Using Abstract Stobjs in ACL2 to Compute Matrix Normal Forms

Laureano Lambán[1], Francisco J. Martín-Mateos[2(✉)], Julio Rubio[1], and José-Luis Ruiz-Reina[2]

[1] Department of Mathematics and Computation, University of La Rioja, Logroño, Spain
{lalamban,julio.rubio}@unirioja.es
[2] Department of Computer Science and Artificial Intelligence, University of Sevilla, Seville, Spain
{fjesus,jruiz}@us.es

Abstract. We present here an application of abstract single threaded objects (*abstract stobjs*) in the ACL2 theorem prover, to define a formally verified algorithm that given a matrix with elements in the ring of integers, computes an equivalent matrix in column echelon form. Abstract stobjs allow us to define a sound logical interface between matrices defined as lists of lists, convenient for reasoning but inefficient, and matrices represented as unidimensional stobjs arrays, which implement accesses and (destructive) updates in constant time. Also, by means of the abstract stobjs mechanism, we use a more convenient logical representation of the transformation matrix, as a sequence of elemental transformations. Although we describe here a particular normalization algorithm, we think this approach could be useful to obtain formally verified and efficient executable implementations of a number of matrix normal form algorithms.

Keywords: Matrices · ACL2 · Abstract stobjs · Matrix normal forms

1 Introduction

Computing normal forms of matrices is a wide subject which presents many applications in different areas of Mathematics. For instance, one of the fundamental processes in Linear Algebra is the resolution of systems of linear equations, and the constructive methods to carry that task out are based on the computation of triangular forms of a given matrix. In the same way, Smith normal form, a particular kind of equivalent diagonal matrix, plays an essential role in the theory of finitely generated modules over a ring and, in particular, it is a key result to determine the structure of a finitely generated abelian group. Smith form also provides a well-known method for finding integer solutions of

Supported by Ministerio de Ciencia e Innovación, projects TIN2013-41086-P and MTM2014-54151-P.

M. Ayala-Rincón and C.A. Muñoz (Eds.): ITP 2017, LNCS 10499, pp. 354–370, 2017.
DOI: 10.1007/978-3-319-66107-0_23

systems of linear Diophantine equations [11]. The key point of all these procedures is to ensure that the output matrix (a reduced form) preserves some of the fundamental invariants of the input matrix such as the row (column) space, the rank, the determinant, the elementary divisors and so on.

There exists a huge range of algorithmic methods for computing normal forms of matrices [12], which are based on well established mathematical results. Nevertheless, it is advisable to have verified programs available in order to avoid the possible inaccuracies which can occur during the path from algorithms to programs. The aim is the paper is to propose a data structure and a logical infrastructure to implement formally verified matrix normal forms algorithms, in the ACL2 theorem prover, with special emphasis on how to efficiently execute the verified algorithms.

The ACL2 system [1] is at the same time a programming language, a logic for reasoning about models implemented in the language, and a theorem prover for that logic. The programming language is an extension of an applicative subset of Common Lisp, and thus the verified algorithms can be executed, under certain conditions, in the underlying Common Lisp. ACL2 has several features mainly devoted to get an efficient execution of the algorithms, in a sound way with respect to the logic. Abstract single-threaded objects [1,7] is one of those features, providing a sound logical connection between efficient concrete data structures and more abstract data structures, convenient for reasoning. We propose here to use this feature to implement and formally verify matrix algorithms for computing normal forms.

In particular, we describe in this paper a formally verified implementation of an algorithm to compute a column echelon form of a matrix with elements in the ring of integers. This formalization is done as an initial step for developing computational homological algebra in the ACL2 system and in particular to calculating (persistent) homology [10]. But although we describe here the formalization of a specific normalization algorithm, we think this approach could be generalized to other normalization algorithms as well.

The organization of the paper is as follows. The next section is devoted to describe a formalization of matrices in ACL2, represented as lists of lists, and also a representation for matrix normalization problems. This representation is natural for reasoning, but has inefficiencies due to the applicative nature of Lisp lists. Section 3 describes how we can compute using more efficient data structures, and still have the more natural representation for reasoning, by means of ACL2 abstract single-threaded objects. In Sect. 4, we illustrate this infrastructure describing how we formally verified an algorithm for computing a column echelon form for integer matrices. The paper ends with some discussion about related work and conclusions. Due to the lack of space, we will omit some ACL2 definitions and skip some technical details (for example, all the functions declarations). The complete source files containing the ACL2 formalization are accessible at: http://www.glc.us.es/fmartin/acl2/mast-cef.

2 A Data Structure for Reasoning in the Logic

In this section, we describe a data structure that can be used to define a matrix normal form algorithm. This data representation is suitable for reasoning, but inefficient for execution, as we will see. We will refer to this as the *abstract representation*.

