# Certification of Matrix Multiplication Algorithms
## Strassen's Algorithm in ACL2

Francisco Palomo-Lozano[1], Inmaculada Medina-Bulo[1], and
José A. Alonso-Jiménez[2]

[1] Department of Computer Languages and Systems. University of Cádiz.
Esc. Superior de Ingeniería de Cádiz. C/ Chile, s/n. 11003 Cádiz. Spain.
{francisco.palomo, inmaculada.medina}@uca.es
[2] Department of Comp. Sciences and Artificial Intelligence. University of Sevilla.
Fac. de Informática y Estadística. Avda. Reina Mercedes, s/n. 41012 Sevilla. Spain.
jalonso@cica.es

**Abstract.** In this work we develop a formalization of matrix arithmetic in ACL2 adequate for the specification and certification, with a high degree of automation, of Strassen's matrix multiplication algorithm. We restrict our attention here to square matrices whose dimension is a power of two. The fundamental properties of this matrix ring have been proved and some experimental results on execution times, including the determination of the optimal threshold, are presented. We also discuss the nature of the induction schemes involved in the proofs.

## 1 Introduction

In this paper we present a formalization of matrix arithmetic. We restrict our attention here to square matrices over the field of complex rational numbers. We also require that their dimensions are a power of two since this allows us to use a convenient recursive representation for them. From a theoretical viewpoint, this can be done without loss of generality since matrices can be completed to accomplish this.

On the other hand, our choice is justified by the fact that this representation is in the heart of some important algorithms belonging to the Strassen-Pan-Coppersmith-Winograd [14, 15, 13, 5] family of sub-cubic matrix multiplication algorithms.

In particular, we formalize here Strassen's algorithm which we have certified by proving its equivalence to the classical algorithm. This was the first sub-cubic algorithm for matrix multiplication known and its discovery in 1969 by V. Strassen [14] broke a long standing belief about the cubic complexity of matrix multiplication and some important related problems.

As far as we know, general matrices have been formalized in the Mizar system [6, 7] where matrix multiplication is defined in the usual way.

The formalism we use to reason about matrix arithmetic is that of ACL2 [9, 10]. 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.

Our main aim has been to provide ACL2 with a reusable *book* of basic matrix operations and theorems about them. But, at the same time, these operations are not devised as mere operational abstractions. They are written in an applicative subset of COMMON LISP, and therefore they are executable.

Since ACL2 is a rule-driven theorem prover, theorems are operationally interpreted as rewrite rules. Therefore, some supplementary theorems that are useful as rewrite rules have been identified and proved in addition to the basic properties. These rewrite rules play a major role in proving the correctness of Strassen's algorithm.

We have also carried out some experiments. The results obtained from them allow us to compare the performance of the classical multiplication algorithm and Strassen's algorithm under ACL2. The empirical optimal threshold is also obtained for the latter algorithm.

Finally, we discuss the degree of automation achieved and we also analyze some possible extensions of this work, including a brief overview of the problems involved in the generalization of the element field to obtain matrix rings over different algebraic structures.

## 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 [8]. In fact, it is necessary to approach ACL2 from three different perspectives to fully understand it.

### 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 $\lambda$-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

[1] An extension of ACL2 has been developed to cope with real number formalization problems.

powerful form of mathematical induction on $\varepsilon_0$-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 [11].)

### 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.

### 2.3 ACL2 is an Automated 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

[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.

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 hypotheses. The *generalization* process decides when to replace non-variable terms with fresh variables. The *irrelevance elimination* process tries to discard those hypotheses 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.

