Reasoning about Matrix Arithmetic in ACL2

F. Palomo-Lozano[†], J. A. Alonso-Jiménez[*], I. Medina-Bulo[†]
{francisco.palomo, inmaculada.medina}@uca.es[†]
jalonso@cica.es[*]

Department of Computing Sciences and Artificial Intelligence[*]
University of Sevilla
Department of Computer Languages and Systems[†]
University of Cádiz

Abstract

In this paper we present a formalization of matrix arithmetic in ACL2 adequate for automatic verification, with a high degree of automation, of the fundamental properties of matrices that structure them as a ring. Without loss of generality, we restrict our attention here to square matrices whose dimension is a power of two. The matrix set of elements used includes arbitrary precision complex rational numbers. We also discuss how the nature of the induction schemes involved makes it difficult for an automatic theorem prover to find them.

1 Introduction

In this paper we present a formalization of matrix arithmetic. We restrict our attention here to square matrices over an element field whose dimension is a power of two since this allows us to use a recursive representation for them and it can be done without loss of generality.

On the other hand, this choice is justified by the fact that this representation is in the heart of some important algorithms, like the *Strassen-Pan-Coppersmith-Winograd* family of sub-cubic matrix multiplication algorithms [12, 11, 5].

The formalism used to reason about matrix arithmetic is that of ACL2 [7, 8]. From a logic viewpoint, ACL2 is an untyped quantifier-free first-order logic of total recursive functions with equality. It only contains two *extension principles*. These extension principles allow the introduction of new function symbols and axioms to the logic while preserving its consistency.

The main aim of this work is to provide ACL2 with a reusable *book* of basic matrix operations and theorems about them. These operations are not mere operational abstractions. They are written in an applicative subset of COMMON LISP and, therefore, executable.

We present a formalization that is shown to be adequate for automatic verification, with a high degree of automation, of the fundamental properties of matrices that structure them as a ring.

The main difficulty that has been overcome here is the development of appropriate induction schemes and the completing of some ACL2 properties of number arithmetic. It is worthwhile to note that the nature of the induction schemes involved makes it difficult for an automatic theorem prover to find them.

Since ACL2 is a rule-driven theorem prover, theorems are operationally interpreted as rewrite rules. Therefore, some supplementary theorems that are useful as rewriting rules have also been identified and proved in addition to the basic properties.

Finally, we discuss the development effort and the degree of automation achieved and also analyze some possible extensions of this work, including a brief overview of the problems involved

[*]Facultad de Informática y Estadística. Avda. Reina Mercedes, s/n. 41012 Sevilla. Spain.
[†]Escuela Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz. Spain.

in the generalization of the element field to obtain matrix rings over different algebraic structures and possible ways to overcome them.

2 An Overview of ACL2

ACL2 (*A Computational Logic for Applicative Common Lisp*) is the successor of NQTHM [1, 3], the *Boyer*-Moore theorem prover. A concise description of ACL2 can be found in [6]. In fact, it is necessary to approach ACL2 from three different perspectives to fully understand it.

1. From a logic point of view.
2. From the perspective of programming languages.
3. From the standpoint of automatic reasoning systems.

2.1 ACL2 is a Computational Logic

ACL2 is a first-order quantifier-free logic with equality. Its *syntax* is that of the LISP programming language. This means that a term in the logic is a constant, a variable symbol or the application of a n-ary function symbol (or a λ-expression) to n terms. Formally speaking, predicate symbols in ACL2 do not exist, though Boolean functions play this role.

In ACL2, the set of *axioms* include those of propositional logic with equality and some basic axioms that are needed to work with the usual data types: numbers (integers, rationals and complex rationals)[1], characters, strings, symbols and lists.

On the other hand, *inference rules* are the same that in propositional calculus with equality, adding variable instantiation, induction and functional instantiation. The *induction rule* reduces theorem proofs to finite sets of cases by a powerful form of mathematical induction on ε_0-ordinals. We can replace function symbols in a theorem with other function symbols by using the *functional instantiation rule*.

