Point-free, set-free concrete linear algebra

Georges Gonthier*

Microsoft Research Cambridge
gonthier@microsoft.com

Abstract. Abstract linear algebra lets us reason and compute with collections rather than individual vectors, for example by considering entire subspaces. Its classical presentation involves a menagerie of different set-theoretic objects (spaces, families, mappings), whose use often involves tedious and non-constructive pointwise reasoning; this is in stark contrast with the regularity and effectiveness of the matrix computations hiding beneath abstract linear algebra. In this paper we show how a simple variant of Gaussian elimination can be used to model abstract linear algebra directly, using matrices only to represent all categories of objects, with operations such as subspace intersection and sum. We can even provide effective support for direct sums and subalgebras. We have formalized this work in Coq, and used it to develop all of the group representation theory required for the proof of the Odd Order Theorem, including results such as the Jacobson Density Theorem, Clifford's Theorem, the Jordan-Holder Theorem for modules, the Wedderburn Structure Theorem for semisimple rings (the basis for character theory).

Key words: Formalization of Mathematics, Linear Algebra, Module Theory, Algebra, Type inference, Coq, SSReflect

1 Introduction

General linear algebra[1] is amongst the most ubiquitous and useful basic non-trivial mathematical theory, probably because it mediates calculations and combinatorial deductive reasoning, linking computations in cartesian coordinates to abstract geometric arguments, or purely combinatorial properties of finite groups with algebraic properties of their linear representations. Developing a good linear algebra library was one of the important side goals of our Feit-Thompson Theorem proof project[2–7].

Naturally, most computer proof systems supply one (or more!) linear algebra libraries[8–13]. However most are limited to the algebra of vectors and/or matrices and do not support point-free reasoning using whole subspaces. The rare exceptions[10, 12, 14] use classical sets to represent subspaces. This basically combinatorial account fails to capture some specifics of linear subsets, in particular their algebraic properties under sum, intersection and linear image.

Note however that all objects used in linear algebra can be represented as matrices: endomorphisms by their matrix, (row) vectors by $1 \times n$ matrices, lists of

* This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

vectors and bases by rectangular matrices, and subspaces by a basis. Under this identification the same matrix multiplication operation AF can mean composing A and F, applying F to A, mapping F over A or taking the image of A under F. The (unique) matrix product associativity and distributivity laws are consistent with all those interpretations — a major simplification of the theory.

We came to this observation by accident. Because we wanted a constructive formalization of linear algebra, we had to define an effective membership test for linear sets. After working out a suitable generalization of Gaussian elimination we realized it actually provided all the set theoretic subspace constructions, so we could do away with the entire set-theoretic boilerplate and use matrices only.

We then applied the resulting library to one of the then outstanding prerequisites of the Feit-Thompson Theorem — an extensive development of group module and representation theory. This worked out remarkably well, and was also invaluable in shaping the details and ironing out all the kinks of the core linear algebra formalization, for instance prompting the development of indexed subspace sums and directed sums.

It is our experience that such large scale use is essential for obtaining a usable formalization. With an appropriate framework, all basic linear algebra proofs are trivial (2-5 lines) and hence provide no useful feedback on the library design choices. Linear subspace theory is in the 10-line range and similarly offers little guidance. It is only with representation theory, with proofs in the 30-50 line range, that we started to identify substantial issues, and the hardest issues, such as the need to support complex direct sums and non-constructive results, only appeared in the Feit-Thompson Theorem proof itself, with proofs in the 200+ line range.

The contributions of this paper are thus: a practical matrix encoding of linear subspaces and their operations (section 3), an innovative use of type inference and dependent types to formalize general direct sums of subspaces (section 4), and a large-scale validation of the resulting library with an extensive library on finite group representations and its application to the Local Analysis part of the Feit-Thompson Theorem proof[3] (section 5).

This work was done using the SSReflect extension of the Coq system[15, 16]. We review the basic SSReflect matrix algebra library [16, 6] in section 2, and use mathematical notation as much as possible in section 3, but due to lack of space we assume some familiarity with our prior work[7, 6] in the more technical sections 3.4 and 4.

The libraries described here can be viewed at `http://coqfinitgroup.gforge.inria.fr/`; they will be distributed as part of the next SSReflect release, early during the review period.

2 Matrix operations

2.1 A combinatorial and algebraic hierarchy

Matrices are a typical *container* type. The properties of a given matrix type will typically be a function of the properties of the type of its coefficients: while all matrices will share some structural properties such as shape, only matrices

with comparable elements can be compared, only matrices over a ring can be multiplied, etc.

In the SSReflect library this notion of "type with properties" is captured with `Structure`s, which are just (higher-kinded) record types with two fields, a *sort*, or carrier type, and a *class*, itself a record providing various operations over the sort along with some of their properties. For example, a "comparable" type, or `eqType`, could be described as follows:

```
Module Equality.
Record class_of (T : Type) : Type :=
  Mixin {op : T -> T -> bool; _ : forall x y, x = y <-> op x y}.
Structure type : Type := Pack {sort; class : class_of sort}.
End Equality.
Notation eqType := Equality.type.
Coercion Equality.sort : eqType >-> Sortclass.
Definition eq_op T := Equality.op (Equality.class T).
Notation "x == y" := (@eq_op _ x y).
```

The `Coercion` line lets us use an `eqType` as a type, as in

```
Let swap {T : eqType} (x y z : T) := if z == x then y else z.
```

Note that this is very similar to the Haskell type class mechanism, except for the extra layer of packaging introduced by the `type Structure`, which is made possible by Coq's higher-kinded types. This extra packaging has important consequences on the feasibility of type checking, especially in the presence of container types such as matrices[7].