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, `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 (see [4]). The predicate `weak-matrixp` will recognize terms constructed with `matrix`.

```
(defstructure matrix
  sub-11 sub-12
  sub-21 sub-22
  (:options
   (:conc-name nil)
   (:weak-predicate weak-matrixp)))
```

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. This definition implies that our matrices have a complete tetrary tree structure of `matrix` constructors. 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))))))
```

ACL2 functions are untyped and must be total. As a consequence, if we want `matrixp` to be admitted under the definitional principle then it must provide an answer even in the case that $k$ is not a natural number. From a logical viewpoint, `(zp k)` $\equiv k \notin \mathbb{N} \vee k = 0$. Thus we can say that non-natural values of $k$ are simply regarded as 0 by `matrixp`.

This recognizer function is admitted by ACL2 without any user assistance. We can also prove theorems stating that the submatrices of a true matrix whose

dimension is greater than one are also true matrices (with half the dimension). This is done without any effort. For example:

```
(defthm matrixp-sub-11
  (implies (and (matrixp a k) (not (zp k)))
           (matrixp (sub-11 a) (- k 1))))
```

# 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`, `-s`, `*m` will stand for matrix addition, negation, subtraction and multiplication operations. On the other hand, `null` and `identity` will represent the null and identity matrices, respectively.

## 4.1 Operations

The recursive representation chosen produces elegant recursive formulations of the usual 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)))))
```

The definition and admission of matrix negation and matrix subtraction are straightforward.

```
(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)))))

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

Following this style, we can define matrix multiplication in a way that resembles the classical method of multiplying two $2 \times 2$ matrices.

```
(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)))))
```

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

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

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

ACL2 admits these functions under its definitional principle automatically. It also proves that they are closed operations without any assistance. For example:

```
(defthm matrixp-*m
  (matrixp (*m a b k) k))
```

### 4.2 Guard Verification and Execution

Although ACL2 assigns a logical meaning to every expression, COMMON LISP functions are partial in nature. ACL2 provides "guards" as a mean of specifying the intended domain of functions. This allows us to indicate function preconditions as predicates. We have used the following guard specifications in our matrix operations:

1. `(naturalp k)` for nullary operations.
2. `(and (naturalp k) (matrixp a k))` for unary operations.

3. `(and (naturalp k) (matrixp a k) (matrixp b k))` for binary operations.

Guards are extralogical artifacts. They are not part of the definitions of functions nor affect the proofs of theorems where guarded functions are used. Guards are related to execution.

Having verified guards we can assure that our functions are executable in their guarded domains. Functions will be evaluated by COMMON LISP without any execution error (provided there are enough computational resources available) whenever their inputs satisfy their guards. Moreover, they will produce the same results on any COMMON LISP compliant platform.

### 4.3 Ring Properties

In order to be proved by ACL2, some of the ring properties require induction hints. The most complex of these induction schemes are defined separately and they are discussed in Sect. 6. Furthermore, some proofs can be shortened by specifying more suitable induction schemes than those selected automatically by the system, though we will not discuss this aspect here.

Most properties can be proved as unconditional theorems. However, a few of them require hypotheses.

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

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

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

The null matrix is shown to be 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)))

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

We can also automatically prove that matrix negation is an inverse of matrix addition. Again, proof order allows the second theorem to be reduced to the first one.

```
(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)))
```

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)))

(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)))
```

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).

```
(defthm associativity-of-*m
  (equal (*m (*m a b k) c k) (*m a (*m b c k) k)))
```

Finally, we must prove that the identity matrix is an identity element of the matrix multiplication operation.

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

(defthm identity-identity-of-*m-2
  (implies (matrixp a k)
           (equal (*m a (identity k) k) a)))
```

### 4.4 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 also 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)))

(defthm null-cancellative-of-*m-2
  (equal (*m a (null k) k) (null k)))
```

## 5 Useful Rewrite Rules

In addition to the usual interpretation of theorems, each theorem can be understood as a (possibly conditional) rewrite rule. This dual character leads to an operational view of theorems as rules. As a rewrite rule, the following theorem promotes the elimination of `(-s` *a b k*`)` in favor of `(+m` *a* `(-m` *b k*`)` *k*`)` for arbitrary *a*, *b* and *k* terms:

```
(defthm -s->-m
  (equal (-s a b k) (+m a (-m b k) k)))
```

The next theorem states that matrix negation is idempotent. This allows the prover to eliminate consecutive applications of the negation operator during the development of a proof on matrices. In this case, it needs an hypothesis, that is, we obtain a conditional rewrite rule.

```
(defthm idempotency-of--m
  (implies (matrixp a k)
           (equal (-m (-m a k) k) a)))
```

The distributivity of matrix negation over matrix addition is a fact that the prover states without any problem. As a rewrite rule, this allows to push 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 introduces 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[4] and the linear arithmetic decision procedure.