2.1 Matrices as Lists of Lists

A very natural way to represent a 2-dimensional matrix in ACL2 is as a list whose elements are lists of the same length, each one representing a row of the matrix. For example, the list `'((1 0 0 0) (0 1 0 0) (0 0 1 0) (0 0 0 1))` represents the identity matrix of dimension 4. The following function `matp` is the recognizer for well-formed matrices represented as lists of lists:

```
(defun matp-aux (A ncols)
  (cond ((atom A) (equal A nil))
        (t (and (true-listp (first A))
                (equal (len (first A)) ncols)
                (matp-aux (rest A) ncols)))))

(defun matp (A)
  (if (atom A)
      (equal A nil)
      (and (consp (first A))
           (matp-aux A (len (first A))))))
```

Note that if `(matp A)`, then the number of rows of `A` is given by its length, and the number of columns by the length of (for instance) its first element. In our formalization, these are defined by the functions `nrows-m` and `ncols-m`, respectively. We have also defined the function `(matp-dim A` m n`)` checking that `A` is a matrix of a given size $m \times n$. As we have said in the introduction, the algorithm we have formalized is restricted to matrices with elements in the ring of integers; the function `integer-matp` (and `integer-matp-dim`) recognizes the ACL2 object that are `matp` and with all its elements being integers.

Accessing and updating matrix elements is done via `nth` and `update-nth`, respectively, as defined by the following functions `aref-m` and `update-aref-m`:

```
(defun aref-m (A i j)
  (nth j (nth i A)))

(defun update-aref-m (i j val A)
  (update-nth i (update-nth j val (nth i A)) A))
```

Using this representation, these operations are not done in constant time, and updating is not destructive, since it follows the usual "update by copy"

semantics of applicative lists. This is a drawback if we want efficient algorithms on matrices. In the next section we will show how to address this issue.

A typical definition scheme for matrix operations or matrix properties is by means of two nested loops, the outer iterating on its rows indices, and the inner on its column indices for a fixed row. In our formalization, this is done using two recursive functions. The following definition of the product of two matrices illustrates this recursion scheme:

```
(defun matrix-product-row-col (A B P i j cA cP)
  (cond ((or (not (natp j)) (not (natp cP))) P)
        ((>= j cP) P)
        (t (let ((P1 (update-aref-m i j
                                    (mp-res-i-j A B i j 0 cA) P)))
             (matrix-product-row-col A B P1 i (1+ j) cA cP)))))

(defun matrix-product-row (A B P i rP cA cP)
  (cond ((or (not (natp i)) (not (natp rP))) P)
        ((>= i rP) P)
        (t (let ((P1 (matrix-product-row-col A B P i 0 cA cP)))
             (matrix-product-row A B P1 (1+ i) rP cA cP)))))
```

Here `mp-res-i-j` implements the sum $\sum_k a_{ik}b_{kj}$, and `P` is a matrix with the same number of rows as `A` and the same number columns as `B`, where we store the resulting matrix product. Thus, matrix product is defined by the following function:

```
(defun matrix-product (A B)
  (let* ((rA (nrows-m A))
         (cA (ncols-m A))
         (cB (ncols-m B))
         (P (initialize-mat rA cB nil)))
    (matrix-product-row A B P 0 rA cA cB)))
```

Using this representation for matrices, we proved a number of well-known algebraic properties of matrix operations. For example, the following are the statements for product associativity and right identity (where `matrix-id` defines the identity matrix of a given dimension):

```
(defthm matrix-product-associative
  (implies (and (matp A) (matp B)
                (equal (nrows-m B) (ncols-m A))
                (equal (nrows-m C) (ncols-m B)))
           (equal (matrix-product (matrix-product A B) C)
                  (matrix-product A (matrix-product B C)))))

(defthm matrix-product-right-identity
  (implies (integer-matp-dim A (len A) n)
           (equal (matrix-product A (matrix-id n)) A)))
```

A general technique we used to prove most of these algebraic properties is based on the property that `(equal P Q)` if `P` and `Q` are matrices of the same dimension $m \times n$ such that $p_{ij} = q_{ij}$ for $0 <= i < m, 0 <= j < n$. We proved this property in a general way using the ACL2 encapsulation mechanism, and then we use it by functional instantiation, after proving the corresponding algebraic property for the individual entries of both sides of the equality. See the book `matrices-lists-of-lists.lisp` in the supporting materials, for details.

2.2 An Abstract Representation for Matrix Normal Form Computation

Algorithms that compute matrix normal forms, often compute also transformation matrices that relate the original matrix with its normal form. For example, in the algorithm we describe in Sect. 4, the goal is to obtain, for a given matrix A, a matrix H in a desired normal form and an invertible transformation matrix[1] T such that $A \cdot T = H$. A general description of a matrix normal form algorithm could be the following: we operate on two matrices, initially the original matrix and the identity matrix; at every step, an elementary transformation (or *operator*) is applied to the first matrix and the same transformation is applied to the second matrix; when the algorithm stops, we have H and T with the desired properties.

