A Formalization of Grassmann-Cayley Algebra in COQ and Its Application to Theorem Proving in Projective Geometry

Laurent Fuchs, Laurent Thery

A Formalization of Grassmann-Cayley Algebra in Coq and its Application to Theorem Proving in Projective Geometry*

Laurent Fuchs[1] Laurent Théry[2]

[1] XLIM-SIC UMR CNRS 6172 - Poitiers University, France,
Laurent.Fuchs@sic.univ-poitiers.fr
[2] INRIA Sophia Antipolis - Méditerranée, France,
Laurent.Thery@inria.fr

Abstract. This paper presents a formalization of Grassmann-Cayley algebra [6] that has been done in the Coq [2] proof assistant. The formalization is based on a data structure that represents elements of the algebra as complete binary trees. This allows to define the algebra products recursively. Using this formalization, published proofs of Pappus' and Desargues' theorem [7,1] are interactively derived. A method that automatically proves projective geometric theorems [11] is also translated successfully into the proposed formalization.

1 Introduction

A well-known application of Grassmann-Cayley algebra is automated theorem proving in projective geometry (see for example [1,4,10]). The usual method is to translate incidence statements of projective geometry into Grassmann-Cayley expressions. These expressions are then translated into bracket polynomials (i.e. the ring of projective invariants [17]). Finally, the bracket polynomial is factorised to get back an equivalent expression in Grassmann-Cayley algebra.

Our motivation in using a proof assistant such as Coq [2] is to capture in a single system all the various aspects of Grassmann-Cayley algebra: we want an abstract generic model on which we can not only reason but also perform both numerical and symbolical evaluations.

Hence, our formalization lets us not only formally check the manipulations of expressions within Grassmann-Cayley algebra but also compute with these very same expressions. De facto, it makes explicit the link between the abstract mathematical object and its applications. In the Coq proof assistant, proofs can be conducted interactively step by step or, using programmed tactics, the expressions can be reduced in a systematic manner. Note that, in our setting, most proofs are parametrized by the dimension of the algebra. So, our development is generic.

Once the formalization of the Grassmann-Cayley algebra is achieved, two kinds of proofs are considered. First some proofs are conducted interactively

* This work has been supported by the ANR Galapagos

following step-by-step what can be found in the literature, such as the proof of the Pappus' theorem in [7] or the proofs of Desargues' theorem in [1]. The second kind of proofs are conducted automatically following a method published in [11]. All the examples proposed in [11] have been tested successfully.

In future work, we also plan to connect our formalization with other approaches of incidence geometry, such as those based over ranks [12,13,14]. So, our work can be seen as a first step in the study of the formal correctness of automated proof methods in incidence geometry.

This paper is organized as follows. Section 2 introduces Grassmann-Cayley algebra and our choices for the formalization. Section 3 explains how the Grassmann-Cayley is formalized, how the algebra elements are represented and how the products are defined. Section 4 describes how the formalization can be use to prove theorems of incidence geometry, interactively and automatically.

2 Formal Grassmann-Cayley Algebra

Usually, in the literature, the products (join and meet) of the Grassmann-Cayley algebra are introduced by given equations defining their properties. So, they could have been defined in Coq using such an axiomatic approach. However, the main drawback of doing so is that we completely lose the computational aspect of this algebra. In particular, the axiomatic approach gives no hint of how the algebra could actually be implemented on a computer.

For this reason, we favor the definitional approach where the algebra operations are defined as recursive functions over the dimension of the algebra. First, we define a model, i.e. a data-structure that represents elements of the algebra. Then, on this model, we define the usual algebra operations (the join product, the meet product and the duality) and prove that they fulfill the axioms that are used to defined them in the literature. As this representation is quite unusual, we spend some time to detail our data-structure and the related operations.

2.1 The underlying vector space

The Grassmann-Cayley algebra G_n is defined by adding a second product, the meet product, to the Grassmann algebra (or exterior algebra) of a vector space of dimension n, V, over a field K [6,1] where the join product (or the exterior product) is defined.

In order to have a concrete representation of the vectors of V, we need to represent them as n-tuples of K^n. This imposes a basis for V, say the canonical basis $e^i_n = (\delta_{i,0}, \ldots, \delta_{i,i}, \ldots, \delta_{i,n-1})$ where $i = 0, \ldots, n-1$ and $\delta_{i,j}$ is the Kronecker symbol. Then V is seen as the set of n-tuples, K^n.

As we will see this choice also induces the definition of a basis for G_n and this leads to an important change of view in the presentation of the algebra compared to the usual coordinate-free presentation. The elements are represented via their coordinates.

However, in the Coq proof assistant, this does not force us to deal only with numerical computations. As all the axiomatic properties of the algebra operations are proved, we can also reason symbolically using the coordinate-free presentation. Hence, we obtain an abstract generic model on which both numerical and symbolical evaluations can be performed.

2.2 The join product

The first step is to define the join product denoted by $\vee$. It is an associative antisymmetric bilinear product and it can be defined axiomatically by:

$$\begin{array}{ll} a \vee a = 0 & \lambda a \vee b = \lambda(a \vee b) \\ b \vee a = -a \vee b & (a+b) \vee c = a \vee c + b \vee c \end{array} \tag{1}$$

where a and b are vectors of K^n.

