Canonical big operators

Yves Bertot, Georges Gonthier†, Sidi Ould Biha, and Ioana Pasca

INRIA Microsoft Research†
[Yves.Bertot|Sidi.Ould_Biha|Ioana.Pasca]@sophia.inria.fr
gonthier@microsoft.com†

Abstract. In this paper, we present an approach to describe uniformly iterated "big" operations, like $\sum_{i=0}^{n} f(i)$ or $\max_{i \in I} f(i)$ and to provide lemmas that encapsulate all the commonly used reasoning steps on these constructs.
We show that these iterated operations can be handled generically using the syntactic notation and canonical structure facilities provided by the Coq system. We then show how these canonical big operations played a crucial enabling role in the study of various parts of linear algebra and multi-dimensional real analysis, as illustrated by the formal proofs of the properties of determinants, of the Cayley-Hamilton theorem and of Kantorovitch's theorem.

1 Introduction

One of the most versatile tools of the working mathematician is the "big operator" notation. At the stroke of a `\bigxx` LaTeX macro, she gets a bird's eye view of the algebra of her problem, revealing hidden symmetries, which she can immediately exploit using a rich set of partitioning, reindexing, and commutation operations.

So far, big operators have been missing from the toolbox of the formal mathematician, at least in their full generality, that is, allowing big of any operator indexed in any way, such as

$$\sum_{d|n} \phi(n/d)m^d \quad \text{or} \quad \bigoplus_{V_i \simeq W} V_i$$

We report here on the design of a generic big operator library for the Coq proof system [4, 2]. This development was motivated and honed by the proof of several advanced results in algebra and analysis, which we also present.

This library is not just a collection of notations, although we do make good use of Coq's facilities in this respect. It contains a generic theory of big operators, including unique lemmas that perform complex operations such as reindexing and dependent commutation, for all operators, with minimal user input and under minimal assumptions.

Critically, the library relies on Coq's canonical structures (described below) for expressing structural and algebraic properties of indices and operators. This allows rewriting and resolution to infer such properties automatically, which

is essential for the library to be usable in practice. Although similar, neither dependent record subtyping nor axiomatic type classes would support this style of operator-centric inference.

The paper is organized as follows. In Section 2, we describe and use Coq's canonical structures to create the level foundation on which we will build our big operator theory, while in Section 3 we use Coq's syntactic notation facility to map a wide range of big operator forms to a single generic function; in Section 4 we develop a library of generic lemmas that can handle most of the common algebraic and logical operations on these forms. Finally, in Sections 5, 6, and 7, we put this library to work in the formalization of some classical results in algebra and analysis, including the Cauchy determinant formula, the Cayley-Hamilton theorem, and Kantorovitch's theorem.

2 Canonical structures

Building a generic library that can accommodate a large variety of iterated operators requires more than notation — although the latter does play an important role, as we shall see in the next section. It calls for a logical framework that can express and classify the key properties of the two main components of big operators, namely, indexes and operations.

We implement this framework with Coq's Canonical Structure declaration, which we use in a new and nonstandard way. Although specific to Coq, the Canonical Structures are fairly close to record subtyping and type classes [15], so our approach could be ported to other systems, given some minor extensions.

For indices we actually reuse combinatorial structures that were developed for the Four Colour Theorem proof, and used in our finite group library [7], so the next section is a review of material from [7, 6] that can also serve as an introduction to Canonical Structures.

2.1 Index Structures

We want to handle big operators indexed by arbitrary types. However, we need to compare and possibly enumerate indices to compute big operators, so the indices must have enriched types. In an object-oriented setting this could be achieved by subtyping; it is well-known that in higher-order logic nested dependent records (aka telescopes) can be used instead [11, 16].

For example we can describe comparable ("equality") types as follows: [1]

```
Structure eqType : Type := EqType {
    sort :> Type;
   eqd : sort -> sort -> bool;
       _ : forall x y, (x == y) <-> (x = y)
} where "x == y" := (eqd x y).
```

[1] The actual code uses a mixin/class presentation and handles Coq technicalities like namespace management and reduction hints.

The `:>` symbol makes `sort` into a coercion, which means we can use a `T : eqType` as if it were a `Type` — type inference will insert the missing `sort` projection. The structure lets us define a unified notation for the generic comparison function `eqd`. Moreover every `eqType` contains an axiom stating that the comparison function *reflects* actual (Leibnitz) equality, so it is valid to rewrite `x` into `y` given `x == y` (i.e., (`x == y`)= `true`). Indeed, equality is decidable (and therefore proof-irrelevant [1]) for all `eqType`s.