Similarly to the type class `Instance` declaration, the `Canonical Structure` declaration lets us tie specific structures to existing types, e.g., allowing us to equip `bool` and `nat` with definitions for `_ == _`.

The SSReflect library defines many such structures (97 at last count), which provide many standard sets of operations, from basic combinatorial fare such as `eqType` above to standard algebraic objects such as rings and fields, and combinations thereof such as finite fields. Here are a few of the less common ones

- `finType` a finite, explicitly enumerable type; any subset A of a `finType` can be enumerated (`enum` A) and counted (`#|`A`|`).
- `choiceType` a type with a choice operator `choose` P x_0 that picks a *canonical* x such that Px holds, given x_0 such that Px_0.
- `zmodType` a type with an addition operation, and therefore integer scaling.
- `lmodType R` a type with both an addition operation and a scaling operation (denoted α`*:`v in Coq) with coefficients α in `R` (which must be a `ringType`).
- `unitRingType` a ring with an effective test for unit (invertible) elements, and a partial inverse function for its units.

These Structures are arranged in a (multiple) inheritance hierarchy in the obvious way[7]. It is important to note that `zmodType`, the smallest algebraic structure, inherits from both `eqType` and `choiceType`.

Let us finally point out that unlike Haskell type classes (but similarly to their Coq reinterpretation[17], `Structure` keys are not limited to types. The "big operator" library[6] uses these to recognize AC operators, and we will be using

below similar structures to quantify over linear functions (between `lmodType`s) and ring morphisms.

2.2 Basic algebra

Matrices are basically tabulations of functions with a finite rectangular domain of the form $[0, m) \times [0, n)$. The SSReflect library defines both finite index types (`ordinal n`, denoted `'I_n`), and a generic tabulation type constructor `{ffun ..}` for functions with a `finType` domain, which we simply combine:

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

Note that `matrix R m n` is a *dependent* type that is specialized to the $m \times n$ shape. This is required to develop an algebraic theory, because too many laws do not generalize to "matrices of unknown shape". The usual Coq notation for this type is `'M_(m, n)` as `R` can usually be inferred from context.

The `finfun` library provides us with a one-to-one correspondence between `{ffun A -> R}` and `A -> R`, which we only need to curry to get a

```
Coercion fun_of_matrix : matrix >-> Funclass.
```

that lets us write `A i j` in Coq for the A_{ij} coefficient of A. We provide a dual notation for defining matrices, which we use for all matrix arithmetic operators:

```
Definition addmx A B := \matrix_(i, j) (A i j + B i j).
Definition mulmx A B := \matrix_(i, k) \sum_j (A i j * B j k).
```

(We have omitted some type declarations.) While they may not use the most efficient algorithms, these definitions have the advantage of actually being *useful* for proving algebraic identities. Indeed most algebraic identities can be proved in one or two lines.

We then declare `Canonical z/lmodType Structure`s so that addition and scaling can be denoted with the generic `+` and `*:` operators, and all lemmas of the generic algebra package become available. However, we still require a separate operator (denoted `*m`) for multiplication, because only nontrivial square matrix types are proper `ringType`s.

This is about the point where most matrix libraries end, but we can easily carry on and define the `unitRingType` structure, along with determinants, cofactors, and adjugate matrices, by leveraging the SSReflect permutation library[5]:

```
Definition determinant n (A : 'M_n) : R :=
  \sum_(s : 'S_n) (-1) ^+ s * \prod_i A i (s i).
Definition cofactor n A (i j : 'I_n) : R :=
  (-1) ^+ (i + j) * determinant (row' i (col' j A)).
Definition adjugate n (A : 'M_n) := \matrix_(i, j) cofactor A j i.
```

Even these proofs remain relatively easy: it takes about 20 lines to show the Cauchy determinant product formula $|AB| = |A|.|B|$ and the Laplace expansion formula for cofactors, and then 9 lines to derive the Cramer rule $A.(\mathbf{adj}\ A) = |A|.1$, from which we can prove the Cayley-Hamilton theorem in three lines[6].

2.3 Block and reshaping operations

Matrices are also combinatorial objects, and the SSReflect `matrix` library supplies some 26 operations for rearranging the contents of matrices over *any* type. This includes transposition and extraction, permutation and suppression of row and columns (the `row'` and `col'` functions above perform the latter). Most importantly, this also includes operations for cutting and pasting *block matrices*, e.g.,

```
Definition row_mx (A : 'M_(m, n)) (B : 'M_(m, p)) : 'M_(m, n + p)
  := \matrix_(i, j)
       match split j with inl k => A i k | inr l => B i l end.
```

computes the block row matrix $(A\ B)$, using the function

```
split : 'I_(n + p) -> 'I_n + 'I_p
```

from the `fintype` library to map column indices to the appropriate submatrix. A set of lemmas extends the usual matrix computation rules to 1×2, 2×1 and 2×2 block matrices, which let us prove many identities without having to consider individual coefficients.