We now explain a possible data structure for such algorithms, which turns out to be natural for reasoning. First, we will represent the matrix A being transformed, using the list of lists representation described in the previous subsection. For the transformation matrix T we adopt a different approach: although the executable algorithm will deal with the whole matrix, in the logic it will be more convenient to see that transformation matrix as a list of operators, describing the sequence of elementary transformations carried out; and each operator will be a short description of the transformation. The reason is that it is easier to prove the properties of the transformation matrix, if we explicitly have the sequence of elementary transformations that this matrix represents.

For our concrete normal form algorithm described in Sect. 4, it turns out that only one type of elementary transformation is needed[2]: given two distinct column indices `c1` and `c2` and four integers `x1`, `x2`, `y1` and `y2`, this transformation replaces column `c1` by the linear combination of column `c1` times `x1` plus column `c2` times `x2`, and also replaces column `c2` by the linear combination of column `c1` times `y1` plus column `c2` times `y2`. We will call this operator a *linear combination of columns (lcc)*, and in the logic it will be represented as the list `(c1 c2 x1 x2 y1 y2)`. In our formalization, the function `(lcc-op l n)` checks if `l` is such

[1] Some algorithms for computing matrix normal forms, like the Smith normal form, need to compute two transformation matrices, but similar ideas would apply in that case.

[2] Of course, other normal forms algorithms needs different elementary transformations, and possible more than one. But again, the same ideas described here could be applied in such cases.

operator, where `c1` and `c2` are less than `n`. And `(lcc-op-seq seq n)` checks if `seq` is a list of lcc operators.

The above considerations lead us to the following predicate `mast$ap`, recognizing the data representation we have just described (the prefix `$a` is for "abstract"):

```
(defun mast$ap (x)
  (and (true-listp x)
       (equal (len x) 2)
       (let ((A (first x))
             (seq (second x)))
        (cond ((atom A) (and (equal A nil) (equal seq nil)))
              (t (and (integer-matp-dim A (nrows-m A) (ncols-m A))
                      (lcc-op-seq seq (ncols-m A))))))))
```

We have defined a number of functions that operate on this data structure. The main operation is linear combination of columns. For that, we first need to define the function `lin-comb-cols-lst`, which effectively carries out the linear combination of columns on a given matrix. Note that here we have an extra parameter `max-r`, which indicates a row index. This allows us to perform the linear combination of columns only until that row, but not below (the reason is that during the transformation process, we will be sure that there will only be zeros below a given row):

```
(defun lin-comb-cols-lst-rows (A c1 c2 r max-r x1 x2 y1 y2)
  (cond ((or (not (natp max-r)) (not (natp r))) A)
        ((> r max-r) A)
        (t (let* ((Arc1 (aref-m A r c1))
                  (Arc2 (aref-m A r c2))
                  (nArc1 (+ (* x1 Arc1) (* x2 Arc2)))
                  (nArc2 (+ (* y1 Arc1) (* y2 Arc2)))
                  (nA (update-aref-m r c2 nArc2
                                     (update-aref-m r c1 nArc1 A))))
             (lin-comb-cols-lst-rows nA c1 c2 (1+ r) max-r
                                     x1 x2 y1 y2)))))

(defun lin-comb-cols-lst (A c1 c2 max-r x1 x2 y1 y2)
  (lin-comb-cols-lst-rows A c1 c2 0 max-r x1 x2 y1 y2))
```

Now, the following function implements the lcc transformation on our abstract representation. Note that the transformation is only effectively carried out on the first matrix:

```
(defun lin-comb-cols$a (mast$a c1 c2 max-r x1 x2 y1 y2)
  (list (lin-comb-cols-lst (first mast$a) c1 c2 max-r x1 x2 y1 y2)
        (cons (list c1 c2 x1 x2 y1 y2) (second mast$a))))
```

We would like to define our matrix normal form algorithm using this and other functions defined on the abstract representation, but as we have said we can improve execution if we do not use applicative lists. And also, probably, if we were not interested in formal verification, we wouldn't have dealt with lcc operators, but with the whole transformation matrix instead.

3 Using Abstract Stobjs to Represent Matrices

So let us now define an executable and efficient data structure representation, and see how we can relate it to the abstract representation described above. Efficient execution is achieved in the ACL2 system mainly by means of two features: guards and single threaded objects. The *guard* of a function is a specification of its intended domain. Although functions in the ACL2 logic are total, guards provide a way to specify and verify the inputs for which the function can be safely executed directly in the underlying raw Common Lisp. A *guard-verified* function respects the guards of all the functions that it calls (including itself in case of a recursive function). All the functions involved in the algorithm of Sect. 4 have been guard-verified.