Unlike telescopes, Coq structures are not restricted to abstract types, and can be created for *existing* types. For example, if we can prove

```
Lemma eqnP : forall m n, eqn m n <-> m = n.
```

for an appropriate function `eqn : nat -> nat -> bool`, then we can make `nat`, a `Type`, behave as an `eqType` by declaring

```
Canonical Structure nat_eqType := EqType eqnP.
```

This creates a new `eqType` with `sort` $\equiv$ `nat` and `eqd` $\equiv$ `eqn` (both are inferred from the type of `eqnP`). Dually to coercions, this declaration allows `nat` to behave as `sort nat_eqType` during type inference. This lets Coq interpret `2 == n` as (`eqn 2 n`), because it can solve the unification equation `sort` $?e \equiv_{\beta\delta\iota}$ `nat` and then evaluate `eqd nat_eqType` to `eqn`.

These details are crucial for the next section. However, the casual user can mostly gloss over them; he only cares that canonical structures let him use generic constructions and properties for specific instances, similarly to type classes.

The computation of a big operator must also enumerate the indices in its range. This is trivial if the range is an explicit sequence of type `seq I`, where `I` has an `eqType` structure, e.g., if the range is a `nat` interval. However, it is often more convenient to specify the range implicitly by a predicate, in which case the computation must be able to enumerate the entire index type, which must thus be *finite*. The following structure supports this capability:

```
Structure finType : Type := FinType {
  sort :> eqType;
  enum : seq sort;
  _ : forall x, count (fun y => y == x) enum = 1
}.
```

The axiom asserts that every value of the type occurs exactly once in `enum`.

This structure is very good for working with finite sets; we have a rich `finType`-based library of over 250 lemmas [7], which includes the construction of a type `ordinal n` (denoted `I_(n)`) of integers $0 \leq i < $ `n`, of function and set types over a `finType`, as well as canonical `eqType` and `finType` structures for all of these.

2.2 Operator structures

Genericity over operations is more difficult to achieve, but probably more important than genericity over indices. We want to be able to use our library for all

kinds of types and operations, from simple integer sums to GCDs of polynomials. Using telescopes here would essentially amount to identifying big operators with generalized summations over a type with the following structure:

```
Structure additive_group : Type := AdditiveGroup {
  sort :> eqType;          zero : sort;
  opp : sort -> sort;      add : sort -> sort -> sort;
  _ : associative add;     _ : commutative add;
  _ : left_unit zero add;  _ : left_inverse zero opp add
}.
```

However this would be wrong, for several reasons:

1. It imposes strong axioms, which will not hold for many interesting operators, such as max over nat. Simply refining the telescope to take into account the many relevant axiom sets, from non-commutative monoids up to commutative rings, would lead to an uncomfortably deep hierarchy. The latter would cause type inference to generate terms bloated with long projection chains.
2. It makes the representation of the summation depend on the *proofs* that the operator satisfies algebraic properties, and by extension on any data on which these proofs might depend. This artificial dependency could only be broken by breaking the summation abstraction thereby losing all generic notations and properties.
3. It is only parametric in the operator *type*, not the operator itself, so we could only have at most one generic big operator per type. This is inadequate even for abstract rings, where we want both sums and products, and woefully inadequate for integers, which have sum, product, min, max, GCD and LCM!

The proper solution to this parametricity problem lies in the observation that unlike type classes or telescopes, canonical structures can be used to enrich not only types, but arbitrary values. Indeed, this was already the case for `finType` and `additive_group`, which both enriched the `eqType` structure — a *record*.

This fact allows us to define iteration for arbitrary operators, because we can use a structure to meet the algebraic requirements of our lemmas. For instance we define

```
Structure law T unit : Type := Law {
  operator : T -> T -> T;   mul1m : left_unit unit operator;
  _ : associative operator; mulm1 : right_unit unit operator
}.
```

and then

```
Canonical Structure andb_monoid := Law andbA andTb andbT.
Canonical Structure addn_monoid := Law addnA add0n addn0.
Canonical Structure gcdn_monoid := Law gcdnA gcd0n gcdn0.
...
```

This simple series of canonical structure declarations lets Coq know that boolean conjunction, integer addition and *GCD*, etc, are monoidal laws, so that it can automatically discharge this condition when a lemma or rewrite rule is used.

We define similar structures for abelian monoids and semirings; note that nesting depth (issue 1 above) is not a problem here as these structures appear in the proof terms only, not in the big operator expression.