The join product $a \vee b$ of two vectors a and b is non-zero if and only if a and b are linearly independent. If $a \vee b$ is non-zero, it is a grade 2 element of the algebra. More generally, if $\{a_1, \ldots, a_k\}$ are linearly independent vectors of K^n then $a_1 \vee \cdots \vee a_k$ is an element of grade k. Such elements, that are join product of vectors, are called *extensors* or *decomposable k-vectors*. Not all elements of grade k are extensors, they could be linear combination of extensors. In that case they are called *homogeneous vectors* or *k-vectors*. Elements that are linear combination of elements with different grades are the general elements of the algebra. They are called *multi-vectors*.

On the basis elements $\{e_n^0, \ldots, e_n^{n-1}\}$ of K^n, the join product has the following two behaviors:

$$\mathrm{e}_n^i \vee \mathrm{e}_n^i = 0 \qquad\qquad \mathrm{e}_n^i \vee \mathrm{e}_n^j = -\mathrm{e}_n^j \vee \mathrm{e}_n^i.$$

This gives the graded structure of G_n. The join product of k basis elements generates the subspace of grade k homogeneous elements. Considering G_3, this means that:

$\{1\}$ generates the elements of grade 0.
$\{\mathrm{e}_3^0, \mathrm{e}_3^1, \mathrm{e}_3^2\}$ generates the elements of grade 1.
$\{\mathrm{e}_3^0 \vee \mathrm{e}_3^1, \mathrm{e}_3^0 \vee \mathrm{e}_3^2, \mathrm{e}_3^1 \vee \mathrm{e}_3^2\}$ generates the elements of grade 2.
$\{\mathrm{e}_3^0 \vee \mathrm{e}_3^1 \vee \mathrm{e}_3^2\}$ generates the elements of grade 3.

Hence, G_n can be seen as a vector space of dimension 2^n. Our model is a representation of this vector space that allows a computational definition of the products of the Grassmann-Cayley algebra.

2.3 The meet product

Retrieving the bracket. Usual presentation of the Grassmann-Cayley algebra [6,1,18] defines a bracket over the vector space V. Given n vectors $a_1, \ldots, a_n$ the bracket $[a_1, \ldots, a_n]$ is a non-degenerate multilinear alternating n-form, taking its values into the field K.

The use of the canonical basis of the vector space K^n defines a bracket implicitly. The set of elements of grade n generated by $e_n^0 \vee \cdots \vee e_n^{n-1}$ is isomorphic to the set of elements of grade 0 via the linear map defined by $i(e_n^0 \vee \cdots \vee e_n^{n-1}) = 1$. This linear map defines a non-degenerate multilinear n-form over the vectors of K^n that is actually a determinant.

Hence, the choice of the canonical basis defines a bracket. This allows us to retrieve the usual definition of the Grassmann-Cayley algebra. This link is used in section 4.2 to introduce automated proof techniques into our formalization.

The Hodge star. Moreover, as $i(e_n^0 \vee \cdots \vee e_n^{n-1}) = [e_n^0, \ldots, e_n^{n-1}] = 1$, the canonical basis is said to be unimodular [1]. Then, the Hodge star defined as follows:

$$*(e_n^{\rho(0)} \vee_n \ldots \vee_n e_n^{\rho(i)}) = e_n^{\rho(i+1)} \vee_n \ldots \vee_n e_n^{\rho(n-1)}$$

where ρ is an even permutation, satisfies the following properties (see [1]):

(i) $*$ maps extensors of grade k to extensors of grade $n-k$,
(ii) $*(1) = e_n^0 \vee \cdots \vee e_n^{n-1}$ and $*(e_n^0 \vee \cdots \vee e_n^{n-1}) = 1$,
(iii) $*(*(A)) = (-1)^{k(n-k)} A$ if A is of grade k.

The Hodge star realizes the duality between the meet and the join products [1]. Hence, the following definition of the meet product, denoted $\wedge$, can be adopted:

$$*(A \vee B) = *(A) \wedge *(B) \quad \text{and} \quad *(A \wedge B) = *(A) \vee *(B).$$

Thus, in the algebra G_3, the meet product can be defined over the basis elements by the table

$\wedge$	1	e_3^0	e_3^1	e_3^2	$e_3^0 \vee e_3^1$	$e_3^0 \vee e_3^2$	$e_3^1 \vee e_3^2$	$e_3^0 \vee e_3^1 \vee e_3^2$
1	0	0	0	0	0	0	0	1
e_3^0	0	0	0	0	0	0	1	e_3^0
e_3^1	0	0	0	0	0	-1	0	e_3^1
e_3^2	0	0	0	0	1	0	0	e_3^2
$e_3^0 \vee e_3^1$	0	0	0	1	0	e_3^0	e_3^1	$e_3^0 \vee e_3^1$
$e_3^0 \vee e_3^2$	0	0	-1	0	$-e_3^0$	0	e_3^2	$e_3^0 \vee e_3^2$
$e_3^1 \vee e_3^2$	0	1	0	0	$-e_3^1$	$-e_3^2$	0	$e_3^1 \vee e_3^2$
$e_3^0 \vee e_3^1 \vee e_3^2$	1	e_3^0	e_3^1	e_3^1	$e_3^0 \vee e_3^1$	$e_3^0 \vee e_3^2$	$e_3^1 \vee e_3^2$	$e_3^0 \vee e_3^1 \vee e_3^2$

3 Data-structures

The programming language of Coq proof assistant [3] is a functional language with dependent types. It is then particularly suitable for the development of abstract algebra. In order to have a generic formalization, our development is parametrized by an abstract field K and its usual operations:

```
Structure FieldParams := {
       K : Set              ;
       0 : K                ;
       1 : K                ;
  _ ≟ _ : K → K → bool      ;
    - _ : K → K             ;
   _ + _ : K → K → K        ;
   _ * _ : K → K → K        ;
    _ ⁻¹ : K → K
}
```

Note that even if every type in Coq is equipped with a propositional equality, i.e. for two elements x and y in `K` the proposition $x = y$ expresses that they are equal with respect to Leibnitz equality, we have an explicit equality test $x \stackrel{?}{=} y$ that lets us decide on this equality. This capability is crucial when defining algorithms over elements of K. Along with this parametric definition of K, there is an associated set of axioms that gives the usual basic properties of the operations (associativity, commutativity, distributivity and neutral elements).