As with the call to `split` above, it is usually not necessary how a block matrix is subdivided — the syntactic shape of the dimensions supplies that information via type inference. There is a downside: we may end up with matrices that have extensionally, but not syntactically the correct shape, for instance when stating block matrix associativity. We use a *cast* operation to mitigate this:

```
    castmx : (m = m') * (n = n') -> 'M_(m, n) -> 'M_(m', n').
Lemma row_mxA : forall m n1 n2 n3 A B C,
  let cast := (erefl m, esym (addnA n1 n2 n3)) in
  row_mx A (row_mx B C) = castmx cast (row_mx (row_mx A B) C).
```

Note that `castmx` is bidimensional; its first argument is proof-irrelevant (because `nat` is an `eqType`) so we can prove rewrite rules that make it easy to move, collect and eliminate casts. We also provide a prototype-based cast: `conform_mx` $A\ B$ returns a matrix that has syntactically the same shape as A, but is equal to B if B has extensionally the same shape as A (and A otherwise).

Finally we define reshaping operations `mxvec` and `vec_mx` that turn a rectangular $m \times n$ matrix into linear $1 \times mn$ row vector and conversely.

3 Gaussian Elimination and Row Spaces

Here we show how to develop an algorithmic theory of linear algebra on the basis of a single Gaussian elimination procedure. We shall assume that all our matrices are over a fixed field.

3.1 Extended Gaussian elimination

All that is needed to extend Gaussian elimination gracefully to singular matrices is to perform *double pivoting*, i.e., to search for a non-zero pivot in all the matrix

and then swap both rows and columns to bring it to the top left corner. This gives an exact value for the rank of the matrix, as the decomposition stops exactly when it reaches a null matrix. This is our Coq code for this algorithm, which can also be viewed as a degenerate, easy case of the Smith normal form computation[1].

```
Fixpoint gaussian_elimination {m n} :=
  match m, n return 'M_(m, n) -> 'M_m * 'M_n * nat with
  | _.+1, _.+1 => fun A : 'M_(1 + _, 1 + _) =>
    if [pick ij | A ij.1 ij.2 != 0] is Some (i, j) then
      let a := A i j in let A1 := xrow i 0 (xcol j 0 A) in
      let u := ursubmx A1 in let v := a^-1 *: dlsubmx A1 in
      let: (L, U, r) := gaussian_elimination (drsubmx A1 - v *m u)
      in (xrow i 0 (block_mx 1 0 v L),
          xcol j 0 (block_mx a%:M u 0 U),
          r.+1)
    else (1%:M, 1%:M, 0%N)
  | _, _ => fun _ => (1%:M, 1%:M, 0%N)
  end.
```

This is virtually identical to the LUP decomposition procedure described in [7], and its correctness is easily established in a similar manner. Besides the double pivoting, the only differences are that row and column permutations are combined with the lower and upper triangular factors of the decomposition, and that the decomposition of a null matrix is a pair of identity matrices (`1%:M` is our Coq notation for a scalar matrix with 1s on the diagonal). If we denote by 1_r a (not necessarily square) matrix that has 1s in r first coefficients on the main diagonal and 0 elsewhere

$$1_r = \begin{pmatrix} 1 & & & \\ & \ddots & & 0 \\ & & 1 & \\ & 0 & & 0 \end{pmatrix}$$

and set `gaussian_elimination` $A = (A_{\hat{C}}, A_{\hat{R}}, r(A))$, then the correctness of the above function is expressed by the five conditions

$$r(A) \leq m, n \qquad A_{\hat{C}}, A_{\hat{R}} \text{ invertible} \qquad A_{\hat{C}}\, 1_{r(A)}\, A_{\hat{R}} = A$$

We call $A_{\hat{C}}$ and $A_{\hat{R}}$ the extended column and row bases of A, respectively. The column (resp. row) basis A_C (resp. A_R) of A consists or the $r(A)$ first columns (resp. rows) of $A_{\hat{C}}$ (resp. $A_{\hat{R}}$). Since $1_{r(A)} 1_{r(A)} = 1_{r(A)}$ we have

$$A_C = A_{\hat{C}}\, 1_{r(A)} \quad A_R = 1_{r(A)}\, A_{\hat{R}} \quad A_C\, A_R = A$$

3.2 Rank theory

The fact that $r(A)$ has indeed the properties of the matrix rank follows directly from the correctness conditions above and from the following two basic facts about matrices over a commutative ring:

Lemma 1. *If A and B are respectively $m \times n$ and $n \times m$ matrices such that $AB = 1$, then $m \leq n$, and if $m = n$ then $BA = 1$.*

We prove the second assertion first. Consider $A' = |B|.(\mathbf{adj}\ A)$; then

$$A'A = |B|.(\mathbf{adj}\ A)A = |B|.|A|.1 = (|A||B|).1 = |AB|.1 = 1$$

so $BA = A'ABA = A'A = 1$. For the first assertion, assume $n < m$, and let $A = (A_l\ A_r)$ where A_l is a square $n \times n$ matrix, and similarly $B = \binom{B_u}{B_d}$ with B_u square. Block product now gives

$$AB = (A_l\ A_r)\begin{pmatrix} B_u \\ B_d \end{pmatrix} = \begin{pmatrix} A_l B_u\ A_r B_u \\ A_l B_d\ A_r B_d \end{pmatrix} = \begin{pmatrix} 1\ 0 \\ 0\ 1 \end{pmatrix}$$

so $A_l B_u = 1$, whence $B_u A_l = 1$ by the second part, and so $1 = A_r B_d = A_r B_u A_l B_d = 0$, a contradiction since this is a nontrivial $(m-n) \times (m-n)$ matrix.