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

Finally, the following rule prevents topmost 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. An induction hint is necessary to complete the proof.

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

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.

[4] ACL2 includes the following distributivity axiom which is suitable for numbers: `(equal (* x (+ y z)) (+ (* x y) (* x z)))`.

## 6 Induction Schemes

One of the highlights of ACL2 is its ability to guess suitable induction schemes during the development of a proof. We have found that some of the matrix ring properties presented resist ACL2 heuristic efforts. For example, let us consider the following induction scheme:[5]

```
(defsch associativity-of-+m-scheme (a b c)
  (sub-11 a) (sub-11 b) (sub-11 c)
  (sub-12 a) (sub-12 b) (sub-12 c)
  (sub-21 a) (sub-21 b) (sub-21 c)
  (sub-22 a) (sub-22 b) (sub-22 c))
```

This scheme is used as an induction hint to prove `associativity-of-+m`. ACL2 guarantees its correction since it has to be admitted under the definitional principle before it can be used. The hint suggests that the inductive proof may consist of a base case and an inductive step with four induction hypotheses. 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 $k \in \mathbb{N}^*$, we may use the fact that it 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 $1 \times 1$ matrices using `acl2-numberp`. But the point is that we need to specify which particular triplets of submatrices are involved in the inductive step. That is why we include four triplets to prove the associativity of matrix addition, one for each pair of entries in the following matrix identity:

$$\underbrace{\begin{bmatrix}(a_{11}+b_{11})+c_{11} & (a_{12}+b_{12})+c_{12}\\ (a_{21}+b_{21})+c_{21} & (a_{22}+b_{22})+c_{22}\end{bmatrix}}_{(A+B)+C} = \underbrace{\begin{bmatrix}a_{11}+(b_{11}+c_{11}) & a_{12}+(b_{12}+c_{12})\\ a_{21}+(b_{21}+c_{21}) & a_{22}+(b_{22}+c_{22})\end{bmatrix}}_{A+(B+C)}$$

A similar problem appears with `distributivity-of-*m-over-+m-1`. Here, the induction scheme is more complex and the number of hypotheses increases to 8. An analogous scheme is used with `distributivity-of-*m-over-+m-2`. The proof of `associativity-of-*m` is the most complex obtained proof: it requires 16 induction hypotheses. For the sake of brevity, we will omit these schemes.

In fact, we can merge all these induction schemes to obtain a single induction scheme that is valid to prove the four properties. Nevertheless, this is a rather complex scheme[6] consisting of 26 induction hypotheses. We arrive at them by joining the triplets of the four induction schemes and discarding duplicated triplets.

[5] ACL2 does not include `defsch`. This tool has been developed by the authors to add syntactic sugar to the specification of structural induction schemes for matrix tuples.

[6] We have noticed a considerable increase of proof times when using this merged induction scheme.

## 7 Strassen's Algorithm

Next, we present a translation of Strassen's algorithm to our matrix formalization. Let us consider `k0` as an arbitrary threshold (the algorithm switch to the classical algorithm when the input matrices are sufficiently small). We defer the suitable selection of this threshold to Sect. 8.

```
(defun *s (a b k)
  (if (zp k)
      (* a b)
    (if  (<= k (k0))
        (*m a b k)
      (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))
             (m1 (*s (-s a12 a22 k-1) (+m b21 b22 k-1) k-1))
             (m2 (*s (+m a11 a22 k-1) (+m b11 b22 k-1) k-1))
             (m3 (*s (-s a11 a21 k-1) (+m b11 b12 k-1) k-1))
             (m4 (*s (+m a11 a12 k-1) b22 k-1))
             (m5 (*s a11 (-s b12 b22 k-1) k-1))
             (m6 (*s a22 (-s b21 b11 k-1) k-1))
             (m7 (*s (+m a21 a22 k-1) b11 k-1))
             (c11 (+m (-s (+m m1 m2 k-1) m4 k-1) m6 k-1))
             (c12 (+m m4 m5 k-1))
             (c21 (+m m6 m7 k-1))
             (c22 (-s (+m (-s m2 m3 k-1) m5 k-1) m7 k-1)))
        (matrix c11 c12 c21 c22)))))
```

