Using ACL2 Arrays to Formalize Matrix Algebra

Ruben Gamboa[1] John Cowles[1] Jeff Van Baalen[1]
Computer Science Department
University of Wyoming
{ruben,cowles,jvb}@cs.uwyo.edu

Abstract

An ACL2 book formalizing matrix algebra is described. The formalization implements matrices with ACL2 two dimensional arrays.

The newly proposed ACL2 macro `mbt`[2] ("must be true") helps to cleanly separate logical considerations, in definitions, from concerns about efficient execution. This use of `mbt` was suggested by an anonymous referee, whom we thank.

Matrix Algebra.

Let p and q be positive integers. A $p \times q$ matrix M is a rectangular array of numbers, with p rows and q columns,

$$M = \begin{pmatrix} m_{1\,1} & \cdots & m_{1\,q} \\ \vdots & \ddots & \vdots \\ m_{p\,1} & \cdots & m_{p\,q} \end{pmatrix}.$$

It is assumed that the reader already knows a great deal about the algebraic operations on such matrices. Here is a brief summary.

- The sum of two $p \times q$ matrices is a $p \times q$ matrix.
- The product of a $p \times q$ matrix and a $q \times r$ matrix is a $p \times r$ matrix.
- The scalar product of a number and a $p \times q$ matrix is a $p \times q$ matrix.
- The transpose of a $p \times q$ matrix is a $q \times p$ matrix.
- Matrix addition and multiplication are associative.

[1]This work supported by NASA grant NAG 2-1570.
[2]`mbt` is a proposed feature of ACL2 Version 2.8.

- Matrix addition is commutative, but matrix multiplication need not be commutative.
- Matrix and scalar multiplication distribute over matrix addition.
- There is an unique $p \times q$ zero matrix $\mathbf{0}$ such that $M + \mathbf{0} = M = \mathbf{0} + M$.
- For square matrices, there is an unique $p \times p$ identity matrix $\mathbf{I}$ such that $M \cdot \mathbf{I} = M = \mathbf{I} \cdot M$.
- Every $p \times q$ matrix has an unique $p \times q$ negative matrix such that $M + (-M) = \mathbf{0} = (-M) + M$.
- Some square matrices, called nonsingular, have unique (multiplicative) inverses such that $M \cdot M^{-1} = \mathbf{I} = M^{-1} \cdot M$.
- If $M_1 \cdot M_2 = \mathbf{0}$, then neither M_1 nor M_2 need be $\mathbf{0}$.

ACL2 arrays.

ACL2 provides functions for accessing and updating both one and two dimensional arrays, with applicative semantics, but good access time to the most recently updated copy and usually constant update time.

¿From the point of view of the applicative semantics, arrays are implemented in the usual way one would represent "sparse" arrays. An array is simply an alist, i.e. a list of pairs. One element of the alist is the "header," which contains the number of rows, d_1, the number of columns, d_2, and a default value. Aside from the header, the other elements of the alist must each be of the form $((i \; . \; j) \; . \; \mathsf{val})$, where i and j are integers with $0 \leq i < d_1$ and $0 \leq j < d_2$, and val is an arbitrary object. Formally speaking, to access the value indexed by the pair $(i \; . \; j)$ in such an alist, the alist is searched (with the function aref2) for the first pair whose car matches the pair $(i \; . \; j)$. If such a pair is found, then aref2 returns the cdr of the pair; otherwise aref2 returns the default value stored in the header.

Fast array accesses are made possible by maintaining, behind the scenes, a "real" Common Lisp array that may currently represent the given alist. In that case, an array access can be very fast because the real array may be accessed directly.

It would seem to be natural and straight forward to implement matrices using ACL2 two dimensional arrays. However, there is a complication, that will now be explained.

Ensuring closure of matrix multiplication.