This yields both upper and lower bounds on the rank function:

Lemma 2. *If M and N are respectively $m \times r$ and $r \times n$ matrices, then $r(MN) \leq r$, and any A such that $NAM = 1$ must have $r(A) \geq r$.*

For the first assertion, let $L = 1_{r(MN)}(MN)^{-1}_{\hat{C}}$ and $U = (MN)^{-1}_{\hat{R}} 1_{r(MN)}$ be respectively $r(MN) \times m$ and $n \times r(MN)$ matrices, and apply the first lemma to

$$(LM)(NU) = L(MN)_{\hat{C}} 1_{r(MN)} (MN)_{\hat{R}} U = 1_{r(MN)} 1 1_{r(MN)} 1 1_{r(MN)} = 1$$

For the second assertion, apply the lemma to $(NA_C)(A_R M) = NAM = 1$.

It then follows immediately that $r(AB) = r(A_C A_R B_C B_R) \leq r(A), r(B)$ and

$$r(A+B) = r(A_C A_R + B_C B_R) = r\left((A_C\ B_C)\begin{pmatrix} A_R \\ B_R \end{pmatrix}\right) \leq r(A) + r(B)$$

3.3 Set operations

The extended row and column bases provide everything we need to implement set-theoretic operations on matrices. We define

$$\begin{array}{rl} \mathbf{ker}\, A = (1 - 1_{r(A)})A^{-1}_{\hat{C}} & A^{\sim 1} = A^{-1}_{\hat{R}} 1_{r(A)} A^{-1}_{\hat{C}} \\ \mathbf{coker}\, A = A^{-1}_{\hat{R}}(1 - 1_{r(A)}) & A +_s B = \binom{A}{B} \\ A \leq B \Leftrightarrow A(\mathbf{coker}\, B) = 0 & A \cap_s B = [\mathbf{ker}\,(A +_s B)]_l A \\ A \equiv B \Leftrightarrow A \leq B \leq A & \overline{A} = (1 - 1_{r(A)})A_{\hat{R}} \end{array}$$

We can see that $\mathbf{ker}\, A$ is the kernel of A viewed as a linear function since $(\mathbf{ker}\, A)A = (1 - 1_{r(A)})1_{r(A)}A_{\hat{R}} = 0$, and likewise that $\mathbf{coker}\, A$ is the cokernel of A. It follows that $A \leq B$ tests whether the row space of A is included in that of B, i.e., whether A (considered as a subspace) is included in B, and that $A \equiv B$ tests whether A and B represent the same subspace. $B^{\sim 1}$ is a partial inverse to B, since, if $A \leq B$ we have

$$A - AB^{\sim 1}B = AB^{-1}_{\hat{R}} B_{\hat{R}} - AB^{-1}_{\hat{R}} 1_{r(B)} B_{\hat{R}} = A(\mathbf{coker}\, B)B_{\hat{R}} = 0$$

Thus $A \leq B$ if and only if $A = DB$ for some matrix D. Note finally that if v is a row vector, then $v \leq B$ tests whether v is in the row space of B.

Obviously the row space of $A +_s B$ is the sum of the row spaces of A and B. In the definition of $A \cap_s B$, $K = \mathbf{ker}\,(A +_s B)$ is a square block matrix which we divide vertically; K_l designates the left (rectangular) block. To see that $A \cap_s B$ is indeed the intersection of A and B observe that

$$0 = K \begin{pmatrix} A \\ B \end{pmatrix} = K_l A + K_r B$$

so indeed $K_l A = -K_r B \leq A, B$. Conversely if $C = A'A = B'B$, then $(A' - B')(A +_s B) = 0$ so $(A' - B') \leq K$ and $C = A'A = DK_l A$ for some D, hence $C \leq A \cap_s B$.

All of the usual results on linear spaces and bases easily follow from these definitions, as well as some more advanced ones like the Frobenius rank inequality

$$r(AB) + r(BC) \leq r(B) + r(ABC)$$

all with proofs under twelve lines (most are under two) and *no induction*. The reason for this is that all the induction we need is neatly encapsulated inside the Gaussian elimination procedure. Indeed it is instructive to consider why $\overline{A}$, defined above as $A_{\hat{R}}$ with the top $r(A)$ rows zeroed out, is indeed a complement to the row space of A. The nonzero rows of $\overline{A}$ are the rows of the identity matrix returned by the base case of `gaussian_elimination`, permuted by the pivot column transpositions during the unwinding of the recursion. Thus these are vectors of the standard basis that complete the row base of A: our seemingly trivial changes to the LUP decomposition algorithms are in fact a proof of the incomplete basis theorem.

3.4 Algebras and subrings

The next structure up from linear spaces are F-algebras, which add a multiplicative ring structure. A finite dimensional F-algebra can always be embedded in its algebra of endomorphisms, which is a matrix algebra, so we ought to be able to extend our program of "doing it all with matrices" to F-algebras as well. However, there is a catch. To enjoy the natural ring structure of matrices, algebra elements should be square matrices; but to be considered as points in our encoding of subspaces, they should be flat row vectors.