The second feature related to efficient execution is provided by single threaded objects (*stobjs*). These are data structures that allow accessing and updating in constant time, and destructive updates on them. When an object is declared to be single-threaded, ACL2 enforces certain syntactic restrictions on its use, ensuring that in every moment, only one copy of the object is needed (for example, one of these restrictions requires that if a function updates a stobj, then it has to return the stobj). With these restrictions, the destructive updates are consistent with the applicative functional semantics of ACL2.

Therefore, it would be good if we can execute our matrix algorithms using stobjs. Nevertheless, although we can use arrays as fields of a stobj, those arrays have to be 1-dimensional and accessing and updating the array is only allowed via elementary operations, so reasoning directly using this representation could be difficult. Fortunately, another ACL2 feature, abstract stobjs, will allow us to define an alternative logical interface for the stobj.

3.1 A Stobj for Computing Matrix Normal Forms

Before describing the abstract stobj we have used, let us show the corresponding stobj, where the execution will take place (we will call this the *concrete representation*). In ACL2, a stobj is defined, using `defstobj`, as a structure with a number of fields, where each field can be either of array type or of non-array type. In our case, we will define a stobj with two 1-dimensional array fields, each one storing the elements of a 2-dimensional matrix, in linearized form. The idea is that one of the 1-dimensional arrays stores the matrix being transformed, and the other stores the transformation matrix. We also need two non-array fields, to store the number of rows and the number of columns of the first matrix. The following defines this stobj (the `$c` suffix is for *concrete*):

```
(defstobj mast$c
  (nrows$c  :type (integer 0 *) :initially 0)
  (ncols$c  :type (integer 0 *) :initially 0)
  (matrix$c :type (array integer (0)) :initially 0 :resizable t)
  (trans$c  :type (array integer (0)) :initially 0 :resizable t))
```

Array fields in stobjs are defined in the logic as ordinary lists, but for execution in the underlying Lisp, raw Lisp arrays are used. The effect of this ACL2 form is to introduce the stobj `mast$c` and its associated recognizers, creator, accessors, updaters, and length and resize functions for its fields. For example, given an index `i`, `(matrix$ci i mast$c)` and `(update-matrix$ci i v mast$c)` respectively access and update (with value `v`) the `i`-th cell of the `matrix$c` array. Similar functions are defined for the `trans$c` array. These operations are executed in constant time and the update is destructive (at the price of syntactic restrictions on the use of the stobj). Logically speaking, they are defined in terms of `nth` and `update-nth`.

We have defined a number of functions operating on this concrete representation. Let us show, for example, how we implement the linear combination of columns. First, the following function performs that operation on the first matrix (we omit some technical details):

```
(defexec lin-comb-cols-matrix$c-rows
                                (mast$c i j s r max-r x1 x2 y1 y2)
  ...
  (cond ((> r max-r) mast$c)
     (t (let* ((mat-i (mat$ci i mast$c))
               (mat-j (mat$ci j mast$c))
               (new-mat-i (+ (* x1 mat-i) (* x2 mat-j)))
               (new-mat-j (+ (* y1 mat-i) (* y2 mat-j))))
          (seq mast$c
               (update-mat$ci i new-mat-i mast$c)
               (update-mat$ci j new-mat-j mast$c)
               (lin-comb-cols-matrix$c-rows mast$c (+ i s) (+ j s)
                                s (1+ r) max-r x1 x2 y1 y2))))))

(defun lin-comb-cols-matrix$c (mast$c c1 c2 max-r x1 x2 y1 y2)
  (lin-comb-cols-matrix$c-rows
      mast$c c1 c2 (ncols$c mast$c) 0 max-r x1 x2 y1 y2))
```

Here `i` and `j` are indices of positions in the 1-dimensional array (initially, `c1` and `c2`, respectively), and `r` is the current row of the corresponding 2-dimensional array (initially 0). Note that to move to the next row in both columns, we add `s` (the number of columns) to both indices.

In a very similar way, we define a function `lin-comb-cols-trans$c` that performs the same operation on the `trans$c` 1-dimensional array. And finally, we sequentially apply both transformations (note that the operation on the transformation matrix is performed until the last row):

```
(defun lin-comb-cols$c (mast$c c1 c2 max-r x1 x2 y1 y2)
  (seq mast$c
       (lin-comb-cols-matrix$c mast$c c1 c2 max-r x1 x2 y1 y2)
       (lin-comb-cols-trans$c mast$c c1 c2
                              (1- (ncols$c mast$c)) x1 x2 y1 y2)))
```

3.2 The Abstract Stobj

Until now, we have defined an abstract representation (convenient for reasoning), and also a concrete representation (suitable for execution). In both representations, we have defined functions that perform the main operations needed for our matrix normal form algorithm. Now we can combine the best of both representations, thanks to abstract stobjs.