From now on, all our definitions are taking this field `K` and another parameter n for the dimension as parameters. They follow the same pattern: they are defined recursively on the dimension n. For a data-structure D, this means that its version D_{n+1} for the $n+1$ dimension is going to be expressed in term of D_n. In this work, only the primitive pairing construct of Coq is used: if a_1 is of type T_1 and a_2 of type T_2, (a_1, a_2) is of type $T_1 \times T_2$.

3.1 Representing the vector space K^n

As a first example, here is how the vectors of K^n are defined for $n \neq 0$:

```
Definition K_n := if n = 1 then K else K x K_{n-1}.
```

Compare to traditional programming where vectors would be represented as arrays, here we use recursion and pairing to mimic this data-structure. The type[3] K_1 is equivalent to K, K_2 to $K \times K$ and K_3 to $K \times (K \times K)$ and an element of K_3 is represented by $(x_1, (x_2, x_3))$.

Operations on this data-structure are also defined recursively. For example, addition of two vectors of dimension n is defined recursively as follows:

```
Definition x +_n y := if n = 1 then x + y else
   let (x1,x2) := x and (y1,y2) := y in
   (x1 + y1 , x2 +_{n-1} y2).
```

[3] The exponent n is changed into an index for notational purpose.

If the parameter n is one, the two elements belong to K so we can add them using the addition on K, otherwise each element can be decomposed into an element of K and an element of one dimension less and the resulting pair can be composed by adding the elements of K on the left and using a recursive call on the right. To end the vector space structure, scalar multiplication can be defined in a similar way:

```
Definition k ·_n y := if n = 1 then k * x else
   let (x1,x2) := x in (k * x1 , k ·_{n-1} x2).
```

3.2 Representing the algebra G_n

Representing elements of G_n, the Grassmann-Cayley of dimension n, follows exactly the same schema. This time, instead of a linear data-structure, binary trees are used:

```
Definition G_n := if n = 0 then K else G_{n-1} x G_{n-1}.
```

The type G_0 is equivalent to K, G_1 to $K \times K$, G_2 to $(K \times K) \times (K \times K)$. Elements of G_n are binary trees of height n. They have 2^n leaves. This corresponds to the fact that G_n is a vector space of dimension 2^n.

The sum and the scalar multiplication for the vector space structure are defined recursively over the dimension as follows:

```
Definition x +_n y := if n = 0 then x + y else
   let (x1,x2) := x and (y1,y2) := y in
   (x1 +_{n-1} y1 , x2 +_{n-1} y2).

Definition k ·_n y := if n = 0 then k * x else
   let (x1,x2) := x in (k ·_{n-1} x1 , k ·_{n-1} x2).
```

The equality test is defined in the same way:

```
Definition x ?=_n y := if n = 0 then x ?= y
   let (x1,x2) := x and (y1,y2) := y in
   (x1 ?=_{n-1} y1) && (x2 ?=_{n-1} y2).
```

In this definition the operator `&&` is a special notation used in this paper for the logical *and* to avoid confusion with the meet product.

Figure 1 explains how the basis components of G_n are mapped to the binary structure. The leaves contain the coefficients. For example, the grade 2 element of G_3,

$$2.(e_3^0 \vee e_3^1) + 3.(e_3^1 \vee e_3^2)$$

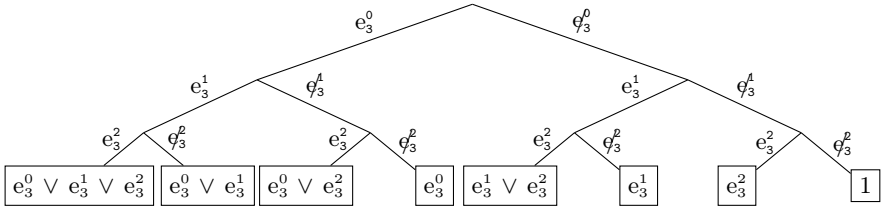

Fig. 1. Mapping of the multi-vector coefficients to the leaves of the binary tree.

is represented as $(((0,2),(0,0)),((3,0),(0,0)))$ and the multi-vector of G_3,

$$2.(e_3^0 \vee e_3^2) + 3.e_3^1 + 4.1$$

is represented as $(((0,0),(2,0)),((0,3),(0,4)))$. Here the sign sum indicates that an element of G_n is a linear combination of the basis components.

For a tree of height n, at the level i, a move toward the left child inserts the basis element e_n^i into the join product, while a move toward the right child insures that this element is not present. Then, on a path from the root of the tree to a leaf a move toward the left child increases the grade by one, while a move toward the right child leaves it unchanged. Hence, the left-most leaf of a tree of height n contains the grade n coefficient of an element of G_n while the right-most leaf contains the coefficient of the grade 0 part.