Our solution is to stick to square matrices, but to use the reshaping function `vecmx` of section 2.3 when we need to test for membership in a subalgebra:

$$(f \in R) \Leftrightarrow (\texttt{mxvec}\ f \in R)$$

Note that if f is an $n \times n$ matrix, then R will have to be an $m \times n^2$ matrix (whose type will be denoted `'A_(m,n)`). The pointwise product of two subalgebras can be defined using the iterated sums of normalized spaces we will define in section 4.2

```
Definition mulsmx m1 m2 n (R1 : 'A_(m1, n)) (R2 : 'A_(m2, n)) :=
  (\sum_i <<R1 *m lin_mx (mulmxr (vec_mx (row i R2)))>>)%MS.
```

We used the `lin_mx` function to tabulate a linear matrix-to-matrix function:

```
Definition lin_mx (f : 'M[R]_(m1, n1) -> 'M[R]_(m2, n2)) :=
  \matrix_(i, j) mxvec (f (vec_mx (delta_mx 0 i))) 0 j.
```

Further, by combining `lin_mx` with our set-like linear functions we can define ideals, subrings, centralisers and centers of algebras, as we can program effective tests for just about any linear condition. For example we can test whether a subalgebra R has an identity element (an $e \neq 0$ such that $ef = fe = f$ for all $f \in R$) with the following predicate

```
Definition has_mxring_id m n (R : 'A_(m , n)) :=
  (R != 0) &&
  (row_mx 0 (row_mx (mxvec R) (mxvec R))
    <= row_mx (cokermx R)
      (row_mx (lin_mx (mulmx R \o lin_mulmx))
              (lin_mx (mulmx R \o lin_mulmxr))))%MS.
```

The lower inclusion is satisfied iff there is an e such that the left-hand side is equal to the product of $v =$ `mxvec` e by the right-hand side, i.e., that $u($**coker** $R) = 0$ and $R = R($`lin_mx` (`mulmx` e)) $= R($`lin_mx` (`mulmxr` e)), that is, $e \in R$ and $ef = fe = f$ for all the $f =$ `vec_mx` (`row` i R), which generate R.

4 General direct sums

The concept of *direct sum* is one of the more powerful tool for reasoning about collections of subspaces, because it links a strong combinatorial property (unique decomposition) to a simple arithmetic (in)equality of ranks. This correspondence is especially useful when applied to general iterated sums, but there are some intricate technical issues that must be addressed to formalize it in Coq.

4.1 On subspace equality

While the theory exposed in Section 3 lets us compute and reason with subspaces represented as matrix row spaces, its does not provide a unique representation for subspaces. Indeed, for any given A, there are many $B \equiv A$, and this remains true even if we restrict ourselves to square matrices.

In addition, the "setoid" framework that implements relational congruence in Coq[18, 15] is incapable of dealing with the multiply dependent, polymorphic relation $A \equiv B$. We must resort to a proxy relation `A :=: B` that lets us replace A by B directly in expressions of the form $r(A)$, $A \leq C$ and $C \leq A$; these three cases cover most of the contexts in which we need to substitute equivalent subspace expressions.

For other contexts, we can either compose context lemmas directly or use the `choiceType` structure to obtain a *standard* representation:

$$\langle A \rangle = \texttt{choose}\ (\lambda B : M_n.\ A \equiv B)\ (1_{r(A)} A_{\hat{R}})$$

This defines $\langle A \rangle$ (Coq notation: `<<A>>`) as a square matrix with the same row space as A, such that $\langle A \rangle = \langle B \rangle$ iff $A \equiv B$.

4.2 Monoidal set operations

While the SSReflect `bigop` library[6] will let us turn the binary subspace operators $+_s$ and $\cap_s$ into n-ary ones, most it its facilities would be unusable because they require strictly monoidal operators (e.g., we need $A +_s 0 = A$, not $A +_s 0 \equiv A$). Fortunately, it turns out we can use $\langle A \rangle$ to fix this, by setting:

$$A +_{ss} B = \begin{cases} A & \text{if } B = 0 \text{ and } A \text{ is square} \\ B & \text{if } A = 0 \text{ and } B \text{ is square} \\ \langle A +_s B \rangle & \text{otherwise} \end{cases}$$

We use the `conform_mx` function of section 2.3 to code the first two cases in Coq. It is easy to show that $+_{ss}$ is strictly monoidal as its identity element 0 is only equivalent to one square matrix — itself. Thus, this definition lets us use generic `bigop` sums for subspace sums (Coq notation `(\sum_ ...)%MS`).

Obtaining a strictly monoidal intersection is similar but more delicate because although we can choose the identity matrix 1 as the identity element, it is by no means unique. We need to ensure that our normalization operation does not return 1 by accident; we thus write $A \simeq 1$ when $A \equiv 1$ and $A = 1$ if A is square, and let $\langle A \rangle_1$ be a canonical square matrix $B \equiv A$ such that $B = 1$ iff $A \simeq 1$. Then we can take $A \cap_{ss} B$ to be B if $A \simeq 1$ and B is square, A if $B \simeq 1$ and A is square, else $\langle A \rangle_1$ if $B \equiv 1$, and $\langle A \cap_s B \rangle_1$ otherwise.

4.3 A direct sum package

A binary sum $A +_s B$ is direct iff $A \cap_s B = 0$, or, equivalently iff $r(A +_s B) = r(A) + r(B)$. Both characterizations are useful, but the latter one generalizes best to arbitrary sums, by which we mean arbitrary combinations of binary and n-ary sums, as

$$\sum A \text{ direct iff } r\left(\sum A\right) = \sum r(A)$$

To formalize this definition it would appear we need do describe arbitrary general sum expressions $\sum A$, which would require some sort of reflexion or quotation. On closer examination, however, note that we do not actually care about the exact makeup of a sum: we only need its value (a subspace), and the sum of the ranks of the summands (an integer), so we can use the type

```
Structure proper_mxsum_expr n := ProperMxsumExpr {
  proper_mxsum_val : 'M_n; proper_mxsum_rank : nat;
  _ : mxsum_spec proper_mxsum_val proper_mxsum_rank
}.
```

where the inductive predicate `mxsum_spec` A s states that s is the sum of the ranks of a finite collection of matrices, whose row space sum is A. Thus, A is direct iff $r(A) = s$.