But before we have to introduce a (non-executable) *correspondence predicate*, describing in what sense the concrete and the abstract representations are related. Basically: the concrete representation stores the size of the matrix in the abstract representation; the first 1-dimensional array of the concrete representation is a linearized version of the matrix of the abstract one; and the second 1-dimensional matrix of the concrete representation is a linearized version of the result of applying the sequence of lcc operators of the abstract representation, to the identity matrix:

```
(defun-nx mast$corr (mast$c mast$a)
  (let ((nrows (len (first mast$a)))
        (ncols (len (first (first mast$a)))))
    (and (equal nrows (nth 0 mast$c))
         (equal ncols (nth 1 mast$c))
         (equal (append-lst (first mast$a)) (nth 2 mast$c))
         (equal (append-lst (apply-lcc-op-seq (second mast$a)
                                              (matrix-id ncols)))
                (nth 3 mast$c)))))
```

Here `(append-lst ls)` is a function that given a list of lists `ls`, concatenates all of them into one single list. And `apply-lcc-op-seq` is a function that applies a sequence of lcc operators to a given matrix. Here it is its definition:

```
(defun apply-lcc-op (op A)
  (let ((c1 (nth 0 op)) (c2 (nth 1 op)) (x1 (nth 2 op))
        (x2 (nth 3 op)) (y1 (nth 4 op)) (y2 (nth 5 op)))
    (lin-comb-cols-lst A c1 c2 (1- (nrows-m A)) x1 x2 y1 y2)))

(defun apply-lcc-op-seq (seq A)
  (cond ((endp seq) A)
        (t (apply-lcc-op (first seq)
                         (apply-lcc-op-seq (rest seq) A)))))
```

Note that this function is only for specification. In particular, we apply `lin-comb-cols-lst` to all the rows of the matrix and not only until a given row, since that optimization will only make sense in the particular implementation of a normalization algorithm.

Now we can define the abstract stobj that provides a sound logical connection between both representations. In ACL2, a `defabsstobj` event defines an abstract single-threaded object that is proven to satisfy a given invariant property, and that can only be accessed or updated by some given functions called *exports*. These functions have an abstract definition that ACL2 uses for reasoning and a different concrete implementation that ACL2 uses for execution on a corresponding concrete stobj. In our case, this is the abstract stobj we have defined:

```
(defabsstobj mast
  :exports (initialize-mast
            nrows ncols
            aref-mat
            lin-comb-cols
            get-mat
            get-trans))
```

Here `initialize-mast` is a function that given an initial matrix `A` (as a list of lists), stores it in the abstract stobj. The abstract definition for this export is straightforward (simply returns `(list A nil)`), but the concrete executable definition is far more difficult, since it has to store each element of `A` and each element of the identity matrix in the corresponding 1-dimensional arrays of the stobj. As for `lin-comb-cols`, we have already presented its abstract and concrete definitions. These two exports update the abstract stobj, and the rest of the exports are only accessors: `nrows` and `ncols` give the number of rows and columns of the first matrix, `aref-mat` access to an element of the first matrix by its row and column indices; and `get-mat` and `get-trans` return, respectively, the first and the second matrices, as list of lists. Note that again this is easy for the abstract representation (especially `get-mat`) but it is not trivial for the concrete definitions.

Unless specified, the names for the corresponding concrete stobj, the correspondence predicate, and for the abstract and concrete functions associated with each export, are obtained appending the suffixes `$a` (abstract) or `$c` (concrete) to the names given in the `defabsstobj`. To accept a `defabsstobj` event, all these corresponding abstract and concrete functions have to be previously defined, their guards verified, and also a number of proof obligations automatically generated by the event must be already proved. These proof obligations guarantee that the correspondence between the abstract and the concrete representation, the recognizer property, and the guards of the exports are preserved after updating the stobj, and also that the abstract and the concrete corresponding accessors return the same values. That is, the proof obligations essentially guarantee that reasoning with the abstract representation and executing with

the concrete representation is logically sound. See `matrices-abstobj.lisp` for the statements of all these proof obligations and a proof of them.

Once this abstract stobj is defined, we can use it as the data structure for a matrix normal form algorithm, provided that the single-threadedness syntactic restrictions are met. The only primitive functions we can use to access or update the abstract stobj are the exports. We emphasize that when proving theorems about the algorithm, ACL2 uses the abstract definitions of the exports (that is, the ones with the `$a` suffix); but for execution, it uses the concrete data structure and definitions (that is, the ones with the `$c` suffix).

4 An Algorithm to Compute a Column Echelon Form

We illustrate how we can use the described absstobj framework, by means of a verified implementation of an algorithm that given a matrix of integers A, computes an equivalent integer matrix C that it is in column echelon form, together with a unimodular transformation integer matrix T such that $A \cdot T = C$. We say that a matrix C is in *column echelon form* if zero columns of C precede nonzero columns and, for each nonzero column of C, its leading entry (the last nonzero element of the column) is above the leading entries of the following columns. This notion of column echelon form is not exactly the same as other classical echelon forms usually defined in the literature, such as Hermite or Howell forms. Nevertheless, as we have said in the introduction, this has to be considered in the context of developing ACL2 programs to compute homology groups of chain complexes, and it turns out that this simple echelon transformation is suitable for this task. And anyway, our main purpose here is to illustrate with this example how we can apply the absstobj infrastructure just described.