The logic also includes two *extension principles*: the *definitional principle* and the *encapsulation principle*. The former is essential because it permits the introduction of new function symbols with an axiomatic definition; to preserve consistency, the system only admits a function under this principle if its termination can be guaranteed under certain conditions.[2] The latter permits the introduction of new function symbols constrained by axioms; to preserve consistence, ACL2 requires "witnesses" of the existence of these functions to be exposed. Functional instantiation is an inference rule derived from this extension principle.

The lack of quantification renders ACL2 a constructive logic. Instead of stating the fact that a certain object exists, a function computing an object with the desired properties must be shown. Another remarkable point is the lack of types[3] and of partial functions. Functions admitted under the definitional principle must be total recursive functions (however, see [9].)

2.2 ACL2 is an Applicative Programming Language

Every ACL2 function admitted under the definitional principle is a LISP function. The reciprocal does not hold because the execution of a function must only depend on their arguments if we want to reason about it in ACL2. On the other hand, functions written in conventional programming languages (LISP not being an exception) are not guaranteed to terminate.

Thus, we can think of ACL2 as an *applicative programming language*, that is, a language in which the result of the application of a function is uniquely determined by its arguments. More precisely, ACL2 can be regarded as a side-effect free subset of COMMON LISP.

[1] Recently, an extension to ACL2 has been developed to cope with real number formalization problems.

[2] This guarantees the existence of one, and only one, mathematical function holding the definitional axiom.

[3] However, a primitive type inference system is build into ACL2. The user can help the system to infer types by supplying *type prescription rules*. Types are just used to simplify formulas.

2.3 ACL2 is an Automatic Reasoning System

When you supply a potential theorem to ACL2, or when you extend the logic by using one of the extension principles, it is necessary to check that several conditions hold. Then, ACL2 behaves as a theorem prover.

ACL2 uses several *proof techniques* when trying to prove a theorem. Each proof technique can be viewed as a "process" receiving a formula as its input and producing a set of formulas as its output. The input formula is a theorem *if* each of the output formulas is a theorem.

Of course, a particular process may not apply to a formula. In this case, the output set of the process only consists of that particular formula. On the other hand, if a process proves that a given formula is a theorem then it returns an empty set.

When the user inputs a conjecture into the system, the formula becomes the proof goal and it goes sequentially through every process until one of them applies or some termination conditions are met. When a process is applied, it produces a set of subgoals that replaces the original goal. This procedure is then iterated while there are subgoals pending to be proved.

The *simplification* process includes decision procedures for propositional logic, equality, and *linear arithmetic*. It also deals with term rewriting and metafunctions [2]. This is the only process that may return an empty set of formulas, thus proving that its input formula is a theorem. The term rewriting system plays a fundamental role: axioms, definitions and theorems are stored and used as rewrite rules.

The *destructor elimination* process allows to replace variables affected by destructor operations with a term consisting of a constructor operation and fresh variables. Thus, this eliminates destructor operations to obtain simpler formulas.

The three following processes have a strong heuristic component. The *crossed fertilization* process decides when to use and discard equality hypothesis. The *generalization* process decides when to replace non-variable terms with fresh variables. The *irrelevance elimination* process tries to discard those hypothesis not affecting the validity of the conjecture. All of them are "dangerous" processes, in the sense that a more general conjecture is obtained when discarding an hypothesis or generalizing a term. The generalized conjecture may well not be a theorem even if the original conjecture is a theorem. Its main aim is to prepare the formula for a later induction since, in order to prove a formula by induction, it is not unusual that a generalization of it may be needed.

The last process is *induction*. It tries to find a suitable induction scheme to prove the conjecture. Conjecture terms may suggest several induction schemes, but system heuristics select a unique scheme (perhaps, after merging some of them). If this process does not find a suitable induction scheme, it fails, and ACL2 reports that the conjecture has not been proved.