This binary tree structure also allows to increase the dimension of an element x of G_n by injecting it to G_{n+1}. This is done with the function inj_{G_n} by simply pairing the binary tree of height n with all its leaves containing 0, denoted 0_n, and x:

```
Definition inj_{G_n} x := (0_n , x).
```

This operation does not change the grade of x, but it shifts the basis components. The basis component $e_n^i \vee \cdots \vee e_n^{i+k}$ is mapped to the basis component $e_{n+1}^{i+1} \vee \cdots \vee e_{n+1}^{i+k+1}$.

From the mapping of the multi-vector coefficients and the injection function, the pairing of two elements x and y of G_n can be interpreted in terms of an element of G_{n+1}. The pair (x, y) represents the element

$$e_{n+1}^0 \vee \text{inj}_{G_n}\, x +_{n+1} \text{inj}_{G_n}\, y. \tag{2}$$

This means that pairing two elements x and y inserts the basis component e_{n+1}^0 into the shifted basis component of x. This puts all the coefficients of x into the left part of the tree.

Now, the field K is injected into the binary tree structure representing G_n with the following functions:

```
Definition inj_{n,K} k := if n = 0 then k else (inj_{n-1,K} 0,inj_{n-1,K} k).

Definition 0_n := inj_{n,K} 0.

Definition 1_n := inj_{n,K} 1.
```

Hence, the tree representing zero has all its leaves set to 0, while the tree representing one has only its right-most leaf set to 1.

Using the injection of K into G_n, we can define the injection of the elements of K_n into G_n:

```
Definition inj_{K_n} x := if n = 0 then 0 else
   if n = 1 then (x , 0) else
   let (x_1,x_2) := x in (inj_{n-1,K} x_1 , inj_{K_{n-1}} x_2).
```

Note that, as K_0 is not defined, a special case is introduced for $n = 0$ sending any element to zero. Let us take a concrete example to explain how this injection works. An element of K_3 is represented by a triplet $(x_1, (x_2, x_3))$. The element $(((0,0),(0,x_1)),((0,x_2),(x_3,0)))$ of G_3 is its image by the injection. For an element of K_n, the coordinates x_i are the coefficients of the basis elements e_n^i

We can also directly exhibit a base $\{e_n^0, e_n^1, \ldots, e_n^{n-1}\}$ for the vectors of G_n, i.e. the image by the injection inj_{K_n} of the base of K_n induced by the coordinates. Again, this is defined recursively:

```
Definition e^i_n := if n = 0 then 1_n else
    if i = 0 then (1_{n-1},0_{n-1}) else (0_{n-1}, e^{i-1}_{n-1}).
```

If we go back to the relation between K_3 and G_3, the base of K_3 induced by the coordinates is $\{(1,0,0),(0,1,0),(0,0,1)\}$. Then, we have

$\mathrm{e}_3^0 = (((0,0),(0,1)),((0,0),(0,0)))$ which corresponds to $(1,0,0)$,
$\mathrm{e}_3^1 = (((0,0),(0,0)),((0,1),(0,0)))$ which corresponds to $(0,1,0)$,
$\mathrm{e}_3^2 = (((0,0),(0,0)),((0,0),(1,0)))$ which corresponds to $(0,0,1)$.

When they are injected respectively into G_n and G_{n+1}, the basis elements of K_n and K_{n+1} are related by $\mathrm{e}_{n+1}^{i+1} = \mathrm{inj}_{G_n}\ \mathrm{e}_n^i$. Hence, in terms of trees, the basis element e_n^i is the right child of the basis element e_{n+1}^{i+1}. In the previous example, we can observe the left zero tree in the representation of e_3^2 and e_3^1 indicating that $e_3^2 = \mathrm{inj}_{G_2}\, e_2^1$ and $e_3^1 = \mathrm{inj}_{G_2}\, e_2^0$. This is coherent with interpretation of the pairing of two elements of G_n (see the relation (2)).

At the moment, from the point of view of the properties that can be formally proved, only the usual properties of vector space for K_n and G_n and the properties of morphism of the different injections can be derived.

3.3 Join product

The next step is to define the join product as a binary tree operation. To explain the definition, we use equation (2) and the mandatory properties expressed by the axioms (1) in section 2.2.

The idea is to define the join product recursively over the dimension. To do so, we decompose the product $x \vee_n y$ in terms of pairing and using the relation (2), we obtain:

$$\begin{aligned} x \vee_n y &= (x_1, x_2) \vee_n (y_1, y_2) \\ &= (e_n^0 \vee_n \mathrm{inj}_{G_{n-1}} x_1 +_n \mathrm{inj}_{G_{n-1}} x_2) \vee_n (e_n^0 \vee_n \mathrm{inj}_{G_{n-1}} y_1 +_n \mathrm{inj}_{G_{n-1}} y_2) \end{aligned}$$

Then, using the axioms (1), we obtain:

$$\begin{aligned} (x_1, x_2) \vee_n (y_1, y_2) \;=\; & e_n^0 \vee_n \mathrm{inj}_{G_{n-1}} x_1 \vee_n \mathrm{inj}_{G_{n-1}} y_2 \\ & +_n \mathrm{inj}_{G_{n-1}} x_2 \vee_n e_n^0 \vee_n \mathrm{inj}_{G_{n-1}} y_1 \\ & +_n \mathrm{inj}_{G_{n-1}} x_2 \vee_n \mathrm{inj}_{G_{n-1}} y_2. \end{aligned} \tag{3}$$

In the second term in the sum of the right part of this latter expression, the factor e_n^0 needs to be commuted with $\mathrm{inj}_{G_{n-1}} x_2$ in order to be able to factorize the expression with e_n^0 and to get an expression that corresponds to a pairing.