Let d_1 be the number of rows and d_2 be the number of columns in an ACL2 two dimensional array. The product, $d_1 \cdot d_2$, is required to fit into 32 bits so that some compilers can lay down faster code. Thus,

$$\begin{aligned} d_1 \cdot d_2 &< \textsf{maximum-positive-32-bit-integer} \\ &= 2^{31} - 1 \\ &= 2,147,483,647. \end{aligned}$$

This restriction on the size of $d_1 \cdot d_2$ means that matrices representable by ACL2 arrays are not closed under matrix multiplication, even when the product is mathematically defined. To illustrate, suppose $d_1 \cdot d_2$ is required to be no larger than 20; M_1 is a matrix with 5 rows and 2 columns; and M_2 is a matrix with 2 rows and 5 columns. Then M_1 and M_2 would both be representable and their product, $M_1 \cdot M_2$, would be mathematically defined, but not representable (since $25 > 20$).

Furthermore, when there are more than two matrices involved in a matrix multiplication, the final product may be both mathematically defined and representable by an ACL2 array, but yet not computable in ACL2. Let's illustrate by extending the example given above with M_1 and M_2. Suppose M_0 is a matrix with 2 rows and 5 columns. Then the product $(M_0 \cdot M_1) \cdot M2$ is mathematically defined, representable in ACL2, and computable in ACL2 (since both partial products $(M_0 \cdot M_1)$ and $(M_0 \cdot M_1) \cdot M2$ are representable in ACL2). But the product $M_0 \cdot (M_1 \cdot M2)$ is mathematically defined, representable in ACL2, but not computable in ACL2 (since the partial product $(M_1 \cdot M_2)$ is not representable in ACL2).

One way to prevent this last problem and also ensure closure for matrix multiplication is to require that each of d_1 and d_2 be less than or equal to 46,340 which is the integer square root of 2,147,483,647, the maximum-positive-32-bit-integer. Then the product of $d_1 \cdot d_2$ is guaranteed to be less than the maximum-positive-32-bit-integer. Furthermore, with this stronger restriction, if the product $M_1 \cdot \cdots \cdot M_n$ is both mathematically defined and representable in ACL2, then, for any way of parenthesizing this product, all the partial products are also mathematically defined and representable in ACL2.

Thus, for matrix multiplication, it is required that both the number of rows and the number of columns be less than or equal to 46,340.

Comments on Matrix Operations in ACL2.

1. It's useful to distinguish among three versions of "two dimensional arrays."

   Logical or "slow" array. This is an alist with the "shape" of a two dimensional array. The alist meets a specified minimal number of conditions, required by the applicative semantics, to be a two dimensional array. In particular, there are no restrictions on the upper-bounds of the number of rows and the number of columns.

   "Fast" executable array. This is a logical array that meets the additional restriction required to ensure fast accessing and updating: The product, of the number of rows and the number of columns, is less than the maximum-positive-32-bit-integer.

Matrix. This is a fast array that meets an additional restriction required to ensure that matrix products of fast arrays are always fast arrays: Both the number of rows and the number of columns are no larger than $\lfloor\sqrt{\text{maximum-positive-32-bit-integer}}\rfloor$.

2. The test for matrix equality, `(m-= M1 M2)`, is defined so that it is an equivalence relation on the entire ACL2 universe. When at least one of `M1` or `M2` is not a logical array, then `m-=` coincides, by definition, with `equal`:

```
(defun
  m-= (M1 M2)
  "Determine if the matrices represented by the alists
   M1 and M2 are equal (as matrices of numbers)."
  (declare (xargs :guard (and (array2p '$arg1 M1)
                              (array2p '$arg2 M2))))
  (if (mbt (and (alist2p '$arg1 M1)
                (alist2p '$arg2 M2)))
      (let ((dim1 (dimensions '$arg1 M1))
            (dim2 (dimensions '$arg2 M2)))
           (if (and (= (first dim1)
                       (first dim2))
                    (= (second dim1)
                       (second dim2)))
               (m-=-row-1 (compress2 '$arg1 M1)
                          (compress2 '$arg2 M2)
                          (- (first dim1) 1)
                          (- (second dim1) 1))
               nil))
      (equal M1 M2))).
```

Note the use of the new macro `mbt` in this definition. Here is an explanation of `mbt` as well as the functions used in this definition.