3 Matrix Representation

The underlying representation of matrices is based on the notion of *weak matrix*. We use record structures for this purpose, which come from COMMON LISP and have been formalized in ACL2 by *Brock* [4]. This provides us with a weak recognizer predicate that we strengthen to develop a recognizer for properly formed matrices.

The set of elements used to define matrices is recognized by an ACL2 predicate that includes arbitrary precision complex rational numbers. Since ACL2 is an untyped programming language, this implies that integers and rationals may well replace complex rationals, without needing a single change.

3.1 Weak matrices

A *structure* is just a convenient way to group and access related data. The `defstructure` facility is a general purpose tool for creating and reasoning about structure specifications.

Our notion of weak matrix is captured by an ACL2 structure. A weak matrix is just a collection of four objects or *slots* called "submatrices". We say that this notion is "weak" because no restrictions are imposed on the types of the elements that can be stored in each submatrix.

The following invocation of `defstructure` defines a constructor operation, `matrix`, and four destructor operations or *readers*, `sub-11`, `sub-12`, `sub-21` and `sub-22`. It also creates an extensive theory for automated reasoning about specifications defined in terms of this structure.

```
(defstructure matrix
  sub-11
  sub-12
  sub-21
  sub-22
  (:options
   (:conc-name nil)
   (:representation :tree)
   (:weak-predicate weak-matrixp)
   (:do-not :tag :read-write :write-write)))))
```

By default, structures are represented as true lists tagged with the structure name. The option `(:representation :tree)` forces a balanced *cons* tree representation of structure terms. This allows $\Theta(\log m)$ access time, where m is the number of slots. The option `(:do-not :tag)` eliminates the name tag from structure terms. The predicate `weak-matrixp` will recognize terms constructed with `matrix`.

3.2 Matrices

A consequence of the weakness of the previous definition is the lack of a uniform representation even if we restrict ourselves to use only weak matrices and numbers in every slot.

We formalize true matrices by defining a recognizer function for square matrices whose dimension is a power of two. At first sight, this may be seen as a restriction, but an arbitrary matrix can always be "completed", at most doubling its size, so that its dimension is a power of two. This is a common choice for several of the most efficient algorithms for dense matrix arithmetic known.

Therefore, we represent a matrix with dimension $n = 2^k$ as a weak matrix of matrices with dimension $n = 2^{k-1}$ if $n \neq 1$, otherwise as a number. As a consequence of the tree structure of weak matrix terms, this definition implies that our matrices have a complete tetrary tree structure of `matrix` operations. The following Boolean function recognizes such a matrix.

```
(defun matrixp (a k)
  (if (zp k)
      (acl2-numberp a)
    (let ((k-1 (- k 1)))
      (and (weak-matrixp a)
           (matrixp (sub-11 a) k-1)
           (matrixp (sub-12 a) k-1)
           (matrixp (sub-21 a) k-1)
           (matrixp (sub-22 a) k-1)))))
```

This recognizer function is admitted by ACL2 without any user assistance. It is a remarkable fact that its admissibility proof can be shortened by specifying the following measure hint:

```
(declare (xargs :measure (acl2-count k)))
```

However, this impacts other subsequent proofs by changing their induction schemes. This would compel us to supply unnecessary induction hints.

We can also prove the following theorems without any effort. They state that the submatrices of a true matrix whose dimension is greater than one are also true matrices (with half the dimension). These theorems are stored as type prescription rules.

```
(defthm matrixp-sub-11
  (implies (and (matrixp a k) (not (zp k)))
```

```
          (matrixp (sub-11 a) (- k 1)))
  :rule-classes :type-prescription)

(defthm matrixp-sub-12
  (implies (and (matrixp a k) (not (zp k)))
           (matrixp (sub-12 a) (- k 1)))
  :rule-classes :type-prescription)

(defthm matrixp-sub-21
  (implies (and (matrixp a k) (not (zp k)))
           (matrixp (sub-21 a) (- k 1)))
  :rule-classes :type-prescription)

(defthm matrixp-sub-22
  (implies (and (matrixp a k) (not (zp k)))
           (matrixp (sub-22 a) (- k 1)))
  :rule-classes :type-prescription)
```

4 Ring Structure

Having selected a representation for matrices, we should show now that it is suitable for devising the usual operations and proving their properties.