3 Notations

To capture the commonalities between all possible big operators, we provide a host of notations that are independent from the operator being used, the operator and the value for the empty range being given as parameters. Thus the notation has the following shape:

`\big [ op / nil ]_ (`*`index and range description`*`) F`

3.1 Range descriptions

The part called *index and range description* is responsible for giving the name of the bound variable and stating the set over which this variable is supposed to range. There are mainly three ways to give the range: take a list of values that have to be covered, take a specific integer interval, or take the entire type of the bound variable (which must then be a `finType`) or a subset thereof. We use the notation (`i <- r`) to range over a list, the notations (`m <= i < n`) or (`i < n`) to range over an interval, the notations (`i`) or (`i : t`) to range over the entire index type, and the notation (`i \in A`) to range over a subset. In all cases, the variable `i` is bound in `F`.

On top of these variants, we choose to add the possibility to filter the range with a predicate, meaning that the big operator takes only the elements of the range that satisfy the predicate. This is simply written by adding `| P` at the end of the index and range description. Again, the variable `i` is bound in the formula `P`. Thus, the following notation represents the addition of all squares of even numbers between 0 and $n - 1$.

```
\big[addn/0]_(i < n | even i) i^2
```

Since natural numbers in an interval can easily be enumerated, all notations reduce to the same function, where the range is a list of values that do not need to belong to a finite type and a filtering predicate is always provided. This notation is implemented by the following code:

```
Definition reducebig R I op nil r (P : pred I) (F : I -> R) : R :=
  foldr (fun i x => if P i then op (F i) x else x) nil r.

Notation "\big [ op / nil ]_ ( i <- r | P ) F" :=
  (reducebig op nil r (fun i => P%B) (fun i => F)) : big_scope.
```

It is a simple structural recursive function which follows the structure of the list `r` and tests whether `P` is satisfied on the first element to decide whether the value of `F` on this element is combined with the value computed for the rest; at the end of the list the `nil` value is used.

3.2 Operator inference

We then define other notation that is specialized for the case where the operator satisfies a particular structure. For a variable ranging on a type, the various patterns are as follows:

- `\sum_(i) F` is used when the result type is `nat`, `nil` is 0 and the operator is `nat` addition, or when the operator is the `add` field of an `additive_group` structure (see 2.2) whose `zero` is `nil`. Thus, when the type of formula `F` is `sort` of a canonical `additive_group` structure, this operation is automatically understood as the iteration of its additive law.
- `\prod_(i) F` is used when the result type is `nat`, `nil` is 1 and the operator is `nat` multiplication, or when the operator is the multiplication of a ring or group with unit `nil`.
- `\max_(i) F` is used when the operator is the `nat` binary max and `nil` is 0.

Note that the denoted term is always of the form `reducebig` ..., so generic lemmas apply uniformly regardless of which notation is used to for the range and operator.

4 Main lemmas

Canonical structures play a crucial role when organizing the large library of lemmas that we provide to reason about big operations (there are around 80 lemmas). A first collection of lemmas helps reasoning about big operations without any assumption on the operator being iterated. Other collections of lemmas are for operators that respect a plain monoid structure (with only associativity and a neutral element), an abelian monoid structure (with commutativity) or a semi-ring structure (where two operators interact through distributivity).

For instance, lemmas applicable to a monoid operator handle a big operator where `op` has the form `operator` l and require l to have the type `law`, while lemmas applicable to an abelian monoid operator handle a big operator where `op` has the form `operator` (`law_of_abelian` l) and require l to have the type `abelian_law`. For a given operator `op` that is both associative and commutative and has a neutral element, two canonical structures are constructed, one with type `law` and the other with type `abelian_law`. The user always writes `\big[op/nil]_...`; when a lemma requiring associativity is applied the corresponding canonical structure is automatically inferred. Thus, we have a single notation that is independent of properties satisfied by operators; lemmas refer to properties through records, and we use canonical structures to reconcile the two approaches at the time we use the lemmas. This will be apparent as we study in more detail some of the lemmas.

4.1 Lemmas for plain operators

Operations like replacing the general term or predicate of a big operation by an equivalent one or unmapping its index range do not require any property of the operator.

To cope with rewriting in parts of a big operation, we provide a variety of congruence lemmas. Here is one example, which can be used to express that we can rewrite in the predicate and the formula parts of a big operation.

```
Lemma eq_big : forall (r : seq I) (P1 P2 : pred I) F1 F2,
  P1 =1 P2 -> {in P1, F1 =1 F2} ->
  \big[op/nil]_(i <- r | P1 i) F1 i =
  \big[op/nil]_(i <- r | P2 i) F2 i.
```

This lemma expresses that two big operations can be proved equal even though their predicates and formulas may appear to be different. The first premise `P1 =1 P2` expresses that it suffices that the predicates are extensionally equal (the two functions are equal on every argument), the second premise {`in P1`, `F1 =1 F2`} that it suffices that the two formulas are extensionnally equal on the subset of the type determined by the predicate `P1`.