- `(alist2p name A)` returns `t` if `A` is a two dimensional logical array. Otherwise it returns `nil`. The extra input argument `name` is included only to be consistent with built-in ACL2 array manipulation functions such as `aref2`, `dimensions`, and `compress2` that include such an argument to make ACL2's "fast" implementation of arrays possible.
- `(array2p name A)` returns `t` if `A` is a two dimensional fast executable ACL2 array. Otherwise it returns `nil`. The extra input argument `name` is used by ACL2's "fast" implementation of arrays.

- `(aref2 name A i j)` where `A` is a two dimensional array and i and j are legal indices into `A`. This function returns the value associated with `(i . j)` in `A`, or else the default value of the alist.

  This function executes in nearly constant time if `A` has been stored, behind the scenes, in a Common Lisp array. When it has not, `aref2` must do a linear search through the alist `A`.

- `(compress2 name A)` where `A` is a two dimensional array. Logically, this function removes irrelevant pairs from the alist `A`. The function returns a new alist, `A'`, equivalent to `A` under `aref2`. `A'` may be shorter than `A` and the non-irrelevant pairs may occur in a different order in `A'` than in `A`.

  This function plays an important role in the efficient implementation of `aref2`. In addition to creating the new alist `A'`, `compress2` allocates and stores (a representation of) the new alist in a Common Lisp array.

- `(dimensions name A)` where `A` is a two dimensional array. This function returns the dimensions list of the array alist. That list contains at least two elements, the first is the positive integer number of rows and the second is the positive integer number of columns.

- `mbt` ("must be true") is a proposed new (starting with Version 2.8) ACL2 macro that can be used to replace an expensive Boolean test, in a :logic mode function definition, with `t` during execution, provided that the Boolean test must be true whenever the function's guard is true.

  Semantically, `(mbt x)` equals `x`. However, in raw Lisp `(mbt x)` ignores `x` entirely, and macro-expands to `t`. ACL2's guard verification mechanism ensures that the raw Lisp code is only evaluated when appropriate, since a guard proof obligation `(equal x t)` is generated.

  In the above definition of `m-=`, since the predicate `alist2p` is implied by `array2p`, the `mbt` can be used to replace the `alist2p` tests with `t` during execution.

- `(m-=-row-1 A1 A2 m n)` returns `t` if for all `i, j` such that $0 \leq$ `i` $\leq$ `m` and $0 \leq$ `j` $\leq$ `n`,

  `(fix (aref2 name A1 i j))` = `(fix (aref2 name A2 i j))`.

  Otherwise it returns `nil`.

3. The test for matrix equality, `(m-= M1 M2)`, is a congruence relation with respect to the matrix operations of transpose, unary minus, scalar multiplication, addition, and multiplication.

4. The matrix operations of addition, `(m-+ M1 M2)`, and multiplication, `(m-* M1 M2)`, are defined to be operations on the entire ACL2 universe. When at least one of `M1` or `M2` is not a logical array, then `m-+` coincides, by definition, with `+` and `m-*` coincides, by definition, with `*`.

This allows some matrix equalities to be stated without any hypotheses restricting the operands to be arrays. For example, `m-+` and `m-*` satisfy these hypotheses-free versions of distributivity

```
(m-= (m-* M1 (m-+ M2 M3))
     (m-+ (m-* M1 M2)
          (m-* M1 M3)))
```

```
(m-= (m-* (m-+ M1 M2) M3)
     (m-+ (m-* M1 M3)
          (m-* M2 M3))).
```

This also allows some matrix equalities to be stated in terms of `equal` in place of `m-=`. For example, `m-+` satisfies this version of commutativity

```
(equal (m-+ M1 M2)
       (m-+ M2 M1))
```

as well as this weaker version

```
(m-= (m-+ M1 M2)
     (m-+ M2 M1)).
```

Similar comments apply to the associativity of `m-+` and `m-*`.