In order to certify its correctness, we take the classical algorithm as our specification reference and try to prove the following equivalence theorem:

```
(defthm *s<->*m
  (implies (and (matrixp a k) (matrixp b k))
           (equal (*s a b k) (*m a b k))))
```

Obviously, this theorem fails because the system lacks of enough knowledge on matrix arithmetic. Ideally, we would like to build a decision procedure for matrix arithmetic in ACL2, perhaps using metafunctions, but this is not an easy task. However, we can overcome this problem by using suitable rewrite rules. Let us consider matrix expressions appearing in Strassen's algorithm.

1. Subtraction can be eliminated in favor of addition and negation by `-s->-m`.
2. Subtraction-free expressions can be reduced to addition chains of multiplicative terms by applying the rules `distributivity-of-*m-over-+m-1`, `distributivity-of-*m-over-+m-2`, `distributivity-of--m-over-+m` and `shift--m-inside-*m`.
3. Negation can be put in front of multiplicative terms by:

```
(defthm extract--m-from-*m
  (equal (*m a (-m b k) k) (-m (*m a b k) k)))
```

4. Negative terms can be lifted up in each addition chain by the second and third rules generated by the following theorem:

```
(defthm restricted-associativity-commutativity-of-+m
  (and (equal (+m (*m a1 a2 k) (+m (*m b1 b2 k) c k) k)
              (+m (*m b1 b2 k) (+m (*m a1 a2 k) c k) k))
       (equal (+m (*m a1 a2 k) (+m (-m b k) c k) k)
              (+m (-m b k) (+m (*m a1 a2 k) c k) k))
       (equal (+m (*m a1 a2 k) (+m b (-m c k) k) k)
              (+m (-m c k) (+m (*m a1 a2 k) b k) k))))
```

5. Each equality appearing in the proof of the equivalence theorem can contain negative topmost terms in its left side. The following rule moves these terms from the left side to the right side, thus eliminating negations:

```
(defthm cancel-leftmost--m-in-equal
  (implies (and (matrixp a k) (matrixp b k) (matrixp c k))
           (iff (equal (+m (-m a k) b k) c) (equal b (+m a c k)))))
```

6. The resulting equalities of addition chains of multiplicative terms are free of negative terms, thus they can be decided by right-rotating the expression trees with `associativity-of-+m`, `commutativity-of-+m` and the first rule of `restricted-associativity-commutativity-of-+m`.

Let $+_k$ , $-_k$, $\cdot_k$ be infix operators representing `+m`, `-s` and `*m`, respectively. The rewriting strategy that we have presented is used to prove automatically that, if $a_{ij}$, $b_{ij}$ $(i, j \in \{1, 2\})$ are matrices of dimension $2^k$, and

$$
\begin{aligned}
m_1 &= (a_{12} -_k a_{22}) \cdot_k (b_{21} +_k b_{22}) & \qquad c_{11} &= ((m_1 +_k m_2) -_k m_4) +_k m_6 \\
m_2 &= (a_{11} +_k a_{22}) \cdot_k (b_{11} +_k b_{22}) & \qquad c_{12} &= m_4 +_k m_5 \\
m_3 &= (a_{11} -_k a_{21}) \cdot_k (b_{11} +_k b_{12}) & \qquad c_{21} &= m_6 +_k m_7 \\
m_4 &= (a_{11} +_k a_{12}) \cdot_k b_{22} & \qquad c_{22} &= ((m_2 -_k m_3) +_k m_5) -_k m_7 \\
m_5 &= a_{11} \cdot_k (b_{12} -_k b_{22}) \\
m_6 &= a_{22} \cdot_k (b_{21} -_k b_{11}) \\
m_7 &= (a_{21} +_k a_{22}) \cdot_k b_{11}
\end{aligned}
$$