Other collections of lemmas concern rewritings that occur simultaneously in the range and in some other part of the big operation. For instance, a combined rewriting in the range and the formula makes it possible to change all elements in the range list, compensating by a composition in the formula and the filtering predicate:

```
Lemma big_maps : forall (J : eqType) (h : J -> I) r F P,
  \big[op/nil]_(i <- maps h r | P i) F i =
  \big[op/nil]_(j <- r | P (h j)) F (h j).
```

We also have lemmas that make it possible to change the length of the range: we can assert that a sum up to n_1 is equal to a sum up to n_2, with $n_1 \leq n_2$, if the predicate filters out all numbers that are larger than or equal to n_1 and smaller than n_2:

```
Lemma big_nat_widen : forall m n1 n2 P F, n1 <= n2 ->
  \big[op/nil]_(m <= i < n1 | P i) F i =
  \big[op/nil]_(m <= i < n2 | P i && (i < n1)) F i.
```

4.2 Plain monoid re-indexing

When the iterated operation is associative and the nil value is the neutral element, nicer decomposition lemmas can be obtained. To express that the operator is a monoid law we use a notation `*%M`. We also use a specific notation for the `nil` value, but this is only to enhance readability. For instance, we can state a lemma that helps decomposing a list range in two sub-lists, where ++ stands for the concatenation of lists:

```
Lemma big_cat : forall I (r1 r2 : seq I) P F,
  \big[*%M/1]_(i <- r1 ++ r2 | P i) F i =
  \big[*%M/1]_(i <- r1 | P i) F i *
    \big[*%M/1]_(i <- r2 | P i) F i.
```

which would be written in standard mathematics:

$$\prod_{i \in r_1 \cup r_2, P_i} F_i = \prod_{i \in r_1, P_i} F_i * \prod_{i \in r_2, P_i} F_i$$

We actually provide half a dozen lemmas that are specific, to monoidal laws.

4.3 Abelian monoid re-indexing

To handle commutative monoidal operators, we redefine our notation $*\%\mathtt{M}$ to express that it has to be the operator of an abelian monoidal law. This is done with the following notation declaration:

```
Notation Local "*%M" := (operator (law_of_abelian op)).
```

In this case, permuting elements in the range or grouping them according to a partition becomes possible. Here are two of the main lemmas, concerned with partitioning an index set and with swapping nested sum operators.

To describe partitions, we use an auxiliary function and view each subset in the partition as the inverse image of one element:

```
Lemma partition_big :
  forall (I J : finType) (P : pred I) p (Q : pred J) F,
  (forall i, P i -> Q (p i)) ->
  \big[*%M/1]_(i | P i) F i =
  \big[*%M/1]_(j | Q j) \big[*%M/1]_(i | P i && (p i == j)) F i.
```

$$(\forall i, P_i \to Q_{p(i)}) \to \prod_{i \in I, P_i} F_i = \prod_{j \in J, Q_j} \prod_{\substack{i \in I \\ P_i \wedge p(i) = j}} F_i$$

To permute nested sum operators, we start by showing that that nested big operations can be reduced to a single big operation where pairs of indices are enumerated. Through a re-indexing operation on the pairs, we then obtain a variety of commutation lemmas, of which we show only the simplest one:

```
Lemma exchange_big : forall (I J : finType) P Q F,
  \big[*%M/1]_(i : I | P i) \big[*%M/1]_(j : J | Q j) F i j =
  \big[*%M/1]_(j | Q j) \big[*%M/1]_(i | P i) F i j.
```

$$\prod_{i \in I, P_i} \prod_{j \in J, Q_j} F_{i,j} = \prod_{j \in J, Q_j} \prod_{i \in I, P_i} F_{i,j}$$

4.4 Distributivity

Distributivity plays a role when several operators interact, usually in a semi-ring structure. Here we adapt our notation so that $*$%M refers to the multiplication operation of a semi-ring and +%M refers to the addition of the same semi-ring. Here is a first simple lemma:

```
Lemma big_distrl : forall I (r : seq I) alpha P F,
  (\big[+%M/0]_(i <- r | P i) F i) * alpha =
  \big[+%M/0]_(i <- r | P i) (F i * alpha).
```

In general, big products of big sums can be transformed into big sums of big products: this is another form of swapping lemma that gives rise to pairs of indices. Here is one of our lemmas to handle this, where {ffun I −> J} describes the set of all functional graphs in I*J (a finite type that actually describes all functions from the finite type I to the finite type J):

```
Lemma bigA_distr_bigA :
  forall (I J : finType) F,
  \big[*%M/1]_(i : I) \big[+%M/0]_(j : J) F i j =
  \big[+%M/0]_(f : {ffun I -> J}) \big[*%M/1]_(i) F i (f i).
```

It is remarkable that none of these lemmas requires a proof that C1, the value of empty "big products", actually be the neutral element for multiplication.