5. Determinants and multiplicative matrix inverses are computed using row and column operations. The determinant is used to determine if a square matrix is singular or nonsingular. ACL2 proofs are still required for the following

   - For square matrices, whenever the determinant is not 0, then `m-/` computes the two-sided multiplicative inverse.
   - Whenever the determinant is 0 then there is no inverse.
   - Non-square matrices do not have two-sided inverses.

   Meanwhile, we use this temporary definition of a nonsingular matrix: A matrix is nonsingular if and only if it is a square matrix and `m-/` does, in fact, compute a two-sided multiplicative inverse.

6. Closure properties of matrix operations:

   - ACL2 logical arrays, fast arrays, and matrices are all closed under the operations of transpose, unary-minus, and scalar multiplication.

- ACL2 logical arrays, fast arrays, and matrices are all closed under the operations of matrix sum and matrix multiplicative inverse, whenever the results of these operations exist mathematically.
- ACL2 logical arrays and matrices, but not ACL2 fast arrays, are closed under matrix multiplication, whenever this operation produces a mathematical result.

Summary: Matrix Operations Implemented in ACL2.

This summary is based on the ACL2 Matrix Algebra Book found in the file `matalg.lisp`.

Recognizers.

Boolean test for a "slow" logical array. `(alist2p name A)`

- Determine if `A` is an alist, meeting a minimal number of conditions, required by the applicative semantics, to be a two dimensional array.
- Enforce no restrictions on the upper-bounds of the number of rows and the number of columns.

Boolean test for a "fast" executable array. `(array2p name A)`

- Determine if `A` is an ACL2 two dimensional array.
- Enforce the restriction (required to ensure fast accessing and updating) that the product, of the number of rows and the number of columns, is less than the maximum-positive-32-bit-integer.

Boolean test for a matrix. `(matrixp m n A)`

- Determine if `A` is a $\mathtt{m} \times \mathtt{n}$ matrix.
- Enforce an restriction ensuring that matrix products of fast arrays are always fast arrays:

$$\begin{aligned} \mathtt{m}, \mathtt{n} &\leq \lfloor \sqrt{\text{maximum-positive-32-bit-integer}} \rfloor \\ &= 46,340. \end{aligned}$$

Number of rows. `(r M)`

- Return the number of rows in the matrix `M`.

Number of columns. `(c M)`

- Return the number of columns in the matrix `M`.

Matrix equality. `(m-= M1 M2)`

- Determine if the matrices represented by the alists `M1` and `M2` are equal (as matrices of numbers).
- `m-=` is an equivalence relation.

Zero matrix. `(m-0 m n)`

- Return an alist representing the `m` × `n` matrix whose elements are all equal to 0.
- Closure of `m-0`

```
(implies (and (integerp m)(> m 0)
              (integerp n)(> n 0))
         (alist2p name (m-0 m n)))


(implies (and (symbolp name)
              (integerp m)(> m 0)
              (integerp n)(> n 0)
              (< (* m n)
                 *MAXIMUM-POSITIVE-32-BIT-INTEGER*))
         (array2p name (m-0 m n)))


(implies
  (and (integerp m)(> m 0)
       (integerp n)(> n 0)
       (<=
         m
         *INT-SQRT-MAXIMUM-POSITIVE-32-BIT-INTEGER*)
       (<=
         n
         *INT-SQRT-MAXIMUM-POSITIVE-32-BIT-INTEGER*))
  (matrixp m n (m-0 m n)))
```