However, the join product is anti commutative and we must pay attention to sign changes into the factors. For example, if x is an homogeneous element of grade k, we have $\mathrm{e}_n^i \vee x = (-1)^k \cdot x \vee \mathrm{e}_n^i$. We want to generalize this property and have a conjugate function, noted $\overline{x}$, such that $\mathrm{e}_n^i \vee x = \overline{x} \vee \mathrm{e}_n^i$ for all x in G_n. Here is the definition of such a function:

```
Definition x̄ⁿ := if n = 0 then x else
   let (x1,x2) := x in (-x̄1ⁿ⁻¹ , x̄2ⁿ⁻¹).
```

Now expression (3) can be rewritten as:

$$\begin{aligned} (x_1, x_2) \vee_n (y_1, y_2) \;=\; & e_n^0 \vee_n (\mathrm{inj}_{G_{n-1}} x_1 \vee_n \mathrm{inj}_{G_{n-1}} y_2 \\ & \qquad +_n \overline{\mathrm{inj}_{G_{n-1}} x_2}^{\,n} \vee_n \mathrm{inj}_{G_{n-1}} y_1) \\ & +_n \mathrm{inj}_{G_{n-1}} x_2 \vee_n \mathrm{inj}_{G_{n-1}} y_2. \end{aligned} \tag{4}$$

Using the definition of the injection inj_G, we get:

$$\begin{aligned} (x_1, x_2) \vee_n (y_1, y_2) \;=\; & e_n^0 \vee_n (\mathrm{inj}_{G_{n-1}}(x_1 \vee_{n-1} y_2 +_{n-1} \overline{x_2}^{\,n-1} \vee_{n-1} y_1)) \\ & +_n \mathrm{inj}_{G_{n-1}}(x_2 \vee_{n-1} y_2). \end{aligned} \tag{5}$$

This leads to the following recursive definition of the join product:

```
Definition x ∨ₙ y := if n = 0 then x * y else
    let (x1, x2) := x and (y1, y2) := y in
    (x1 ∨ₙ₋₁ y2 +ₙ₋₁ x̄2ⁿ⁻¹ ∨ₙ₋₁ y1, x2 ∨ₙ₋₁ y2)
```

From this definition, we have proved formally that this join product verifies its basic properties (associativity, bilinearity and anti commutativity) defined by the axioms (1).

3.4 Meet Product

In order to define the meet product, we follow exactly the same path than for the join product. We generalize the fact that, for an homogeneous element x of grade k of G_n, we have $\mathrm{e}_n^i \wedge x = (-1)^{(n-k)} \cdot x \wedge \mathrm{e}_n^i$ and define a dual version of the conjugate function, noted $\overline{x}^{\,d}$, such that $\mathrm{e}_n^i \wedge x = \overline{x}^{\,d} \wedge \mathrm{e}_n^i$.

```
Definition x̄^dn := if n = 0 then x else
   let (x1,x2) := x in (x̄1^dn-1 , -x̄2^dn-1).
```

Again, with this auxiliary function, the meet product can be defined recursively as follows:

```
Definition x ∧n y := if n = 0 then x * y else
    let (x1, x2) := x and (y1, y2) := y in
    (x2 ∧n-1 y2, x1 ∧n-1 y2 +n-1 x2 ∧n-1 ȳ1^dn-1)
```

Note that our recursive approach avoids the use of bracket algebra to define the meet product so that our formalization works internally and independently from the bracket algebra framework. As for the join product, it is quite direct to derive the basic properties of the meet product formally from its definition.

3.5 Duality

The Hodge star operator presented in section 2.3 is also defined recursively over the dimension as follows:

```
Definition *n(x) := if n = 0 then x else
    let (x1, x2) = x in (*n-1(x̄2^n-1), *n-1(x)).
```

Note that in our representation, upto sign flips, the Hodge star just reverses the leaves of the binary tree. For example in G_3, the dual of the element

$$(((x_1, x_2), (x_3, x_4)), ((x_5, x_6), (x_7, x_8)))$$

is the reverse element with two sign flips

$$(((x_8, x_7), (-x_6, x_5)), ((x_4, -x_3), (x_2, x_1))).$$

Because of these sign flips, the Hodge star is not an involution. For homogeneous elements, the following theorem (corresponding to property (iii) in section 2.3) is proved into our formalization using the defined Hodge star:

```
Lemma dual_invo: ∀n k v, if homn^k v then *n(*n(v)) = (-1)^k(n-k) ·n v.
```

where $\texttt{hom}_n^k$ tests if an element is homogeneous and is defined as follows:

```
Definition hom_n^k x := if n = 0 then (k = 0 || x ?= 0) else
   let (x1,x2) := x in
   (if k = 0 then x1 ?=_{n-1} 0_{n-1} else hom_{n-1}^{k-1} x1) && hom_{n-1}^k x2.
```

As previously, the notation $\|$ is a special notation for the logical *or* to avoid confusion with the join product.