5 Some results on determinants

The first motivating example for our big operator library was the study of determinants; it uses many key features of the library, including the compact notation, generic indexing, and reindexing and swapping lemmas.

5.1 The Leibnitz formula

While in practice determinants are best computed from a triangular decomposition, or by using Laplace's formula to expand with respect to a fixed row, it is impractical to derive any of the theoretical properties of determinants from such expressions because of their lack of symmetry. In contrast, the highly symmetrical but impractical Leibnitz formula calls for summing over permutations; our generic library handles this quite gracefully:

```
Definition determinant n (A : M_(n)) :=
  \sum_(s : S_(n)) (-1)^+s * \prod_(i) A i (s i).
```

The actual Coq proofs that this definition yields a multilinear alternate form are only 7 and 14 lines long, respectively; the proof of the Laplace formula is 80 lines (most of which compute the parity of a cyclic permutation), but then we only need 16 lines to prove the Cramer rule:

$$A.\mathrm{adj}\ A = \mathrm{adj}\ A.A = \det A.\mathrm{Id} \tag{1}$$

5.2 The Cauchy formula

The Cauchy formula simply states that the determinant commutes with matrix product. It is fairly tricky to establish rigorously for abstract rings; here is a self-contained proof, for $n \times n$ matrices:

$$\begin{aligned}
\det AB &= \sum_{\sigma \in S_n} (-1)^\sigma \prod_i \left(\sum_j A_{ij} B_{j\sigma(i)} \right) \\
&= \sum_{\phi:\ [1,n] \to [1,n]} \sum_{\sigma \in S_n} (-1)^\sigma \prod_i A_{i\phi(i)} B_{\phi(i)\sigma(i)} \\
&= \sum_{\phi \notin S_n} \sum_{\sigma \in S_n} (-1)^\sigma \prod_i A_{i\phi(i)} B_{\phi(i)\sigma(i)} + \sum_{\phi \in S_n} \sum_{\sigma \in S_n} (-1)^\sigma \prod_i A_{i\phi(i)} B_{\phi(i)\sigma(i)} \\
&= \sum_{\phi \notin S_n} \left(\prod_i A_{i\phi(i)} \right) \sum_{\sigma \in S_n} (-1)^\sigma \prod_i B_{\phi(i)\sigma(i)} \\
&\quad + \sum_{\phi \in S_n} (-1)^\phi \left(\prod_i A_{i\phi(i)} \right) \sum_{\sigma \in S_n} (-1)^{\phi^{-1}\sigma} \prod_k B_{k\sigma(\phi^{-1}(k))} \\
&= \sum_{\phi \notin S_n} \left(\prod_i A_{i\phi(i)} \right) \det \left(B_{\phi(i)j} \right)_{ij} + (\det A) \sum_{\tau \in S_n} (-1)^\tau \prod_k B_{k\tau(k)} \\
&= 0 + (\det A)(\det B)
\end{aligned}$$

The first step swaps the iterated product of the Leibnitz formula with the sum in the general term of the matrix product, generating a sum over all functions from indices to indices. This is split into a sum over non-injective functions and a sum over permutations. The former is rearranged into a weighted sum of determinants of matrices with repeated rows, while the latter is reindexed, using the group properties of permutations, to become the desired product of determinants.

Remarkably, the formal Coq proof is only 25 lines long, and actually shorter than the above proof sketch, because all of the required sum manipulations are directly supported by the library, and our previous work on finite groups [7] supplies the all required permutation facts.

6 The Cayley-Hamilton Theorem

After proving the Cramer rule, the next step was formalizing the Cayley-Hamilton theorem [3]. For a commutative ring R and a square matrix A on R, this theorem states that A is a root of its characteristic polynomial $p_A(x) = \det(xI_n - A)$.

To prove the Cayley-Hamilton theorem, we apply the Cramer rule (1) to the $(xI_n - A) \in M_n(R[x])$ and we obtain:

$$\text{adj}\,(xI_n - A) * (xI_n - A) = \det(xI_n - A) * I_n = p_A(x) * I_n \tag{2}$$

This is an equality in $M_n(R[x])$. However the ring $M_n(R[x])$ of matrices with polynomial coefficients and the ring of polynomials with matrix coefficients $(M_n(R))[x]$ are isomorphic. For example, the following equality exhibits the correspondence:

$$\begin{pmatrix} x^2+1 & x-2 \\ -x+3 & 2x-4 \end{pmatrix} = x^2 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + x \begin{pmatrix} 0 & 1 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & -2 \\ 3 & -4 \end{pmatrix}$$