The symbols `+m`, `-m`, `*m` will stand for matrix addition, negation and multiplication operations. On the other hand, `null` and `identity` will represent the null and identity matrices, respectively. In order to prevent name conflicts[4], matrix operations and properties are supposed to be defined in a new package, `MATRIX`.

4.1 Operations

The recursive representation chosen produces elegant recursive formulations of common matrix operations. To begin with, the addition of two matrices is accomplished by recursively adding their submatrices pairwise.

```
(defun +m (a b k)
  (if (zp k)
      (+ a b)
    (let ((k-1 (- k 1)))
      (matrix (+m (sub-11 a) (sub-11 b) k-1)
              (+m (sub-12 a) (sub-12 b) k-1)
              (+m (sub-21 a) (sub-21 b) k-1)
              (+m (sub-22 a) (sub-22 b) k-1)))))
```

ACL2 admits this function and also proves that it is a closed operation. Although it is not necessary to supply the induction scheme, it shortens the proof.

```
(defthm matrixp-+m
  (implies (and (matrixp a k) (matrixp b k))
           (matrixp (+m a b k) k))
  :rule-classes :type-prescription
  :hints (("Goal" :induct (+m a b k))))
```

The definition and admission of matrix negation are straightforward. So is the proof of its closure property.

[4] ACL2 defines `null` and `identity` with a different meaning.

```
(defun -m (a k)
  (if (zp k)
      (- a)
    (let ((k-1 (- k 1)))
      (matrix (-m (sub-11 a) k-1) (-m (sub-12 a) k-1)
              (-m (sub-21 a) k-1) (-m (sub-22 a) k-1)))))

(defthm matrixp--m
  (matrixp (-m a k) k)
  :rule-classes :type-prescription)
```

Following this, we can define matrix multiplication in a way that resembles the classic definition of multiplication of two 2×2 matrices. The fact that this is also a closed operation is easily stated.

```
(defun *m (a b k)
  (if (zp k)
      (* a b)
    (let ((k-1 (- k 1))
          (a11 (sub-11 a)) (a12 (sub-12 a))
          (a21 (sub-21 a)) (a22 (sub-22 a))
          (b11 (sub-11 b)) (b12 (sub-12 b))
          (b21 (sub-21 b)) (b22 (sub-22 b)))
      (matrix (+m (*m a11 b11 k-1) (*m a12 b21 k-1) k-1)
              (+m (*m a11 b12 k-1) (*m a12 b22 k-1) k-1)
              (+m (*m a21 b11 k-1) (*m a22 b21 k-1) k-1)
              (+m (*m a21 b12 k-1) (*m a22 b22 k-1) k-1)))))

(defthm matrixp-*m
  (implies (and (matrixp a k) (matrixp b k))
           (matrixp (*m a b k) k))
  :rule-classes :type-prescription)
```

The null matrix of a given dimension is computed from four null submatrices.

```
(defun null (k)
  (if (zp k)
      0
    (let ((null (null (- k 1))))
      (matrix null null null null))))

(defthm matrixp-null
  (matrixp (null k) k)
  :rule-classes :type-prescription)
```

The identity matrix is computed from two identity submatrices and two null submatrices.

```
(defun identity (k)
  (if (zp k)
      1
    (let ((null (null (- k 1)))
          (identity (identity (- k 1))))
      (matrix identity null null identity))))

(defthm matrixp-identity
  (matrixp (identity k) k)
  :rule-classes :type-prescription)
```

4.2 Properties

Having found a proper representation for our notion of matrix and its basic operations, it is time to formally prove that it satisfies the properties that everyone expects from matrices.