Identity matrix. `(m-1 n)`

- Return an alist representing the `n` × `n` identity matrix.
- Closure of `m-1`

```
(implies (and (integerp n)(> n 0))
         (alist2p name (m-1 n)))


(implies (and (symbolp name)
              (integerp n)(> n 0)
              (< (* n n)
                 *MAXIMUM-POSITIVE-32-BIT-INTEGER*))
         (array2p name (m-1 n)))
```

```
(implies
  (and (integerp n)(> n 0)
       (<=
         n
         *INT-SQRT-MAXIMUM-POSITIVE-32-BIT-INTEGER*))
  (matrixp n n (m-1 n)))
```

Transpose of a matrix. `(m-trans M)`

- Return an alist representing the transpose of the matrix represented by the alist `M`.
- idempotency of `m-trans`

```
(implies (alist2p name M)
         (m-= (m-trans (m-trans M))
              M)))
```

- `m-=` is a congruence for `m-trans`

```
(implies (m-= M1 M2)
         (m-= (m-trans M1)
              (m-trans M2)))
```

- `m-trans` of `m-0`

```
(implies (and (integerp m)(> m 0)
              (integerp n)(> n 0))
         (m-= (m-trans (m-0 m n))
              (m-0 n m)))
```

- `m-trans` of `m-1`

```
(implies (and (integerp n)(> n 0))
         (m-= (m-trans (m-1 n))
              (m-1 n)))
```

- Closure of `m-trans`

```
(implies (alist2p name M)
         (alist2p name (m-trans M)))


(implies (array2p name M)
         (array2p name (m-trans M)))
```

```
(implies (matrixp m n X)
         (matrixp n m (m-trans X)))
```

Unary minus of a matrix. `(m-- M)`

- Return an alist representing the unary minus of the matrix represented by the alist `M`.
- idempotency of `m--`

  ```
  (implies (alist2p name M)
           (m-= (m-- (m-- M))
               M))
  ```

- `m-=` is a congruence for `m--`

  ```
  (implies (m-= M1 M2)
           (m-= (m-- M1)
                (m-- M2)))
  ```

- `m--` of `m-0`

  ```
  (implies (and (integerp m)(> m 0)
                (integerp n)(> n 0))
           (m-= (m-- (m-0 m n))
                (m-0 m n)))
  ```

- `m-trans` of `m--`

  ```
  (implies (alist2p name M)
           (m-= (m-trans (m-- M))
                (m-- (m-trans M))))
  ```

- Closure of `m--`

  ```
  (implies (alist2p name M)
           (alist2p name (m-- M)))
  ```

  ```
  (implies (array2p name M)
           (array2p name (M-- M)))
  ```

  ```
  (implies (matrixp m n X)
           (matrixp m n (m-- X)))
  ```

Scalar multiplication of a matrix. `(s-* a M)`

- Return an alist representing the multiplication of the scalar a times the matrix represented by the alist `M`.
- `m-=` is a congruence for `s-*`

```
(implies (m-= M1 M2)
         (m-= (s-* a M1)
              (s-* a M2)))
```

- associate scalars left for `s-*`

```
(implies (alist2p name M)
         (m-= (s-* a1 (s-* a2 M))
              (s-* (* a1 a2) M)))
```

- multiply by scalar 0

```
(implies (alist2p name M)
         (m-= (s-* 0 M)
              (m-0 (r M)(c M))))
```

- multiply by scalar 1

```
(implies (alist2p name M)
         (m-= (s-* 1 M)
              M))
```

- multiply by scalar -1

```
(implies (alist2p name M)
         (m-= (s-* -1 M)
              (m-- M)))
```

- multiply `m-0` by scalar

```
(implies (and (integerp m)(> m 0)
              (integerp n)(> n 0))
         (m-= (s-* a (m-0 m n))(m-0 m n)))
```

- `m-trans` of `s-*`

```
(implies (alist2p name M)
         (m-= (m-trans (s-* s M))
              (s-* s (m-trans M))))
```

- `s-*` of `m--`

  ```
  (implies (alist2p name M)
           (m-= (s-* a (m-- M))
                (m-- (s-* a M))))
  ```