We call $\phi : M_n(R[x]) \to (M_n(R))[x]$ the isomorphism from one ring to the other. In $(M_n(R))[x]$, the equality (2) is written :

$$\phi(\text{adj } (xI_n - A)) * (x - A) = p_A^{\phi}(x) \tag{3}$$

where $p_A^{\phi}(x)$ is in fact the polynomial with scalar matrix coefficients obtained by applying ϕ to $p_A(x) * I_n$. This shows that $(x - A)$ is a factor of $p_A^{\phi}(x)$ in $(M_n(R))[x]$, so $p_A^{\phi}(A) = O_n$.

To formalize this proof, we developed a library to describe polynomials.

6.1 Polynomials

A polynomial is formally defined by the list of its coefficients a_i which are elements of a ring R :

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

It is natural to represent a polynomial with the list $(a_0, \ldots, a_n)$; however, it is also handy to use polynomials as functions of type $nat \to R$, which return 0 almost everywhere.

We can easily change from one representation to the other by using a lemma that states that two polynomials are equal if their functional representations are extensionally equal. The list representation is used to define the operations on polynomials by induction on the coefficient list. With the representation as function of type $nat \to R$, we can reuse lemmas on big operators to prove algebraic properties of polynomials, in a style that is close to standard mathematics. For example the following property of the coefficients of the product of two polynomials is expressed using a big sum.

```
Lemma coef_mul_poly : forall p1 p2 i,
  coef (p1 * p2) i = \sum_(j < i.+1) coef p1 j * coef p2 (i - j).
```

With this new point of view, we prove the associativity of polynomial multiplication by simply reusing re-indexation and distributivity lemmas for big operators.

In the following, the notation `\poly_(i < n) E`, where `E` is an expression on `i`, corresponds to the polynomial $\sum_{i<n} E_i x^i$. The notations `\X`, `\C c` and `p.[c]` correspond respectively to the monomial `x`, the constant polynomial `c` and the evaluation of a polynomial `p` in a value `c`.

In the polynomials library we give a proof of the factor theorem :

```
Theorem factor_theorem : forall p c,
  (exists q, p = q * (\X - \C c)) <-> (p.[c] = 0).
```

We proved the equivalence, but we only need the implication from left to right for the Cayley-Hamilton theorem. This proof is only 12 lines long, thanks to the lemmas on big operators.

6.2 Proving the Cayley-Hamilton theorem

The morphism between the ring of matrices of polynomials and the ring of polynomials of matrices is the central part of the proof. It is best described using big operators:

```
Definition phi (A : M(R[X])) : M(R)[X] :=
  \poly_(k < \max_(i) \max_(j) size (A i j))
     \matrix_(i, j) coef (A i j) k.
```

In this formula, the notation `\matrix_(i, j) E` denotes the matrix whose coefficient at position (`i`, `j`) is described by the expression `E`. The length of the resulting polynomial is the maximum size of coefficient lists in the input matrix, described with the `\max` operator. Big operator lemmas are also instrumental in the proofs of morphism properties for ϕ.

The characteristic polynomial is defined as follow :

```
Definition char_poly (A : M(R)) : R[X] := \det(\Z \X - matrixC A).
```

In this formula `\Z` stands for the scalar multiplication by the identity matrix. We also define `Zpoly` as the canonical injection from the ring of polynomials with scalar coefficients into the ring of polynomials with matrix coefficient. With these definitions the Cayley-Hamilton theorem has the following statement.

```
Theorem Cayley_Hamilton : forall A, (Zpoly (char_poly A)).[A] = 0.
```

The main proof is done in three lines.

7 Multivariate real analysis and Kantorovitch's theorem

We also conducted an experiment in giving a complete formalization for Kantorovitch's theorem [14]. This theorem in numerical analysis gives sufficient conditions for the convergence of Newton's method for finding the root of a function $f : \mathbb{R}^p \to \mathbb{R}^p$. The main challenge was to find a representation for vectors in $\mathbb{R}^p$ and formalize multivariate analysis concepts.

After a careful analysis, the choice was made to base this formalization on the `Reals` from standard Coq and on the SSReflect extension. This choice turned out to be adequate for vectors, matrices, and the use of big operations to abstract over dimensions.

We provide a canonical structure of field for $\mathbb{R}$ and encode vectors as functional graphs from a finite type of dimension p to $\mathbb{R}$. In practice, this gives both

a view of vectors as lists and vectors as functions over the index type, thus facilitating the description of component-wise operations.