Some of these properties require induction hints. The most complex of them are defined separately as functions representing induction schemes and they are discussed in 6.

4.2.1 Ring properties

Associativity of matrix addition requires an induction scheme. On the other hand, commutativity of matrix addition can be stated without any user guidance.

```
(defthm associativity-of-+m
  (equal (+m (+m a b k) c k) (+m a (+m b c k) k))
  :hints (("Goal" :induct (scheme-1 a b c k))))

(defthm commutativity-of-+m
  (equal (+m a b k) (+m b a k)))
```

The null matrix is an identity element of matrix addition. The order in which the theorems are proved allows the second theorem to be reduced to the first one, by using the commutativity theorem previously proved.

```
(defthm null-identity-of-+m-2
  (implies (matrixp a k)
           (equal (+m a (null k) k) a))
  :hints (("Goal" :induct (+m a _ k))))

(defthm null-identity-of-+m-1
  (implies (matrixp a k)
           (equal (+m (null k) a k) a)))
```

The hypothesis `(matrixp a)` is required in both theorems. However, it is not necessary to supply the induction scheme, though it shortens the proof. The scheme suggested by `:induct (+m a _ k)` is based on `+m`, but it does not take into account the second argument (`_` does not appear as a variable in the theorem body).

We can also automatically prove that matrix negation is an inverse of matrix addition.

```
(defthm -m-inverse-of-+m-2
  (equal (+m a (-m a k) k) (null k)))

(defthm -m-inverse-of-+m-1
  (equal (+m (-m a k) a k) (null k)))
```

The order in which the theorems are proved allows the second theorem to be reduced to the first one, by using the commutativity theorem previously proved.

Distributivity of matrix multiplication over matrix addition is proved by using two separate induction schemes. Since matrix multiplication is not commutative, we can not reduce one of the theorems to the other.

```
(defthm distributivity-of-*m-over-+m-1
  (equal (*m a (+m b c k) k)
         (+m (*m a b k) (*m a c k) k))
  :hints (("Goal" :induct (scheme-2 a b c k))))

(defthm distributivity-of-*m-over-+m-2
  (equal (*m (+m a b k) c k)
         (+m (*m a c k) (*m b c k) k))
  :hints (("Goal" :induct (scheme-3 a b c k))))
```

The proof of the associativity of matrix multiplication uses a complex induction scheme. It also requires several of the previous theorems (notably, the distributivity of the multiplication over the addition) and a pair of technical lemmas.

```
(defthm associativity-of-*m
  (equal (*m (*m a b k) c k) (*m a (*m b c k) k))
  :hints (("Goal"
           :induct (scheme-4 a b c k)
           :in-theory (disable commutativity-of-*
                               associativity-of-+m
                               commutativity-of-+m)))))
```

Finally, we must prove that the identity matrix is an identity element of the matrix multiplication operation. The technical lemma `arithmetic-1` acts as a convenient replacement for `unicity-of-0` and `unicity-of-1` axioms in these proofs.

```
(local
  (defthm arithmetic-1
    (implies (acl2-numberp x)
             (and (equal (+ 0 x) x)
                  (equal (* 1 x) x)
                  (equal (* 0 x) 0)))))

(defthm identity-identity-of-*m-1
  (implies (matrixp a k)
           (equal (*m (identity k) a k) a))
  :hints (("Goal"
           :induct (+m a _ k)
           :in-theory (disable unicity-of-0 unicity-of-1))))

(defthm identity-identity-of-*m-2
  (implies (matrixp a k)
           (equal (*m a (identity k) k) a))
  :hints (("Goal"
           :induct (+m a _ k)
           :in-theory (disable unicity-of-0 unicity-of-1))))
```

The hypothesis `(matrixp a k)` is also necessary here. However, it is not necessary to supply the induction schemes, though they shorten the proofs.