As hinted by the `Structure` keyword, we wish to declare `Canonical` instances of `proper_mxsum_expr` so that we can infer these structures from either of their two projections. This poses no problem for the proper binary and n-ary sums; however for trivial (unary) sums we would need to declare

```
Canonical Structure trivial_mxsum n A :=
  @ProperMxsum n A (\rank A) (TrivialMxsum A).
```

whose `proper_mxsum_val` projection is an arbitrary matrix A. This is interpreted by Coq as a *default* projection, which will be used eagerly for any matrix expression that is not immediately a binary or n-ary sum (the `Canonical Structure` selection process is *determinate* and driven by the head symbol of the projection value). This is undesirable because in actual use n-ary sums are often rather large expressions that need abbreviations, and we expect these to be transparent to the direct sum predicate.

Getting the right unification behavior requires a few helper structures:

```
Structure wrapped T := Wrap {unwrap : T}.
Canonical Structure wrap T x := @Wrap T x.
```

is a generic wrapper with a default instance. A unification problem `unwrap` $w \sim t$ will immediately be turned into $w \sim$ `wrap` t, unless t is of the form `unwrap` u.

We then define the `mxsum_expr` structure as a "wrapped" `proper_mxsum_expr`

```
Structure mxsum_expr m n := Mxsum {
  mxsum_val : wrapped 'M_(m, n); mxsum_rank : wrapped nat;
  _ : mxsum_spec (unwrap mxsum_val) (unwrap mxsum_rank)
}.
Canonical Structure sum_mxsum n (S : proper_mxsum_expr n) :=
  Mxsum (wrap (proper_mxsum_val S)) (wrap (proper_mxsum_rank S))
      ...
Canonical Structure trivial_mxsum m n A :=
  Mxsum (Wrap A) (Wrap (\rank A)) (TrivialMxsum A).
```

Since `wrap` is "self-inserting", matching `unwrap (mxsum_val ?)` to some arbitrary matrix expression E will first try to use `sum_mxsum`, matching T to `proper_mxsum_val ?`. This will succeed if E is a proper binary or n-ary sum; otherwise, Coq will expand `wrap` E into `Wrap` E and use `trivial_mxsum`. In effect we use the `wrapped` structure to explicitly introduce limited nondeterminism in the otherwise determinate `Canonical Structure` inference process.

With these structures we can now put

```
Definition mxdirect_def m n T
    of phantom 'M_(m, n) (unwrap (mxsum_val T)) :=
  \rank (unwrap (mxsum_val T)) == unwrap (mxsum_rank T).
Notation mxdirect A := (mxdirect_def (Phantom 'M_(_,_) A%MS)).
```

where `Inductive phantom T (x : T):= Phantom` is the generic tagged unit type. These definitions let us write `mxdirect` S for an arbitrary subspace sum S, and have Coq infer the corresponding `mxsum_expr` that actually defines the meaning of this expression. We also `mxsum_expr` structure to define generic lemmas about direct sum, such as

```
Lemma mxrank_sum_leqif : forall m n (S : mxsum_expr m n),
  \rank (unwrap S) <= unwrap (mxsum_rank S) ?= iff mxdirect (
      unwrap S).
```

which gives the conditionally strict rank inequality. The `leqif` predicate denoted m `<=` n `?= iff` C reads $m \leq n$, with $m = n$ iff C. The `ssrnat` library defines several combinators for `leqif`, and when applying such combinators to `mxrank_sum_leqif` the unknown S can be inferred from any one of the three arguments of `leqif`, thanks to the dual set of canonical projections of `mxsum_expr`.

5 Module and Representation Theory

Giving a full account of our development of representation theory, or of its use in the proof of the Feit-Thompson Theorem, is clearly beyond the scope of this paper. This section therefore only samples the two subjects, to illustrate how the design choices of our matrix linear algebra library fare in practice.

5.1 Group representation

Group representations are basically morphisms from a given finite group G to some general linear group, so we adopt the design pattern introduced in [5] and define representations as a `structure` that can be inferred for specific group-to-matrices functions.

```
Definition mx_repr (G : {set gT}) n (r : gT -> 'M[R]_n) :=
    r 1%g = 1%:M
  /\ {in G &, {morph r : x y / (x * y)%g >-> x *m y}}.
Structure mx_representation G n :=
  MxRepresentation {repr_mx :> gT -> 'M_n; _ : mx_repr G repr_mx}.
```

Recall that the `%g` is the Coq overloading disambiguation operator. Note that the structure encapsulates both the morphism property, and a specific subgroup on which it holds.

Given `rG : mx_representation G n` we can define the global stabilizer of a row space U, and therefore test whether U is a G-module (i.e., stable under the action of G).

```
Definition rstabs U := [set x \in G | U *m rG x <= U]%MS.
Definition mxmodule U := G \subset rstabs U.
```

Given a G-module U, we can use the matrix bases of U to define a new representation that is the corestriction of `rG` to U, by composing `rG` with the following injection and projection:

```
Definition val_submod m : 'M_(m, \rank U) -> 'M_(m, n) :=
  mulmxr (row_base U).
Definition in_submod m : 'M_(m, n) -> 'M_(m, \rank U) :=
  mulmxr (invmx (row_ebase U) *m pid_mx (\rank U)).
```

Here `mulmxr` A is the function $B \mapsto BA$, and `row_base` U, `row_ebase` U, and `pid_mx` r are the Coq lingo for what was denoted U_R, $U_{\hat{R}}$ and 1_r in section 3.1. We also give a complementary construction for the factor representation `rG`$/U$.