We then simply formalize a norm on vectors as a big operation. The norm is $\|x\| = \max_i |x_i|$, which in Coq can be expressed as:

```
Definition norm (v : Rvec p) := \big[Rmax/0]_(i < p) Rabs (v i).
```

With this definition, a lemma stating the positivity of the norm

```
Lemma norm_pos : forall v, 0 <= norm v.
```

is easily proved by applying a generic lemma named `big_prop`. This induction scheme states that a property which is closed with respect to the operator, satisfied by the `nil` value, and by the formula for every index is also satisfied by the result of big operation. In this case, the property is positiveness. Other required properties for norms have about the same level of complexity.

Nevertheless, the use of the maximum as an indexed operation posed some difficulties. As stated before, the lemmas on big operations are organized in a sort of hierarchy following the algebraic structure given by the operator. In the case of the maximum, we have associativity and commutativity, but we do not have a neutral element on the type of real numbers. Since we work only with positive numbers (and the maximum on this subset has 0 for neutral element), we would like to be able to use the lemmas that deal with an abelian monoid structure.

There are two possible solutions for this problem. The first is to have a new type for positive reals. We can define the canonical structure of abelian monoid on this new type, manipulate the indexed operation as desired and inject the result in the original type. The second solution is to define a new operator that gives the type the desired structure. This operator has to be equal to the original one on the target subset (here, the positive reals). Such a change of operator is covered in the library by a lemma called `eq_big_op`.

We adopted the second approach, as we had this definition at hand:

$$\max'\ x\ y = \begin{cases} \max\ x\ y & \text{if } x > 0 \text{ or } y > 0; \\ \min\ x\ y & \text{if } x, y \leq 0 \end{cases}$$

However, we could easily have fallen back to the first solution if such a construction had not been available.

Another interesting example regards the decomposition of a vector to prove a multi-dimensional variant of the mean value theorem.

$$\begin{aligned} f(x_1, \ldots, x_p) - f(y_1, \ldots, y_p) &= f(x_1, \ldots, x_p) - f(y_1, x_2, \ldots, x_p) + \\ f(y_1, x_2, \ldots, x_p) - f(y_1, y_2, x_3, \ldots, x_p) &+ \ldots + f(y_1, \ldots, y_{p-1}, x_p) - f(y_1, \ldots, y_p) \\ &= \sum_{i=1}^{p} (x_i - y_i) \frac{\partial f(y_1, \ldots, y_{i-1}, c_i, x_{i+1}, \ldots, x_p)}{\partial x_i} \end{aligned}$$

This simple and elegant proof goes through naturally in our formalization.

During the development we also needed a formalization of matrices in order to represent, for example, the Jacobian of a partially derivable function. We used the `matrix` library developed during the formalization of the Cayley-Hamilton theorem, which we enriched with additional concepts, like the norm of a matrix, compatible with our vector norm: $\|A\| = \max_i \sum_j |a_{ij}|$.

Most of the lemmas we have described so far are concerned with equality, but results about norms also exhibit the need for lemmas concerned with binary relations. For instance, we use a lemma named `big_rel` which states that if a relation R is reflexive and transitive, R satisfies some stability condition with respect to the operator, and formulas F and G are related by R for every index, then the big operation on F is related by R with the big operation on G. Such a lemma is instrumental in the proof of the following results:

$$\|AB\| \leq \|A\|\|B\| \quad \text{and} \quad \|A\| < 1 \rightarrow \det(I_p - A) \neq 0$$

The first result relies on `big_rel` and distributivity lemmas, while the second result relies on the convergence of a series of matrices. One of the intermediate lemmas for the second result is expressed as follows:

```
Lemma mat_norm_sum : forall (A : nat -> MR(p)) n,
  norm (\sum_(i <= n) A i) <= \sum_(i <= n) norm (A i).
```

This lemma is a direct consequence of one of the generic lemmas from the library, named `big_morph_rel`: it suffices to show that `norm` has a morphism-like property with respect to the relation <=, addition of matrices (on the left), and addition of real numbers (on the right).

8 Conclusion

This work is based on the SSReflect extension of Coq [6]. This extension relies extensively on canonical structures and reflexion. The work described in this paper is available on Internet at: `www-sop.inria.fr/marelle/bigops`.

8.1 Related work

The HOL-Light system [8] also provides generic iterated operations and applications to multi-dimensional spaces. Separate work of T. Hales and J. Harrison [9] provide formalizations of euclidean space.

HOL-Light lacks dependent types but does not restrict itself to constructive logic. As a result, finite types cannot be described as records like our `finType` and iteration is actually defined on subsets of infinite types. Properties are mainly provided for abelian monoidal laws with a neutral element. This approach is less generic than ours, but it is already strong enough for many results. In particular, the system library contains results for real matrices and determinants similar to ours, but its applications do not go all the way to the Cayley-Hamilton theorem. Our work lives in a different setting: the main part is done in constructive logic

with dependent types and we use enumerations for finite types which allows us to define big operations for plain operators.