4.2.2 Additional properties

An elemental property that we can prove automatically states that matrix negation of a null matrix is also a null matrix.

```
(defthm -m-null-is-null
  (equal (-m (null k) k) (null k)))
```

The following theorems are interesting. They prove that the null matrix is a cancellative element of matrix multiplication.

```
(defthm null-cancellative-of-*m-1
  (equal (*m (null k) a k) (null k))
  :hints (("Goal" :induct (+m a _ k))))

(defthm null-cancellative-of-*m-2
  (equal (*m a (null k) k) (null k))
  :hints (("Goal" :induct (+m a _ k))))
```

As in many other theorems, the induction schemes are not required. However, they slightly shorten the proofs, though, in this particular case, the induction scheme selected by ACL2 is very close to our induction hint.

5 Useful Rewrite Rules

In addition to the usual interpretation of theorems, each theorem can be understood as a (possibly conditional) rewrite rule[5]. This dual character leads to an operational view of theorems as rules. Sometimes, it is useful to prove a theorem just due to its adequacy as a rewrite rule. For example, the following theorem shows that matrix negation is idempotent:

```
(defthm idempotency-of--m
  (implies (matrixp a k)
           (equal (-m (-m a k) k) a))
  :hints (("Goal" :induct (-m a k))))
```

But, in fact, the theorem is stored as a (left to right) rewrite rule once it has been proved. This allows the prover to eliminate consecutive applications of the negation operator during the development of a proof on matrices. In this case, the hypothesis is strictly necessary, that is, this is a conditional rewrite rule. However, it is not necessary to supply the induction scheme, though it shortens the proof.

The distributivity of matrix negation over matrix addition is a fact that the prover states without problems. As a rewrite rule, this allows pushing the negation operator into the addition operator.

```
(defthm distributivity-of--m-over-+m
  (equal (-m (+m a b k) k)
         (+m (-m a k) (-m b k) k)))
```

Another interesting rewrite rule introduce matrix negation inside matrix multiplication. A technical lemma is required to prove the associated theorem. This lemma uses an instance of the distributivity of the multiplication over the addition of numbers[6] and the linear arithmetic decision procedure.

```
(defthm arithmetic-2
  (equal (- (* x y)) (* x (- y)))
  :hints (("Goal"
           :use (:instance distributivity (z (- y))))))

(defthm introduce--m-inside-*m
  (equal (-m (*m a b k) k)
         (*m a (-m b k) k)))
```

Finally, the following rule prevents top-most occurrences of negation operators in the first parameter of a multiplication operator during a proof. It shifts-right matrix negation inside matrix multiplication. Similarly to the previous rule, it uses a simple arithmetic property that requires some trickery to be proved. The induction hint is necessary to complete the proof.

```
(defthm arithmetic-3
  (equal (* (- x) y) (* x (- y)))
  :hints (("Goal"
           :use
           ((:instance distributivity (z (- y)))
            (:instance distributivity (x y) (y x) (z (- x)))))))

(defthm shift--m-inside-*m
  (equal (*m (-m a k) b k)
         (*m a (-m b k) k))
  :hints (("Goal" :induct (*m a b k))))
```

[5] In fact, a theorem may generate several rewrite rules.

[6] ACL2 includes the following `distributivity` axiom: `(equal (* x (+ y z)) (+ (* x y) (* x z)))`.

The combination of these rewrite rules is useful in a certain sense: it allows a kind of "normalization" of the negation operator occurrences in a matrix expression. For example, let us consider three matrices `a`, `b` y `c` and the matrix expression `(-m (+m a (*m (-m b k) c k) k) k)`:

`(-m (+m a` **`(*m (-m b k) c k)`** `k) k)`

reduces by `shift--m-inside-*m` to:

`(-m (+m a (*m b (-m c k) k) k) k)`

reduces by `distributivity-of--m-over-+m` to:

`(+m (-m a k)` **`(-m (*m b (-m c k) k) k)`** `k)`

reduces by `introduce--m-inside-*m` to:

`(+m (-m a k) (*m b` **`(-m (-m c k) k)`** `k) k)`

reduces by `idempotency-of--m` to:

`(+m (-m a k) (*m b c k) k)`

Therefore, the term rewriting system has been able to reduce `(-m (+m a (*m (-m b k) c k) k) k)` to `(+m (-m a k) (*m b c k) k)` by pushing the negation operation deep inside its argument.

