Approx-By-Float-Haskell	66
IArray-Haskell	111
Code-Rational	129
Code-Generation-IArrays-Haskell	59
Subtotal	2 884
Total Isabelle code	15 067
SML generated code (file Gauss-SML)	2 344
Haskell generated code (files Rational, IArray, Gauss-Haskell)	1 319

https://doi.org/10.1017/S0956796815000155 Published online by Cambridge University Press

Mathematica®). The infrastructure created can be certainly reused; some of the serialisations introduced in Section 4 are already part of the Isabelle Library. The mathematical results are proved over a well-established Isabelle Library (HMA). Results on matrices include many basic operations. The data type refinement from vectors as matrices to immutable arrays is also easily reusable, not only for Linear Algebra purposes.

The number of lines of verified code generated in our work (*ca.* 2,300 in SML, 1,300 in Haskell) is considerable and covers a wide range of applications in Linear Algebra. Both its formalisation and the refinement, available in Aransay & Divasón (2014c), took 15,000 lines of Isabelle code. The HMA Library infrastructure was intensively reused, reducing significatively the amount of mathematical results to be formalised. Isabelle/HOL also provided powerful and efficient features to produce data type refinements. Table 4 shows the collection of files and their sizes produced in our work.

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

As a natural continuation to our work, and once we have observed that mutable data types and *imperative* algorithms do not dramatically increase the performance of the obtained code, we consider that the refinement of algorithms should be a sensible subject. Refining our *naive* implementation of Gauss-Jordan to better performing algorithms shall be an interesting task. Additionally, once we have solved the problem of computing the solution of systems of linear equations, we could also explore the least squares problem that allows to find the closest point to a solution in systems of linear equations with no solution.

Acknowledgments

David Matthews, Matthew Fluet, Tjark Weber and Jasmin C. Blanchette gave us valuable advice to speed up our programs in Poly/ML, MLton and GHC. David Matthews also modified Poly/ML 5.5.1 to improve the performance offered by variable binders.

Juan Luis Varona helped us to perform Mathematica® computing tests and gave us valuable information about this system. His work encourages formalisation efforts and consequently motivates ours.

The referees made significant contributions to the first version of this paper, specially concerning the presentation of the generated programs and related work in Section 5.

This work has been supported by the research grant FPI-UR-12, from Universidad de La Rioja and by the project MTM2014-54151-P from Ministerio de Economía y Competitividad (Gobierno de España).

References

Aransay, J. & Divasón, J. (2014a) Formalization and execution of Linear Algebra: From theorems to algorithms. In Post Proceedings of the International Symposium on Logic-Based Program Synthesis and Transformation: LOPSTR 2013, Gupta, G. & Peña, R. (eds), LNCS, vol. 8901. Madrid, Spain: Springer, pp. 01–19.

Aransay, J. & Divasón, J. (2014b) Gauss-Jordan algorithm and its applications. *Arch. Formal Proofs*. Available at: http://afp.sf.net/entries/Gauss_Jordan.shtml, Formal proof development.

Aransay, J. & Divasón, J. (2014c) Gauss-Jordan elimination in Isabelle/HOL. Available at: http://www.unirioja.es/cu/jodivaso/Isabelle/Gauss-Jordan-2013-2-Generalized/

Aransay, J. & Divasón, J. (2015) Generalizing a Mathematical Analysis Library in Isabelle/HOL. NASA Formal Methods, Havelund, K., Holzmann, G. & Joshi, R. (eds), LNCS, vol. 9058. Pasadena, CA, USA: Springer, pp. 415–421.

Avigad, J., & Harrison, J. (2014) Formally verified mathematics. *Commun. ACM* **57**(4), 66–75.

Avigad, J., Hölzl, J. & Serafin, L. (2014) *A Formally Verified Proof of the Central Limit Theorem*. Available at: http://arxiv.org/abs/1405.7012v1.

Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L. & Matthews, J. (2008) Imperative functional programming with Isabelle/HOL.

In TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics, O. Mohamed, C. Muñoz & Tahar, S. (eds), LNCS, vol. 5170. Montreal, Canada: Springer, pp. 352–367.

Dénès, M. (2013) *Étude Formelle d'algorithmes Efficaces En Algèbre Linéaire*. PhD Thesis, INRIA Sophia Antipolis, France.