Due to our choice of the underlying vector space basis, the Hodge star implements the duality between the join product and the meet product (see section 2.3 and reference [1]). Then the following theorem are proved within our Coq formalization:

```
Lemma  dual_prod: ∀n v1 v2, *_n(v1 ∨_n v2) = *_n(v1) ∧_n *_n(v2).
Lemma dual_dprod: ∀n v1 v2, *_n(v1 ∧_n v2) = *_n(v1) ∨_n *_n(v2).
```

This proves that the join product and the meet product are correctly defined.

At this point, Grassmann-Cayley algebra could already be considered as formalized in the Coq proof assistant.

4 Theorem Proving in Projective Geometry

In this section, we first show how we can use our formalization of Grassmann-Cayley algebra to model the geometry of incidence. Then, in a second step, we show how proofs in this setting can be fully automatized within Coq.

4.1 Modeling the Geometry of Incidence

Now that we have Grassmann-Cayley algebra in Coq, we can use it to represent theorems in projective geometry. All this is standard and can be found by example in [18] or in [15] chapter 3. We just explain how this has been instantiated to our formalization. We work over an arbitrary field K and restrict ourselves to G_3. We take a conservative approach and consider only non-degenerated configurations for constructed points. In this setting, points are vectors, so in our case we are going to use our injection from K_3 to G_3:

```
Definition point K := K_3.
```

To define the fact that a point p_1 is the intersection of the line composed of p_2 and p_3 and the line composed of p_4 and p_5, we simply implement it by saying that using the join to create the line and the meet to perform the intersection:

```
Definition p1 is the intersection of [p2,p3] and [p4,p5] :=
   inj_{3,K} p1 = (inj_{K_3} p2 ∨_3 inj_{K_3} p3) ∧_3 (inj_{K_3} p4 ∨_3 inj_{K_3} p5).
```

Note that the equality imposes the meet product to be a point, so the lines to be defined and intersecting.

To define the fact that a point p_1 is on the line composed of p_2 and p_3, we simply implement it by saying that the line is well-defined, i.e. the join product of p_2 and p_3 is not zero, and the joint product of the three points is zero:

```
Definition p1 is free on [p2,p3] :=
       (inj_K3 p2) ∨3 (inj_K3 p3) ≠ 03
   and
       (inj_K3 p1) ∨3 (inj_K3 p2) ∨3 (inj_K3 p3) = 03.
```

Finally, we consider the collinearity of three points and the concurrency of three lines:

```
Definition {p1,p2,p3} are collinear :=
   (inj_K3 p1) ∨3 (inj_K3 p2) ∨3 (inj_K3 p3) = 03.

Definition {[p1,p2],[p3,p4],[p4,p5]} are concurrent :=
   ((inj_K3 p1) ∨3 (inj_K3 p1)) ∧3
   ((inj_K3 p3) ∨3 (inj_K3 p4)) ∧3
   ((inj_3,K p4) ∨3 (inj_K3 p5))    = 03.
```

With these definitions, we can start stating some classic theorems of geometry of incidence. First, let us consider Pappus' theorem:

```
Theorem Pappus: ∀ a b c a' b' c' p q r: point K,
if p is the intersection of [a,b'] and [a',b] and
   q is the intersection of [b,c'] and [b',c] and
   r is the intersection of [c,a'] and [c',a] and
   {a,b,c} are collinear and {a',b',c'} are collinear
then {p,q,r} are collinear.
```

Introducing the universal quantification and eliminating the points p, q and r, we are left with proving that[4]:

```
if a ∨ b ∨ c = 0 and a' ∨ b' ∨ c' = 0 then
(a ∨ b' ∧ a' ∨ b) ∨ (b ∨ c' ∧ b' ∨ c) ∨ (b ∨ c' ∧ b' ∨ c) = 0
```

Remaining inside the algebra and applying the basic properties it is possible to prove this statement interactively in Coq. For this, we have followed of the proof given in [7]. This requires 10 interactions where the prover is guided in order to apply the symbolic manipulations that leads to the proof.

A more involved proof is Desargues' theorem. It can be stated as:

[4] We voluntarily omit the injections and the indices to make the expression more legible.

```
Theorem Desargues: ∀ a b c a' b' c': point K,
if p is the intersection of [a,b] and [a',b'] and
  q is the intersection of [a,c] and [a',c'] and
  r is the intersection of [b,c] and [b',c'] and
then
  {p,q,r} are collinear
iff
  {a,b,c} are collinear or {a,b,c} are collinear or
  {[a,a'],[b,b'],[c,c']} are concurrent.
```

Again, introducing the universal quantification and eliminating the points p, q and r, we are left with proving that:

```
  (a ∨ b ∧ a' ∨ b') ∨ (a ∨ c ∧ a' ∨ c') ∨ (b ∨ c ∧ b' ∨ c') = 0
iff
a ∨ b ∨ c = 0 or a' ∨ b' ∨ c' = 0 or a ∨ a' ∧ b ∨ b' ∧ c ∨ c' = 0
```

In order to prove this interactively, this time we have followed the paper proof given in [1]. The proof is more intricate and has required 60 interactions with the prover.

4.2 Automating Proofs

Proving the last two theorems is very satisfying because it shows that our algebra can be manipulated symbolically within Coq but clearly we are at the limit of what is bearable for a user to prove interactively. So, the next step is to automate the proof of such theorems. For this, we are going to introduce bracket algebra and follow the path of [11].