Although the algorithm is implemented for integer matrices, it could be generalized to matrices in a more general class of rings, namely, the class of Bézout domains. Roughly speaking, a Bézout domain is an integral domain where every finite ideal is principal. This property is equivalent to the existence of an explicit Greatest Common Divisor (*gcd*) operation providing the Bézout identity of every pair of elements: if d is the gcd of two elements a and b, there exist two elements x and y such that $d = ax + by$. Note that in a ring we do not have in general the inverse of an element, so we cannot apply here usual techniques employed when the entries are in a field (like Gaussian elimination).

4.1 Definition of the Algorithm

Let us now present the ACL2 implementation of the column echelon form algorithm. First, a key ingredient is the extended Euclides algorithm which, besides the greatest common divisor of two integers, computes the coefficients of the Bézout identity. In particular, we have defined a function `(bezout a b)` such that given two integers a and b, returns a tuple of integers `(g s1 t1 s2 t2)` such that $g = gcd(a, b), s_1a + t_1b = d$ and $s_2a + t_2b = 0$. Note that these properties can be expressed in matrix form:

$$(a \;\; b) \cdot \begin{pmatrix} s_2 & s_1 \\ t_2 & t_1 \end{pmatrix} = (0 \;\; gcd(a,b))$$

This 2×2 matrix has the property that it is unimodular (determinant 1 or -1) and thus invertible *in the ring of integers*. It is an elementary transformation matrix that can be also easily generalized to size $n \times n$, in such a way that right multiplication by this elementary matrix is just like applying a lcc operator. Essentially, the algorithm iteratively applies this transformation with the aim of obtaining the zero entries needed in the echelon form. This is done from the last row to the first one, and in every row, from a given column to the first one.

The following functions implement the algorithm operating on the abstract stobj `mast`. This means that the only elementary operations we can apply to `mast` are the exports specified in its `defabsstobj`. The first function is `cef-bezout-row-col` below, which given a row index `(- i 1)` and column indices `(- c 1)` and `(- j 1)`, apply the lcc transformation on those columns, and thus obtaining a zero in the position of row `(- i 1)` and column `(- c 1)`, using as pivot the entry of the same row and column `(- j 1)`. This is done when we already know that the entries of the given columns that are below the given row are already zero, so it is justified to do the linear combination only until that row:

```
(defun cef-bezout-row-col (mast c i j)
  (mv-let (g s1 t1 s2 t2)
      (bezout (aref-mat mast (- i 1) (- c 1))
              (aref-mat mast (- i 1) (- j 1)))
      (lin-comb-cols mast (- c 1) (- j 1) (- i 1) s2 t2 s1 t1)))
```

Given the position of a pivot, this lcc transformation is applied for all the columns to the left, obtaining zeros in the row of the pivot, until the column of the pivot. This recursive process is carried out by the function `cef-reduct-row-col` and initiated by the function `cef-reduct-row`, from a given pair of row and column indices:

```
(defun cef-reduct-row-col (mast c i j)
  (cond ((zp c) mast)
        (t (seq mast
                (cef-bezout-row-col mast c i j)
                (cef-reduct-row-col mast (- c 1) i j)))))

(defun cef-reduct-row (mast i j)
  (cond ((zp j) mast)
        (t (cef-reduct-row-col mast (- j 1) i j))))
```

To get the echelon form, we iteratively apply this process from the last row to the first one. We also have to take into account that the column of the pivot is changing from one row to the next, depending on the result obtained after reducing that row. If we have a zero in the position of the pivot, the column of the pivot is unchanged. Otherwise is decremented by one:

```
(defun cef-row-col (mast i j)
  (cond ((or (zp i) (zp j)) mast)
        (t (let ((mast (cef-reduct-row mast i j)))
             (if (= (aref-mat mast (- i 1) (- j 1)) 0)
                 (cef-row-col mast (- i 1) j)
                 (cef-row-col mast (- i 1) (- j 1)))))))
```

Given an input matrix `A` (represented as lists of lists). The algorithm is initiated calling the export `initialize-mast`, and then the function `cef-row-col`, starting in the last row and columns:

```
(defun cef (A mast)
  (seq mast
       (initialize-mast mast A)
       (cef-row-col mast (nrows mast) (ncols mast))))
```