then:

$$
\begin{aligned}
c_{11} &= a_{11} \cdot_k b_{11} +_k a_{12} \cdot_k b_{21} \\
c_{12} &= a_{11} \cdot_k b_{12} +_k a_{12} \cdot_k b_{22} \\
c_{21} &= a_{21} \cdot_k b_{11} +_k a_{22} \cdot_k b_{21} \\
c_{22} &= a_{21} \cdot_k b_{12} +_k a_{22} \cdot_k b_{22} \quad .
\end{aligned}
$$

ACL2 develops a successful inductive proof of the equivalence theorem once this helper lemma has been proved. Let $p(a, b, k)$ denote the property stating that both multiplication functions, `*s` and `*m`, produce the same result when

their inputs, $a$ and $b$, are matrices of dimension $2^k$. ACL2 inducts according to a scheme suggested by `(*s a b k)`. If we represent the submatrices of $a$ and $b$ by $a_{ij}$, $b_{ij}$ $(i, j \in \{1, 2\})$ then the proof sketch is:

1. Base case: there are two trivial subcases ($k \notin \mathbb{N} \vee k = 0$ and $k \in \mathbb{N}^* \wedge k \leq k_0$), the proof proceeds by the expansion of the functions involved and simplification.
2. Inductive step ($k \in \mathbb{N}^* \wedge k > k_0$): the proof is reduced to

$$\begin{aligned} p(a_{12} -_{k-1} a_{22}, b_{21} +_{k-1} b_{22}) \wedge p(a_{11} +_{k-1} a_{22}, b_{11} +_{k-1} b_{22}) \wedge \\ p(a_{11} -_{k-1} a_{21}, b_{11} +_{k-1} b_{12}) \wedge p(a_{11} +_{k-1} a_{12}, b_{22} \wedge \\ p(a_{11}, b_{12} -_{k-1} b_{22}) \wedge p(a_{22}, b_{21} -_{k-1} b_{11}) \wedge \\ p(a_{21} +_{k-1} a_{22}, b_{11}) \implies p(a, b, k) \end{aligned}$$

   and this is proved by the expansion of the multiplication functions and simplification with the helper lemma.

## 8 Execution Times and Optimal Thresholds

It is a known fact that the threshold choice can have a deep impact in the execution times of Strassen's algorithm (this is usual when working by "divide and conquer"). Although the value of the threshold does not affect the asymptotic growth rate, which is $\Theta(n^{\log_2 7})$, it affects the hidden constants in the asymptotic notation.

In theory, optimal thresholds can be analytically computed, but a theoretical optimal threshold should only be regarded as an estimate. In real life, there are many factors that can deviate an optimal threshold from its theoretical location.

We have determined that the empirical optimal threshold for Strassen's algorithm is $k_0 = 5$ in our ACL2 implementation, as we can see in Table 1. Deviation from this optimal threshold has dramatical consequences. Time is measured in CPU seconds of a particular computer using the most recent version of ACL2 to date.[7]

## 9 Conclusions and Further Work

We have presented here a correctness proof of Strassen's matrix multiplication algorithm with arbitrary threshold for square matrices whose dimension is a power of two. This result is by no means trivial and it is a testimonial to the high level of automation that can be reached in ACL2 when a proper formalization is used.

There are many applications of matrix and polynomial arithmetic ranging from DSP to computer graphics and CAD. Therefore, we think that it is important that basic libraries of algorithms and theorems on these structures are

[7] We have used an Intel Pentium III 600 MHz, 128 MB SDRAM 133 MHz under ACL2 2.5/GCL 2.3.