Bracket algebra and its relation with Grassmann-Cayley is a well-known topic [6,1]. Here, we are just going to explain how it has been introduced in our setting. For the moment, this has only been implemented for G_3 but we believe that this could be easy generalised to G_n for an arbitrarily n. In the following, in order to increase legibility we will systematically omit the indices and the injections, so for example ($\text{inj}_{K_3}\ p_1 \vee_3 \text{inj}_{K_3}\ p_2$) will be noted ($p_1 \vee p_2$) only. A bracket is a function that takes three points and returns an element of our field K. Its definition is the following:

```
Definition [p1,p2,p3] := dC (p1 ∨ p2 ∨ p3).
```

where `dC` stands for the dual of the constant component, i.e the left-most leaf of the tree-structure given in Figure 1 of page 7. The usual relations between the bracket, the join product and the meet product in G_3 are derived.

```
Lemma bracket_defE: ∀p1 p2 p3,
 p1 ∨ p2 ∨ p3 = [p1,p2,p3] · e0 ∨ e1 ∨ e2.
Lemma bracket_defI: ∀p1 p2 p3,p1 ∧ (p2 ∨ p3) = [p1,p2,p3] · 1.
```

We have also formally proved that it behaves as a determinant:

```
Lemma bracket0l: ∀p1 p2, [p1,p1,p2] = 0
Lemma bracket_swapl: ∀p1 p2 p3,[p1,p2,p3] = − [p2,p1,p3].
Lemma bracket_swapr: ∀p1 p2 p3,[p1,p2,p3] = − [p1,p3,p2].
Lemma bracket_free: ∀α β p1 p2 p3 p4 p5,
 if p1 = α · p4 + β · p5
 then [p1,p2,p3] = α ∗ [p4,p2,p3] + β ∗ [p5,p2,p3].
```

In order to automate as described in [11], we are going to restrict ourselves to a specific skeleton of proofs. The goals we are going to be able to prove automatically have the following shape:

$$\forall p_1\, p_2\, \ldots\, p_m,\ \textit{if}\ H_1\ \textit{and}\ \ldots\ H_n \quad \textit{then} \quad \{p_i, p_j, p_k\}\ \texttt{are collinear}$$

where the H_is are either the construction of a free point on a line

$$p_j\ \texttt{is free on}\ [p_r, p_s]$$

or the construction of an intersection

$$p_j\ \texttt{is the intersection of}\ [p_r, p_s]\ \texttt{and}\ [p_t, p_u].$$

How does the automatic procedure proceed? As the conclusion is a collinearity property, it can be turned into an equality to zero of a bracket expression by the following lemma that is a direct consequence of the lemma `bracket_defE`:

```
Lemma collinear_bracket: ∀p1 p2,
 {p1,p2,p3} are collinear iff [p1,p2,p3] = 0
```

Then, the constructed points are progressively eliminated from the assumptions to obtain a bracket expression. Two lemmas are used corresponding to each construction. In the first case, the assumption is the construction of a free point on a line. The following lemma can be proved thanks to the conservative approach we observed:

```
Lemma online_def: ∀p1 p2 p3,
 if p1 is free on [p2,p3] then ∃ α β,p1 = α · p2 + β · p3
```

Coupled with the lemma `bracket_free`, this lets us remove the free point from all bracket expressions. In the second case, the assumption is the construction of an intersection then the second rule[5] given in [11] is used to remove the point:

```
Lemma bracket_expand: ∀p1 p2 p3 p4 p5 p6 p7,
 if p1 is the intersection of [p4,p5] and [p6,p7] then
 [p1, p2, p3] = −[p4, p2, p3] ∗ [p5,p6,p7] + [p5,p2,p3] ∗ [p4,p6,p7].
```

Once all the eliminations of constructed points have been performed, we get an expression that contains sums and products of bracket of initial points and the αs and the βs introduced by the eliminations of the free points. So for the theorem to be true generically, this expression must be equal to zero modulo Plücker relations (see [11]). In order to simplify the obtained expression a contraction rule is used in [11]). In our setting it is stated as:

```
Lemma contraction_v0: ∀p1 p2 p3 p4 p5,
[p1,p2,p3] ∗ [p1,p4,p5] − [p1,p2,p5] ∗ [p1,p4,p3] = [p1,p2,p4] ∗ [p1,p3,p5].
```

Surprisingly applying this rule unrestrictively as a rewrite rule from left to right as described in [11] is very effective. However, it is not sufficient in our setting to prove all the given examples. To fix this problem, we implement a normalisation method that is very expensive but is known to be complete. This captures the remaining examples. The method is based on an implicit ordering of the initial points $p_1 < p_2 < \cdots < p_i$. Applying some permutation, brackets can always be ordered with respect to this order: $[p_i,p_j,p_k]$ with $p_i < p_j < p_k$.

The order can be lifted to brackets $[p_i,p_j,p_k] \leq [p_{i'},p_{j'},p_{k'}]$ if $p_i \leq p_{i'}$ and $p_j \leq p_{j'}$ and $p_k \leq p_{k'}$. The normalisation proceeds in trying to order the product of brackets from the smallest to the largest. For this, we consider the product of two brackets $[p_i,p_j,p_k] * [p_{i'},p_{j'},p_{k'}]$. Without loss of generality, we can suppose that $p_i \leq p_{i'}$. There are only two situations where this product is not ordered:

1. $p_i < p_{i'}$ and $p_{j'} < p_j$
2. $p_i \leq p_{i'}$ and $p_j < p_{j'}$ and $p_{k'} < p_k$ (or equivalently $p_i < p_{i'}$ and $p_j \leq p_{j'}$ and $p_{k'} < p_k$)

The first rewrite rule takes care of the first case and assures that the resulting expression has every first two elements of brackets in a product properly ordered.

```
Lemma split3b_v1: ∀pi pi' pj pj' pk pk',
 [pi,pj,pk] ∗ [pi',pj',pk'] =
   [pi,pi',pj'] ∗ [pj,pk,pk'] − [pi,pi',pk'] ∗ [pj,pk,pj']+
   [pi,pj',pk'] ∗ [pj,pk,pi'].
```

[5] This rule corresponds to the elimination of the area method [9].