Results in representation theory are alternatively formulated in terms of the representation (rarely), of modules (frequently), and sometimes of algebras. For the latter we use the encoding of section 3.4:

```
Definition enveloping_algebra_mx :=
  \matrix_(i < #|G|) mxvec (rG (enum_val i)).
```

defines the *enveloping algebra* of `rG`. Note how we use the `enum_val` function provided by the `fintype` library to effectively index the matrix rows by elements of G.

Results on modules and algebra often refer to module homomorphisms. Rather than defining a predicate testing whether a linear function f (given as a matrix) is a G-homomorphism on a given submodule U, we find it more convenient to define the largest domain on which f is a G-homomorphism:

```
Definition dom_hom_mx f : 'M_n :=
  let commGf := cent_mx_fun (enveloping_algebra_mx rG) f in
  kermx (lin1_mx (mxvec \o mulmx commGf \o lin_mul_row)).
```

and then test whether U is included in `dom_hom_mx` f, as in this definition of module isomorphism

```
CoInductive mx_iso (U V : 'M_n) : Prop := MxIso f of
  f \in unitmx & (U <= dom_hom_mx f)%MS & (U *m f :=: V)%MS.
```

Note that this definition concerns modules over the *same* representation; we need another predicate `mx_rsim` to state that different representations are similar.

5.2 Simple modules

Many results in group module theory depend on breaking down modules into minimal or *simple* submodules. For example, Schur's lemma states that a non-trivial homomorphism between simple modules yields an isomorphism:

```
Lemma mx_Schur_iso : forall U V f,
    mxsimple U -> mxsimple V -> (U <= dom_hom_mx f)%MS ->
  (U *m f <= V)%MS -> U *m f != 0 -> mx_iso U V.
```

Unlike the `mxmodule` predicate, `mxsimple` is *non-effective*. To test whether modules are simple we need a means of testing whether polynomials are reducible, which we have not assumed. As a consequence we cannot prove constructively within Coq some obvious classical properties, such as the fact that any non-trivial module contains a simple submodule. This turns out to be only a minor nuisance, because we can still prove such facts *classically*:

```
Lemma mxsimple_exists m (U : 'M_(m, n)) : mxmodule U -> U != 0 ->
  classically (exists2 V, mxsimple V & V <= U)%MS.
```

where `classically` is a simple variation on double negation

```
Definition classically P := forall b : bool, (P -> b) -> b.
```

Whenever we are trying to prove an effective property (in `bool`), the SSReflect `without loss` tactic lets us conveniently use such results in a declarative style:

```
without loss [V simV sVU]: / exists2 V, mxsimple V & V <= U.
  exact: mxsimple_exists.
```

We prove classically the existence of module decomposition series, of splitting and closure fields, and of *socles*.

The socle of a representation is the sum of all its simple modules. Within the socle simplicity and isomorphism become decidable, so once a socle is known most constructivity issues vanish. A socle can alternatively be described as the direct sum of the *components* of the representation – the sums of isomorphic simple modules. We define a `socleType` "quasi-structure" that contains enough data to compute components, and coerces uniformly to a type that contains exactly the components.

```
Record socleType := EnumSocle {
  socle_base_enum : seq 'M[F]_n;
  _ : forall M, M \in socle_base_enum -> mxsimple M;
  _ : forall M, mxsimple M -> has (mxsimple_iso M) socle_base_enum
  }.
Definition socle_enum sG := map component_mx (socle_base_enum sG).
Inductive socle_sort sG := PackSocle W of W \in socle_enum sG.
Coercion socle_sort : socleType >-> sortClass.
```

5.3 Some classic results

The framework we have briefly surveyed allows us to formulate and prove all of the basic results in representation theory, including:

```
Lemma mx_Maschke :
  [char F]^'.-group G -> mx_completely_reducible 1%:M.
Theorem Clifford_component_basis : forall M, mxsimple rH M ->
  {t : nat & {x_ : sH -> 'I_t -> gT |
    forall W, let sW := (\sum_j M *m rG (x_ W j))%MS in
      [/\ forall j, x_ W j \in G, (sW :=: W)%MS & mxdirect sW]}}.
Theorem mx_JordanHolder : forall U V compU compV (m := size U),
    (last 0 U :=: last 0 V)%MS ->
  m = size V /\ (exists p : 'S_m, forall i : 'I_m,
    mx_rsim (@series_repr U i compU) (@series_repr V (p i) compV))
Lemma mx_Jacobson_density :
    mx_irreducible rG -> let E_G := enveloping_algebra_mx rG in
  ('C('C(E_G)) <= E_G)%MS.
```

Maeshke's theorem asserts that representations in coprime characteristic are completely reducible; this is classically equivalent but constructively slightly weaker than "semi-reducible". Clifford's theorem explains how an irreducible (i.e., simple) representation of G decomposes into a sum of components when restricted to some $H \triangleleft G$. The Jordan-Hölder theorem asserts the equivalence up to permutation of module composition series U and V (implemented as matrix sequences). In finite dimension, the Jacobson density theorem asserts that the enveloping algebra of an irreducible (i.e., simple) representation is equal to its double centraliser (in infinite dimension equality is replaced by density, hence the name). It combines with Schur's lemma to yield the definition and construction of splitting and closure fields for groups.