- Closure of `s-*`

  ```
  (implies (alist2p name M)
           (alist2p name (s-* a M)))


  (implies (array2p name M)
           (array2p name (s-* a M)))


  (implies (matrixp m n X)
           (matrixp m n (s-* a X)))
  ```

Matrix sum. `(m-+ M1 M2)`

- Return an alist representing the matrix sum of the matrices represented by the alists `M1` and `M2`.
- `m-=` is a congruence for `m-+`

  ```
  (implies (m-= M1 M2)
           (equal (m-+ M1 M3)
                  (m-+ M2 M3)))


  (implies (m-= M2 M3)
           (equal (m-+ M1 M2)
                  (m-+ M1 M3)))
  ```

- commutativity of `m-+`

  ```
  (equal (m-+ M1 M2)
         (m-+ M2 M1))


  (equal (m-+ X Y Z)
         (m-+ Y X Z))
  ```

- associativity of `m-+`

  ```
  (equal (m-+ (m-+ M1 M2) M3)
         (m-+ M1 M2 M3))
  ```

- m-+ unicity of m-0

```
(implies (alist2p name M)
         (m-= (m-+ M (m-0 (r M)(c M)))
              M))


(implies (alist2p name M)
         (m-= (m-+ (m-0 (r M)(c M)) M)
              M))
```

- m-+ inverse of m--

```
(implies (alist2p name M)
         (m-= (m-+ (m-- M) M)
              (m-0 (r M)(c M))))


(implies (alist2p name M)
         (m-= (m-+ M (m-- M))
              (m-0 (r M)(c M))))


(implies (and (alist2p name X)
              (alist2p name Y)
              (equal (r X)(r Y))
              (equal (c X)(c Y)))
         (m-= (m-+ X (m-- X) Y)
              Y))


(implies (and (alist2p name X)
              (alist2p name Y)
              (equal (r X)(r Y))
              (equal (c X)(c Y)))
         (m-= (m-+ (m-- X) X Y)
              Y))
```

- distribute s-* over +

```
(implies (alist2p name M)
         (m-= (s-* (+ a b) M)
              (m-+ (s-* a M)(s-* b m))))
```

- distribute `s-*` over `m-+`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (r M1)(r M2))
              (equal (c M1)(c M2)))
         (m-= (s-* a (m-+ M1 M2))
              (m-+ (s-* a M1)(s-* a M2))))
```

- `m-trans` of `m-+`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (r M1)(r M2))
              (equal (c M1)(c M2)))
         (m-= (m-trans (m-+ M1 M2))
              (m-+ (m-trans M1)(m-trans M2))))
```

- distribute `m--` over `m-+`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (r M1)(r M2))
              (equal (c M1)(c M2)))
         (m-= (m-- (m-+ M1 M2))
              (m-+ (m-- M1)(m-- M2))))
```

- Closure of `m-+`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (r M1)(r M2))
              (equal (c M1)(c M2)))
         (alist2p name (m-+ M1 M2)))
```

```
(implies (and (array2p name M1)
              (array2p name M2)
              (equal (r M1)(r M2))
              (equal (c M1)(c M2)))
         (array2p name (m-+ M1 M2)))
```

```
(implies (and (matrixp m n X1)
              (matrixp m n X2))
         (matrixp m n (m-+ X1 X2)))
```

Matrix product. `(m-* M1 M2)`

- Return an alist representing the matrix product of the matrices represented by the alists `M1` and `M2`.
- `m-=` is a congruence for `M-*`

```
(implies (m-= M1 M2)
         (equal (m-* M1 M3)
                (m-* M2 M3)))
```

```
(implies (m-= M2 M3)
         (equal (m-* M1 M2)
                (m-* M1 M3)))
```