**Table 1.** Influence of the threshold in Strassen's algorithm (time in s)

| $k$ | $k_0 =$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | | 0.01 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.00 | 0.00 |
| 5 | | 0.12 | 0.11 | 0.10 | 0.03 | 0.03 | 0.03 | 0.03 | 0.02 |
| 6 | | 0.44 | 0.38 | 0.32 | 0.28 | 0.26 | 0.27 | 0.30 | 0.29 |
| 7 | | 3.52 | 3.10 | 2.35 | 2.02 | 1.96 | 1.99 | 2.13 | 2.30 |
| 8 | | 31.21 | 29.12 | 22.70 | 19.06 | 17.91 | 18.02 | 19.11 | 20.19 |
| 9 | | 198.49 | 182.76 | 143.92 | 119.27 | 119.14 | 114.81 | 129.28 | 140.42 |

available. In this sense, our work is complementary to [12], 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 high dimension dense matrices and the degree of automation achieved is quite acceptable. We had to devise several induction schemes, though eventually we realized that four of them could be merged into one scheme. The rewriting strategy has also played an important role.

Some technical lemmas on basic arithmetic properties of numbers were also required during the proofs, though it is possible to eliminate them by including a book on number arithmetic from the standard ACL2 distribution.

All the functions and theorems have been collected in ACL2 books to increase their reusability. As an application, a non-trivial matrix multiplication algorithm, Strassen's algorithm, has been certified. There are algorithms more efficient asymptotically but they are of limited practical interest due to an important increase of the hidden constants.

It is difficult to give a precise measure of the development effort, but we think that it is still far away from typical programming efforts for similar projects. On the other hand, by using formal certification we can assure correctness in exchange for this effort.

Winograd's algorithm is a variant of Strassen's algorithm that reduces the number of matrix additions/subtractions from 18 to 15 by a clever bookkeeping of common subexpressions. Although we have omitted it here, this algorithm has been also certified: an analogous equivalence theorem has been proved.

Fast matrix exponentiation is one of our goals. Again, "divide and conquer" provides the necessary algorithmic techniques. A well-known fast exponentiation algorithm can be obtained by binary reduction. This can be combined with Strassen's algorithm to obtain a fast matrix exponentiation algorithm.

We are also 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, 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.

A rather complex work to be considered would be the obtainment of an efficient formalization for arbitrary dimension matrices akin to the formalization presented here, but this would require a totally different approach.

## 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: Boyer, R. S., Moore, J S. (eds.): The Correctness Problem in Computer Science. Academic Press (1981)
3. Boyer, R. S., Moore, J S.: A Computational Logic Handbook. Academic Press. $2^{nd}$ edn. (1998)
4. Brock, B.: `defstructure` for ACL2. Computational Logic, Inc. (1997)
5. Coppersmith, D., Winograd, S.: Matrix Multiplication via Arithmetic Progressions. J. Symbolic Computation **9** (1990)
6. Jankowska, K.: Matrices. Abelian Group of Matrices. J. Formalized Mathematics **3** (1991)
7. Jankowska, K.: The Product and the Determinant of Matrices with Entries in a Field. J. Formalized Mathematics **5** (1993)
8. Kaufmann, M., Moore, J S.: An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. IEEE Trans. on Software Engineering **23**(4) (1997)
9. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (2000)
10. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
11. Manolios, P., Moore J S.: Partial Functions in ACL2. ACL2 Workshop 2000 (2000)
12. Medina-Bulo, I., Alonso-Jiménez, J. A., Palomo-Lozano, F.: Automatic Verification of Polynomial Rings Fundamental Properties in ACL2. ACL2 Workshop 2000 (2000)
13. Pan, V.: Strassen's Algorithm is not Optimal. $19^{th}$ Annual IEEE Symposium on the Foundations of Computer Science (1978)
14. Strassen, V.: Gaussian Elimination is not Optimal. Numerische Mathematik **13** (1969)
15. Winograd, S.: Some Remarks on Fast Multiplication of Polynomials. In: Traub, J. F. (ed.): Complexity of Sequential and Parallel Numerical Algorithms. Academic Press (1973)