Work by Gamboa, Cowles, and van Baalen also describes matrix computations in ACL2 [5]; they don't make a systematic use of big operations and determinants are described through a process of gaussian elimination, but almost no properties are proved.

In the Coq system N. Magaud implemented vectors and matrices as dependent lists [12] but this is mainly an exercise in dependent types. J. Stein [17] and S. Obua [13] also describe linear algebra using big operations with monoid laws, for instance for matrix multiplication, but do not study determinants.

8.2 Overview and perspectives

A commonly held opinion is that the formalization of mathematics is a long and difficult process for two reasons: first, more detail is required than in standard mathematical proofs and second, the formalized corpus is too small as a foundation, so that many lemmas have to be re-proved before addressing significant results. This opinion overlooks an important area where progress can be made, the area of infra-structure. Infra-structure can help in formalizing mathematics if statements and proofs can be expressed concisely and if the details can be collected automatically. This paper brings a contribution to the infra-structure aspect of formalizing mathematics.

We also bring a collection of lemmas organized in a way that increases their reusability drastically and we illustrate the gain with big operators for proofs on the properties of determinants and matrices. We feel we can approach new landmarks that were hitherto considered out of reach like the Cayley-Hamilton theorem. In the Mathematical Components project [7] we also reuse big operations to study generated groups.

Two questions come to mind: if this library on big operators has such a positive and structuring impact, what is the infrastructure behind the it that makes it so powerful? What is the next concept that deserves a systematic treatment and will have the same structuring effect?

To answer the first question, we propose to consider canonical structures as the key advance. First proposed by Saïbi in his study of category theory [10], these structures are instrumental here as they take over the automatic search for relevant information attached to each operator. Also, we propose to use canonical structures to attach properties to *operators*, while usual approaches attach properties to *types*. We can now write big operations simply, the required properties are inferred from the canonical structure declarations when applying lemmas.

We can't actually answer the second question yet, but we believe that big operations have opened the road to a re-newed study of linear algebra, with notions like bases, linear combinations, and so on, or of algorithms in other parts of algebra, like the algorithm of sub-resultants, the proof of which already relied on an abstract notion of determinants.

References

1. Franco Barbanera and Stefano Berardi. Proof-irrelevance out of Excluded-middle and Choice in the Calculus of Constructions. *Journal of Functional Programming*, 6(3):519–525, 1996.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.
3. Sidi Ould Biha. Formalisation des mathématiques : une preuve du théorème de Cayley-Hamilton. In *Journées Francophones des Langages Applicatifs*, pages 1–14, 2008.
4. Coq development team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006.
5. John Cowles, Ruben Gamboa, and Jeff Van Baalen. Using ACL2 Arrays to Formalize Matrix Algebra. In *ACL2 Workshop*, 2003.
6. Georges Gonthier and Assia Mahboubi. *A small scale reflection extension for the Coq system*. INRIA Technical report, `http://hal.inria.fr/inria-00258384`.
7. Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In *Theorem Proving in Higher-Order Logics*, volume 4732 of *LNCS*, pages 86–101, 2007.
8. John Harrison. HOL Light: A Tutorial Introduction. In *FMCAD*, pages 265–269, 1996.
9. John Harrison. A HOL Theory of Euclidian Space. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 114–129. Springer, 2005.
10. Gérard Huet and Amokrane Saïbi. Constructive category theory. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 239–275, Cambridge, MA, USA, 2000. MIT Press.
11. Florian Kammuller. Modular Structures as Dependent Types in Isabelle. In *Proc. TYPES Workshop*, pages 121–132, 1998.
12. Nicolas Magaud. Ring properties for square matrices. `http://coq.inria.fr/contribs-eng.html`.
13. Steven Obua. Proving Bounds for Real Linear Programs in Isabelle/HOL. In *Theorem Proving in Higher-Order Logics*, pages 227–244, 2005.
14. Ioana Paşca. A Formal Verification for Kantorovitch's Theorem. In *Journées Francophones des Langages Applicatifs*, pages 15–29, 2008.
15. Lawrence C. Paulson. Organizing Numerical Theories Using Axiomatic Type Classes. *J. Autom. Reason.*, 33(1):29–49, 2004.
16. Robert Pollack. Dependently Typed Records for Representing Mathematical Structure. In *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 462–479, London, UK, 2000. Springer-Verlag.
17. J. Stein. Documentation for the formalization of Linerar Agebra. `http://www.cs.ru.nl/~jasper/`.