- nullity of `m-0` for `m-*`

```
(implies (and (alist2p name M1)
              (integerp m)
              (> m 0))
         (m-= (m-* (m-0 m (r M1))
                   M1)
              (m-0 m (c M1))))
```

```
(implies (and (alist2p name M1)
              (integerp p)
              (> p 0))
         (m-= (m-* M1
                   (m-0 (c M1) p))
              (m-0 (r M1) p)))
```

- unity of `m-1` for `m-*`

```
(implies (alist2p name M1)
         (m-= (m-* (m-1 (r M1))
                   M1)
              M1))
```

```
(implies (alist2p name M1)
         (m-= (m-* M1
                   (m-1 (c M1)))
              M1))
```

- associativity of `m-*`

```
(equal (m-* (m-* M1 M2) M3)
       (m-* M1 M2 M3))
```

- distribute `m-*` over `m-+`

```
(m-= (m-* M1 (m-+ M2 M3))
     (m-+ (m-* M1 M2)
          (m-* M1 M3)))


(m-= (m-* (m-+ M1 M2) M3)
     (m-+ (m-* M1 M3)
          (m-* M2 M3)))
```

- `s-*` and `m-*`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (m-= (m-* (s-* a M1) M2)
              (s-* a (m-* M1 M2))))


(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (m-= (m-* M1 (s-* a M2))
              (s-* a (m-* M1 M2))))
```

- `m-trans` of `m-*`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (m-= (m-trans (m-* M1 M2))
              (m-* (m-trans M2)(m-trans M1))))
```

- `m-*` and `m--`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (m-= (m-* (m-- M1) M2)
              (m-- (m-* M1 M2))))
```

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (m-= (m-* M1 (m-- M2))
              (m-- (m-* M1 M2))))
```

- Closure of `m-*`

```
(implies (and (alist2p name M1)
              (alist2p name M2)
              (equal (c M1)(r M2)))
         (alist2p name (m-* M1 M2)))
```

```
(implies (and (array2p name M1)
              (array2p name M2)
              (equal (c M1)(r M2))
              (< (* (r M1)(c M2))
                 *MAXIMUM-POSITIVE-32-BIT-INTEGER*))
         (array2p name (m-* M1 M2)))
```

```
(implies (and (matrixp m n X1)
              (matrixp n p X2))
         (matrixp m p (m-* X1 X2)))
```

Matrix inverse and determinant. `(m-/ M)` and `(determinant M)`

- Return an alist representing the matrix inverse of the matrix represented by the alist `M`. Also compute the determinant of `M`.
- Row and column operations are used to compute the inverse and determinant.
- The determinant is used to determine if a square matrix is singular or nonsingular.
- ACL2 Proofs still required:
  - `m-=` is a congruence for `m-/` and `determinant`.
  - For square matrices, whenever the determinant is not 0, then `m-/` computes the two-sided multiplicative inverse.
  - Whenever the determinant is 0 then there is no inverse.
  - Non-square matrices do not have two-sided inverses.

- Meanwhile we use this temporary definition of singular

```
(defun
  m-singularp (M)
  (declare (xargs :guard (array2p '$c M)
                  :verify-guards nil))
  (not (and (mbt (alist2p '$c M))
            (= (r M)(c M))
            (m-= (m-* M (m-/ M))
                 (m-1 (r M)))
            (m-= (m-* (m-/ M) M)
                 (m-1 (r M))))))
```

- `m-/` of `m-*`

```
(implies (and (not (m-singularp M1))
              (not (m-singularp M2))
              (not (m-singularp (m-* M1 M2)))
              (equal (c M1)(r M2)))
         (m-= (m-/ (m-* M1 M2))
              (m-* (m-/ M2)(m-/ M1))))
```

- Closure of `m-/`

```
(implies (and (alist2p name M)
              (equal (r M)(c M)))
         (alist2p name (m-/ M)))


(implies (and (array2p name M)
              (equal (r M)(c M)))
         (array2p name (m-/ M)))


(implies (and (matrixp (r M)(c M) M)
              (equal (r M)(c M)))
         (matrixp (r M)(c M)(m-/ M)))
```