Note that the above function `cef` receives as input the `mast` abstract stobj and thus, due to the single-threadedness requirements, it has to return also the abstract stobj. Nevertheless, we can define a function `cef-matrix` in which the input and output are not explicitly connected to the stobj. This can be done using `mast` locally (by means of `with-local-stobj`), and finally returning the computed matrices represented as lists of lists (using the exports `get-mat` and `get-trans`):

```
(defun cef-matrix (A)
  (with-local-stobj mast
    (mv-let (mast mat trans)
            (seq mast
                 (cef A mast)
                 (mv mast (get-mat mast) (get-trans mast)))
            (mv mat trans))))
```

4.2 Main Theorems Proved

We proved in ACL2 the following theorems, stating that given an integer matrix `A`, the algorithm `cef-matrix` computes an equivalent integer matrix that is in column echelon normal form:

```
(defthm cef-cef-matrix
  (implies (integer-matp A)
           (let ((H (first (cef-matrix A))))
                (and (integer-matp-dim H (nrows-m A) (ncols-m A))
                     (cef-p H)))))

(defthm matrix-product-cef-matrix
  (implies (integer-matp A)
           (let ((H (first (cef-matrix A)))
```

```
                (TR (second (cef-matrix A))))
               (and (integer-matp-dim TR (ncols-m A) (ncols-m A))
                    (equal (matrix-product A TR) H)))))

(defthm inverse-matrix-cef-matrix
   (implies (integer-matp A)
            (let ((TR (second (cef-matrix A)))
                  (TR-INV (cef-matrix-transinv A)))
                 (and (equal (matrix-product TR TR-INV)
                             (matrix-id (ncols-m A)))
                      (equal (matrix-product TR-INV TR)
                             (matrix-id (ncols-m A)))))))
```

In the first of three above theorems, the function `cef-p` is a predicate checking that a matrix is in column echelon form. The result is proved by defining a more general invariant about the form of the matrix during the transformation process; the stopping condition of the algorithm and this invariant implies the theorem.

The second theorem establishes that the second matrix computed by the algorithm is indeed the transformation matrix. This is also an invariant of the process, and note that we have to deal also with the fact that we do the linear combination only until a given row, since from that row on, we have zeros. Additionally, we need to prove the relation between the linear combination of columns carried out by `lin-comb-cols` and the matrix product by the elementary transformation matrix that can be obtained from a lcc operator.

Finally the third theorem establishes that the transformation matrix is invertible, where `(cef-matrix-transinv A)` is a function that obtains the inverse of the transformation matrix computed by the algorithm. We emphasize that the abstract representation is specially convenient, among other reasons, for defining this function and proving the theorem. This is its definition:

```
(defun-nx cef-matrix-transinv (A)
  (let ((res (cef A '(nil nil))))
    (apply-lcc-op-seq
     (inv-lcc-op-seq (second res)) (matrix-id (ncols-m A)))))
```

Given a lcc operator whose coefficients have been obtained as the result of an application of the extended Euclides algorithm, then we can prove that there exists a corresponding lcc operator describing the inverse linear combination (that is, the operator is invertible). Given a sequence of lcc invertible operators, the function `inv-lcc-op-seq`, obtains the reversed sequence of the inverses of each operator. We apply this function to the sequence of operators stored in the second element of the final abstract stobj computed by the algorithm, and then we apply this inverse sequence to the identity matrix. Note that we are taking advantage from the fact that our abstract representation contains the lcc operators explicitly (although our executable concrete representation deals only with the transformation matrix, not with the abstract operators).

For details about the ACL2 proof of these theorems, we urge the interested reader to consult the supporting materials, books `matrices-abstobj-properties.lisp` and `cef-mast.lisp`.

4.3 Experimental Results

To check how this formally verified abstract stobj implementation influences the execution performance of the algorithm, we tested it on several random matrices of different sizes. We compared it to two other implementations of the same algorithm: an analogous unverified ACL2[3] implementation, that uses matrices represented as applicative lists of lists, instead of the abstract stobj; and also an iterative version of the same algorithm in Python 3, using (mutable) lists, which have accesses and updates in constant time. For each size, we generated a number of matrices, and averaged the execution time obtained.

Table 1. Execution times for random matrices

Size	10	20	30	40
List	0.00	0.01	0.54	153.83
Mast	0.00	0.01	0.53	151.96
Python	0.00	0.01	0.59	55.90

Table 2. Execution times for random first column based matrices

Size	160	170	180	190	200
List	32.82	42.07	53.36	65.79	82.62
Mast	0.19	0.23	0.27	0.33	0.38
Python	2.62	3.10	3.60	4.71	5.81

In the Table 1, we show the execution time for random matrices until size 40×40. We see that for sizes below 30×30, the execution times are good for the three implementations. Nevertheless, for sizes 40×40 and bigger, the execution times become unacceptable for both ACL2 implementations, and even for the Python implementation. Nevertheless, we conjecture that the data structures used are not responsible of this slow down: this algorithm and other dealing with integers matrices, usually generate very big numbers. A naive treatment of the arithmetic operations is not enough for dealing with this complexity (and the techniques usually applied [6,12] are out of the scope of this paper).