Dénès, M., Mörtberg, A. & Siles, V. (2012) A refinement-based approach to computational algebra in COQ. In ITP - 3rd International Conference on Interactive Theorem Proving - 2012, Beringer, L. & Felty, A. (eds), LNCS, vol. 7406. Princeton, NJ, USA: Springer, pp. 83–98.

Dénès, M., Mörtberg, A. & Siles, V. (2013) Refinements for free!. *Certified Programs and Proofs*, Gonthier, G. & Norrish, M. (eds), LNCS, vol. 8307. Melbourne, VIC, Australia: Springer, pp. 147–162.

Durán, A. J., Pérez, M. & Varona, J. L. (2014) Misfortunes of a mathematicians' trio using computer algebra systems: Can we trust? *Notices AMS* **61**(10), 1249–1252.

Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A. & Smaus, J. G. (2013) A fully verified executable LTL model checker, Computer Aided Verification: CAV 2013, Sharygina, N. & Veith, H. (eds), LNCS, vol. 8044. Saint Petersburg, Russia: Springer, pp. 463–478.

Gockenbach, M. S. (2010) *Finite-Dimensional Linear Algebra*. CRC Press.

Haftmann, F. (2013) Code generation from Isabelle/HOL theories. Available at: http://isabelle.in.tum.de/dist/Isabelle2013-2/doc/codegen.pdf.

Haftmann, F., Krauss, A., Kuncar, O. & Nipkow, T. (2013) Data refinement in Isabelle/HOL. In Interactive Theorem Proving: ITP 2013, Blazy, S., Paulin-Mohring, C., & Pichardie, D. (eds), LNCS, vol. 7998. Rennes, France: Springer, pp. 100–115.

Haftmann, F. & Nipkow, T. (2010) Code generation via higher-order rewrite systems. In Functional and Logic Programming: 10th International Symposium: FLOPS 2010, Blume, M., Kobayashi, N. & Vidal, G. (eds), LNCS, vol. 6009. Sendai, Japan: Springer, pp. 103–117.

Hales, T. C. & Ferguson, S. P. (2011) *The Kepler Conjecture. The Hales-Ferguson Proof*. New York: Springer.

Harrison, J. (2005) A HOL theory of Euclidean space. In Theorem Proving in Higher Order Logics, Hurd, J. & Melham, T. (eds), LNCS, vol. 3603. Oxford, UK: Springer, pp. 114–129.

Harrison, J. (2013) The HOL Light theory of Euclidean space. *J. Autom. Reason.* **50**(2), 173–190.

Haskell. (2014) The Haskell Programming Language. Available at: http://www.haskell.org/.

Heras, J., Dénès, M., Mata, G., Mörtberg, A., Poza, M. & Siles, V. (2012) Towards a certified computation of homology groups for digital images. In Computational Topology in Image Context: CTIC 2012, Ferri, M., Frosini, P., Landi, C., Cerri, A. & Fabio, B. D. (eds), LNCS, vol. 7309. Bertinoro, Italy: Springer, pp. 49–57.

HMA. (2014) HOL Multivariate Analysis Library. Available at: http://isabelle.in.tum.de/library/HOL/HOL-Multivariate_Analysis/index.html.

Milner, R., Harper, R., MacQueen, D. & Tofte, M. (1997) *The Definition of Standard ML, revised edition*. MIT Press.

MLton. (2014) The MLton website. Available at: http://mlton.org/.

Nipkow, T. (2011) Gauss-Jordan elimination for matrices represented as functions. *Arch. Formal Proofs*. Available at: http://afp.sf.net/entries/Gauss-Jordan-Elim-Fun.shtml, Formal proof development.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002) *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS, vol. 2283. Springer.

Paulson, L. C. (1990) *Logic and Computer Science*. Academic Press. Chap. Isabelle: The next 700 theorem provers, pp. 361–388.

Poly/M. L. (2014) The Poly/ML website. Available at: http://www.polyml.org/.

Roman, S. (2008) *Advanced Linear Algebra*. 3rd edn. Springer.

Sternagel, C. (2013) Proof pearl - a mechanized proof of GHC's mergesort. *J. Autom. Reasoning* **51**(4), 357–370.

Strang, G. (2009) *Introduction to Linear Algebra*. 4th edn. Wellesley-Cambridge Press.