The regular representation of a group G interprets G as the basis of a module on which G acts by right translation. If the scalar field of the representation is a splitting field whose characteristic does not divide the order of G, then the Wedderburn structure theorem asserts that the algebra of the regular representation R_G (known as the group ring) decomposes into a direct sum of simple subrings R_i isomorphic to matrix rings. The R_i correspond to the components of the regular representations, so we formalize this result by giving an explicit construction for the R_i given a `socleType sG`, and then establishing all key properties of the construction.

```
Definition Wedderburn_subring (i : sG) := <<i *m R_G>>%MS.
Lemma Wedderburn_sum : (\sum_i R_ i :=: R_G)%MS.
Lemma Wedderburn_direct : mxdirect (\sum_i R_ i)%MS.
Lemma Wedderburn_is_ring : forall i, mxring (R_ i).
Lemma Wedderburn_subring_center i : ('Z(R_ i) :=: mxvec (e_ i))%MS
Lemma rank_Wedderburn_subring i : \rank (R_ i) = (n_ i ^ 2)%N.
Lemma sum_irr_degree : (\sum_i n_ i ^ 2 = nG)%N.
```

We are now using this part of the theory as the basis for the formalization of character theory needed for the second part of the Feit-Thompson Theorem proof[4].

5.4 *p*-stability and extraspecial representations

One of the early driving applications for our work on matrix linear algebra and representations was the study of p-stability, an important technical property of groups of odd order that underpins the proof of the two "deep" results on which the first part of the Feit-Thompson Theorem proof is based, the Thompson Transitivity and Uniqueness theorems[3]. The variant of p-stability we are interested in is an extension to groups G with a non-trivial p-core $O_p(G)$ of the property of "no p-element of G has a quadratic minimal polynomial in a faithful representation with a characteristic p field", whose rather technical formulation translates in Coq as

```
Definition p_stable p G :=
  forall P A : {group gT},
     p.-group P -> 'O_p^'(G) * P <| G ->
     p.-subgroup('N_G(P)) A -> [~: P, A, A] = 1 ->
  A / 'C_G(P) \subset 'O_p('N_G(P) / 'C_G(P)).
Theorem odd_p_stable : forall gT p G, odd #|G| -> p_stable p G.
```

The proof of this theorem is about 300 lines long, and summarizes about 6 pages of the textbook it is drawn from[2], with some improvements (like eliminating "proof by ellipsis").

The most challenging representation theory result in [3] was Theorem 2.5, whose 240-line proof uses representation theory to derive numerical properties of the orders of a specific class of groups (semidirect products of a cyclic group acting in a prime manner on an extraspecial p-group).

```
Theorem repr_extraspecial_prime_sdprod_cycle :
   forall p n gT (G P H : {group gT}),
   p.-group P -> extraspecial P -> P ><| H = G -> cyclic H ->
   let h := #|H| in #|P| = (p ^ n.*2.+1)%N -> coprime p h ->
   {in H^#, forall x, 'C_P[x] = 'Z(P)} ->
 [/\ (h %| p ^ n + 1) || (h %| p ^ n - 1)
  & (h != p ^ n + 1)%N ->
      forall F q (rG : mx_representation F G q),
    [char F]^'.-group G -> mx_faithful rG -> rfix_mx rG H != 0)].
```

References

1. Lang, S.: Algebra. Springer-Verlag (2002)
2. Gorenstein, D.: Finite groups. Second edn. Chelsea, New York (1980)
3. Bender, H., Glauberman, G.: Local analysis for the Odd Order Theorem. Number 188 in London Mathematical Society Lecture Note Series. Cambridge University Press (1994)
4. Peterfalvi, T.: Character Theory for the Odd Order Theorem. Number 272 in London Mathematical Society Lecture Note Series. Cambridge University Press (2000)
5. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. In: Theorem Proving in Higher-Order Logics. Volume 4732 of LNCS. (2007) 86–101
6. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical big operators. In: Theorem Proving in Higher-Order Logics. Volume 5170 of LNCS. (2008) 86–101
7. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Theorem Proving in Higher-Order Logics. Volume 5674 of LNCS. (2009) 327–342
8. Pottier, L.: User contributions in Coq, Algebra (1999) Available at `http://coq.inria.fr/contribs/Algebra.html`.
9. Blanqui, F., Coupet-grimal, S., Delobel, W., Koprowski, A.: Color: a coq library on rewriting and termination (2006) Eighth Int. Workshop on Termination (WST 06), to appear in MSCS.
10. Rudnicki, P., Schwarzweller, C., Trybulec, A.: Commutative algebra in the Mizar system. J. Symb. Comput. **32**(1) (2001) 143–169
11. Obua, S.: Proving Bounds for Real Linear Programs in Isabelle/HOL. In: Theorem Proving in Higher-Order Logics. (2005) 227–244
12. Harrison, J.: A HOL Theory of Euclidian Space. In Hurd, J., Melham, T.F., eds.: TPHOLs. Volume 3603 of LNCS., Springer (2005) 114–129
13. Cowles, J., Gamboa, R., Baalen, J.V.: Using ACL2 Arrays to Formalize Matrix Algebra. In: ACL2 Workshop. (2003)
14. Stein, J.: Documentation for the formalization of Linerar Agebra `http://www.cs.ru.nl/~jasper/`.
15. Coq development team: The Coq Proof Assistant Reference Manual, version 8.3. (2010)
16. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical report, `http://hal.inria.fr/inria-00258384`.
17. Sozeau, M., Oury, N.: First-Class Type Classes. In: Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (August 2008) 278–293
18. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming **13**(2) (2003) 261–293