To concentrate on how the data structures used really influence the execution times, we generate matrices of sizes until 200×200, in which only the first column is random, and the rest of the columns are multiples of the first one. In this way, the arithmetic operations are very straightforward, and the execution times essentially come from accessing and updating the arrays. These execution times are shown in the Table 2. We can see that the applicative ACL2 version is also very slow for that sizes, but the ACL2 abstract stobjs implementation is fast, and even better than the Python implementation.

[3] We used ACL2 Version 7.2 compiled with SBCL 1.2.16.

5 Related Work and Conclusions

We have presented in this paper an approach to formally verify matrix normal form algorithms, while still having efficient data structures for execution. For that, we use the ACL2 system and in particular abstract single-threaded objects, which allow both a convenient logical representation of data and a more efficient concrete representation for execution. We have illustrated this approach showing an ACL2 formal verification of an algorithm to compute echelon forms of integer matrices.

Several formalizations in which matrix algebra plays an important role have been presented in most of theorem provers. For example, using the Coq system [4,8,9] or in Isabelle [2,3]. In all these works, the emphasis is mainly put in the formalization, and in particular they formalize more general results with respect to the algebraic structures involved. In [2] it is also described how to speed-up execution times of the formalized algorithm, first by data type refinements and then by generating code to be executed in a functional programming language. In our case, the approach is different: since ACL2 is built on top of Common Lisp and the logic formalizes an applicative subset of it, we reason directly on the final implementation and execution and reasoning is carried out on the same system.

In addition to stobjs, ACL2 provides 2-dimensional arrays, which under reasonable assumptions provide access in constant time to the entries of the array. This data structures is used in [5] to formalize some common operations and properties of matrices in ACL2. However, the stobj approach is generally more efficient when there are updates [1].

We think abstract stobjs provide a suitable framework for dealing with matrices in ACL2. They provide a clean separation between the data structures used for execution, and the properties of the algorithms that operate on them. In particular, we think the approach shown here for a concrete matrix normalization algorithm can be applied in general to other algorithms that compute normal forms of matrices.

It is worth noting that previous to the introduction of abstract stobjs in ACL2, it was also possible to have a similar formalization strategy: we could have defined two different versions of the algorithm (abstract and concrete, stobj based), prove the main properties of the abstract algorithm and then prove that both versions compute the same results. Now, abstract stobjs provide sound and enhanced support from the system, to carry out this proof strategy: first, we can specify in advance the elementary operations (exports) that will be allowed to operate on the data structures; and second, once introduced, we can concentrate on the abstract definitions, to reason about the properties of the algorithms that use it. A significant downside of the older approach was that one had to prove the correspondence between every newly introduced concrete and abstract function, whereas all such work is done once and for all when using abstract stobjs, thereby easing the maintenance of a formally verified ACL2 implementation.

References

1. ACL2 version 7.4. http://www.cs.utexas.edu/users/moore/acl2/
2. Aransay, J., Divasón, J.: Formalisation in higher-order logic and code generation to functional languages of the Gauss-Jordan algorithm. J. Funct. Program. **25**(9), 1–21 (2015)
3. Aransay, J., Divasón, J.: Formalization of the computation of the echelon form of a matrix in Isabelle/HOL. Form. Asp. Comput. **28**, 1005–1026 (2016)
4. Cano, G., Cohen, C., Dénès, M., Mörtberg, A., Siles, V.: Formalized linear algebra over elementary divisor rings in Coq logical methods in computer. Science **12**(2), 1–29 (2016)
5. Cowles, J., Gamboa, R., Van Baalen, J.: Using ACL2 arrays to formalize matrix algebra. In: Proceedings of ACL2 2003 (2003)
6. Domich, P.D., Kannan, R., Trotter Jr., L.E.: Hermite normal form computation using modulo determinant arithmetic. Math. Oper. Res. **12**, 50–69 (1987)
7. Goel, S., Hunt Jr., W.A., Kaufmann, M.: Abstract stobjs and their application to ISA modeling. In: Proceedings of ACL2 2013, pp. 54–69 (2013)
8. Gonthier, G.: Point-free, set-free concrete linear algebra. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 103–118. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22863-6_10
9. Heras, J., Coquand, T., Mörtberg, A., Siles, V.: Computing persistent homology within Coq/SSReflect. ACM Trans. Comput. Log. **14**(4), 1–26 (2013)
10. Lambán, L., Martín-Mateos, F.-J., Rubio, J., Ruiz-Reina, J.-L.: Towards a verifiable topology of data. In: Proceedings of EACA-2016, pp. 113–116 (2016)
11. Newman, M.: The Smith normal form. Linear Algebra Appl. **254**, 367–381 (1997)
12. Storjohann, A.: Algorithms for matrix canonical forms. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich (2013)