6 Induction Schemes

One of the highlights of ACL2 is its ability to guess suitable induction schemes during the development of a proof. Surprisingly, we have found that some of the matrix ring properties presented resist ACL2 heuristic efforts. For example, let us consider the following induction scheme:

```
(defun scheme-1 (a b c k)
  (if (zp k)
      (+ a b c)
    (+ (scheme-1 (sub-11 a) (sub-11 b) (sub-11 c) (- k 1))
       (scheme-1 (sub-12 a) (sub-12 b) (sub-12 c) (- k 1))
       (scheme-1 (sub-21 a) (sub-21 b) (sub-21 c) (- k 1))
       (scheme-1 (sub-22 a) (sub-22 b) (sub-22 c) (- k 1)))))
```

This scheme is used as an induction hint to prove `associativity-of-+m`. The hint suggests that the inductive proof may consist of a base case and an inductive step with four induction hypothesis. It can be interpreted in the following way:

> "Given a property stated on three matrices, we must prove it for matrices having dimension $n = 2^0$ and, in order to prove the property for matrices whose dimension is $n = 2^k$, where $k \neq 0$, we may use the fact that the property holds for certain triplets of submatrices with dimension $\frac{n}{2} = 2^{k-1}$."

As we can see here, this is a sort of multiple structural induction on the arguments. The base case is just a property on numbers since we recognize 1×1 matrices using `acl2-numberp`. But the point is that we need to specify which particular triplets of submatrices are involved in the inductive step.

It is worthwhile to note that ACL2 guarantees the correction of this induction scheme since the function `scheme-1` has to be admitted under the definitional principle before it can be used as a hint. This implies the existence of a strictly decreasing measure in the well-founded domain of ε_0-ordinals.

A similar problem appears when proving `distributivity-of-*m-over-+m-1`. In this case, the induction scheme is more complex:

```
(defun scheme-2 (a b c k)
  (if (zp k)
      (+ a b c)
    (+ (scheme-2 (sub-11 a) (sub-11 b) (sub-11 c) (- k 1))
       (scheme-2 (sub-12 a) (sub-21 b) (sub-21 c) (- k 1))
       (scheme-2 (sub-11 a) (sub-12 b) (sub-12 c) (- k 1))
       (scheme-2 (sub-12 a) (sub-22 b) (sub-22 c) (- k 1))
       (scheme-2 (sub-21 a) (sub-11 b) (sub-11 c) (- k 1))
       (scheme-2 (sub-22 a) (sub-21 b) (sub-21 c) (- k 1))
       (scheme-2 (sub-21 a) (sub-12 b) (sub-12 c) (- k 1))
       (scheme-2 (sub-22 a) (sub-22 b) (sub-22 c) (- k 1)))))
```

Notice that the number of induction hypothesis has increased to 8. A similar scheme (`scheme-3`) is used to prove `distributivity-of-*m-over-+m-2`. The proof of `associativity-of-*m` is the most complex obtained proof: it requires a scheme (`scheme-4`) with 16 induction hypothesis. For the sake of brevity, we omit these two schemes.

In fact, we can generalize all these induction schemes to obtain a single induction scheme that is valid to prove all these properties. Nevertheless, this is a rather complex scheme[7] consisting of 26 induction hypothesis.