The second rewrite rule takes care of the second case and insures that the resulting products of brackets are all properly ordered.

```
Lemma split3b_v2: ∀p_i p_i' p_j p_j' p_k p_k',
 [p_i,p_j,p_k] * [p_i',p_j',p_k'] =
   [p_i,p_j,p_i'] * [p_k,p_j',p_k'] − [p_i,p_j,p_j'] * [p_k,p_i',p_k']+
   [p_i,p_j,p_k'] * [p_k,p_i',p_i'].
```

5 Conclusion

We have described how our formalization of Grassmann-Cayley algebra has been achieved. It is a generic one: it is parametrized both by the underlying field K and by the dimension n. A snapshot of the formalization with a complete zipped archive is available at

`http://www-sop.inria.fr/marelle/GeometricAlgebra`.

Recursive definitions have played a central role in this formalization. Elements of the algebra are represented as binary trees. With this representation, operations like the meet, the join and the duality can be described as recursive functions in a very direct way. The nice thing about defining these operations in a proof assistant like Coq is that not only can we compute with them like in any programming language but also we can reason about them. This lets us derive all the standard properties of Cayley-Grassmann operations. Our implementation is then verified: we have a certified computational model of Grassmann-Cayley algebra.

One of the most satisfying part of this formalization is, without a doubt, the instantiation that has been done in order to prove Pappus' and Desargues' theorem as proposed in [7] and [1]. We have been capable of justifying formally every step of the paper proofs. Moreover, the method that automatically proves projective geometric theorems proposed in [11] has also be translated successfully into our formalization. An efficient Coq tactic has been developed. This makes us very confident in the potential of our formalization.

In addition to the formalization of the Grassmann-Cayley algebra basics properties, we have also considered other useful operations such as contraction $\langle \phi, v \rangle$ of a linear form ϕ on a vector v and factorisation have also been formalized.

As already mentioned in the introduction, we are very interested in studying the links with other formalized approaches of incidence geometry such as those based over ranks [14,13].

Finally, we also plan to develop our formalization to capture the powerful framework of the geometric algebra [8,5]

References

1. Barnabei, M., Brini, A., Rota, G.C.: On the Exterior Calculus of Invariant Theory. Journal of Algebra 96, 120–160 (1985)

2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
3. Coq development team: The Coq Proof Assistant Reference Manual, Version 8.2. LogiCal Project (2008), http://coq.inria.fr
4. Crapo, H., Richter-Gebert, J.: Automatic proving of geometric theorems. In: White [16], pp. 167–196
5. Dorst, L., Fontijne, D., Mann, S.: Geometric Algebra for Computer Science: An Object Oriented Approach to Geometry. Morgan Kauffmann Publishers (2007)
6. Doubilet, P., Rota, G.C., Stein, J.: On the foundations of combinatorial theory. IX. Combinatorial methods in invariant theory. Studies in Applied Mathematics 53, 185–216 (1974)
7. Hawrylycz, M.: A geometric identity for Pappus' theorem. Proceedings of the National Academy of Sciences U.S.A. 91(8), 2909 (1994)
8. Hestenes, D., Sobczyk, G.: Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics, Fundamental Theories of Physics, vol. 5. Kluwer Academic Publishers (1984)
9. Janicic, P., Narboux, J., Quaresma, P.: The Area Method : a Recapitulation. Journal of Automated Reasoning (2010), published online
10. Li, H.: Algebraic Representation, Elimination and Expansion in Automated Geometric Theorem Proving. In: ADG'02. LNAI, vol. 2930, pp. 106–123 (2002)
11. Li, H., Wu, Y.: Automated short proof generation for projective geometric theorems with Cayley and bracket algebras: I. Incidence geometry. Journal of Symbolic Computation 36(5), 717–762 (2003)
12. Magaud, N., Narboux, J., Schreck, P.: Formalizing Projective Plane Geometry in Coq. In: Proceedings of ADG'2008. pp. 1–20 (Sept 2008)
13. Magaud, N., Narboux, J., Schreck, P.: Formalizing Desargues' theorem in Coq using ranks. In: Proceedings of the ACM Symposium on Applied Computing SAC 2009. ACM, ACM Press (March 2009), http://lsiit.u-strasbg.fr/Publications/2009/MNS09
14. Michelucci, D., Schreck, P.: Incidence constraints: A combinatorial approach. International Journal of Computational Geometry & Applications 16(5-6), 443–460 (2006)
15. Sturmfels, B.: Algorithms in Invariant Theory. Springer, New York (1993)
16. White, N.L. (ed.): Invariants Methods in Discrete and Computational Geometry. Kluwer, Dordrecht (1995)
17. White, N.L.: A tutorial on Grassmann-Cayley algebra. In: Invariants Methods in Discrete and Computational Geometry [16], pp. 93–106
18. White, N.L.: Geometric applications of the Grassmann-Cayley algebra. In: Handbook of discrete and computational geometry, pp. 881–892. CRC Press, Inc., Boca Raton, FL, USA (1997)