7 Conclusions and Further Work

There are many applications of matrix and polynomial arithmetic ranging from DSP to computer graphics and CAD. Therefore, it is important that basic libraries of algorithms and theorems on these structures are available. In this sense, our work is complementary to [10], where a formalization of basic polynomial arithmetic is presented. Both works include arithmetic operations, ring properties and some useful operational rules.

We think that the representation issues are the key to obtain clear statements of the properties to be proved. The formalization that we have shown is suitable for operating with dense matrices and the degree of automation achieved is acceptable. We had to devise several induction schemes, but eventually we realized that four of them could be merged into one scheme. Some technical lemmas on basic arithmetic properties of numbers were also required during the proofs.

Although it is difficult to give a precise measure of the development effort, we estimate it in 0.5 man-month. This is still far away from typical programming efforts for similar projects. By using formal certification, we can assure correctness in exchange for this effort. This is clearly a benefit when developing algorithms for critical systems.

All the *events* (functions and theorems) have been collected in an ACL2 *book* to increase their reusability. Nevertheless, we are working in abstracting the set of elements to obtain matrices over arbitrary (non-commutative) rings. This can be achieved by using the encapsulation principle to constrain element operations to the desired properties. Later, functional instantiation can be used to obtain concrete implementations. However, there is a potential problem: once the set of elements has been abstracted, we can not use the linear arithmetic decision procedure in our proofs any more. Thus, linear arithmetic must be replaced with ad hoc properties.

This work does not include *guard verification*, though it is an interesting extension. ACL2's support for guards allows us to indicate function preconditions and verify that the corresponding COMMON LISP code can be executed with the same results on any platform.

It is also necessary in many applications to include efficient algorithms in order to manipulate high dimension matrices. Fast multiplication and exponentiation algorithms are our two main goals. "Divide and conquer" provides the necessary algorithmic techniques.

A well-known fast exponentiation algorithm can be obtained by binary reduction. On the other hand, *Strassen-Pan-Coppersmith-Winograd* family of sub-cubic matrix multiplication algorithms includes the asymptotically most efficient multiplication algorithms known. It is true that even

[7] We have noticed a considerable increase of the proof time.

the original *Strassen*'s algorithm is of limited practical interest due to an important increase of hidden constants. Nevertheless, its mechanical verification is a challenging problem.

Acknowledgments

B. Brock developed the `defstructure` facility for ACL2. We have found this tool specially well-documented and reusable.

References

[1] *Boyer, R. S. & Moore, J S. A Computational Logic.* Academic Press. 1978.

[2] *Boyer, R. S. & Moore, J S. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.* In *The Correctness Problem in Computer Science.* (R. S. Boyer and J S. Moore eds.) Academic Press. 1981.

[3] *Boyer, R. S. & Moore, J S. A Computational Logic Handbook.* Academic Press. 2nd ed. 1998.

[4] *Brock, B.* `defstructure` *for ACL2 Version 2.0.* Computational Logic, Inc. 1997.

[5] *Coppersmith, D. & Winograd, S. Matrix Multiplication via Arithmetic Progressions.* Journal of Symbolic Computation. Vol. 9. 1990.

[6] *Kaufmann, M. & Moore, J S. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp.* IEEE Transactions on Software Engineering. 1997.

[7] *Kaufmann, M.; Manolios, P. & Moore, J S. Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers. 2000.

[8] *Kaufmann, M.; Manolios, P. & Moore, J S.* (eds.) *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers. 2000.

[9] *Manolios, P. & Moore J S. Partial Functions in ACL2.* ACL2 Workshop 2000.

[10] *Medina-Bulo, I.; Alonso-Jiménez, J. A. & Palomo-Lozano, F. Automatic Verification of Polynomial Rings Fundamental Properties in ACL2.* ACL2 Workshop 2000.

[11] *Pan, V. Strassen's Algorithm is not Optimal.* 19th Annual IEEE Symposium on the Foundations of Computer Science. 1978.

[12] *Strassen, V. Gaussian Elimination is not Optimal.* Numerische Mathematik. Vol. 13. 1969.