Report on a SSReflect Week*

Y. Bertot, L. Rideau and ForMath La Rioja node

21-24th June 2010

Abstract

From 21th to 24th June, Y. Bertot and L. Rideau visited Logroño to give a course on SSReflect, in the frame of the ForMath European project. The activities in the week were organized into two parts: the course itself (lectures plus exercises) and joint work in a problem posed by the La Rioja team (namely, the formalization of finite simplicial complexes). In this short document, these activities are reported.

1 The course: Introduction to SSReflect

Lectures:

- Introduction to Ssreflect
  - The stack model. Basic commands.
  - The tactics. The rewrite tactics. Tools.
  - Small Scale Reflection (bool and nat).
- Basic Libraries.
  - ssrnat, ssrbool, ssrfun.
  - eqtype, fintype , seq.
- Canonical Structures.
- Hierarchy of algebraic structures.
- Big Operators.
- Matrices.

Exercices (see the Appendix):

- Some propositional tautologies

*Partially supported by by European Commission FP7, STREP project ForMath.

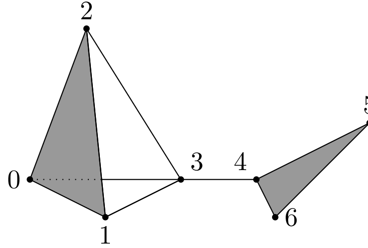

Figure 1: Butterfly Simplicial Complex

- Proofs about `forall` `exists`, `not`
- Views, reflection
- naturals numbers, lists,
- Proof that $\sqrt{2}$ is not a rational.

2 Formalization of finite simplicial complexes

2.1 The mathematics

The notion of simplicial complex is the most elementary method to settle a connection between common "general" topology and homological algebra. The notion of topological space is too "abstract" in order to perform computations. A triangulation, by means of simplicial complexes, can be provided for "sensible" spaces, so every topological space can be considered as a simplicial complex, making the computations easier.

Let us start with some basic terminology. Let V be a set, called the *vertex set*. An *(abstract) simplex* over V is any finite subset of V. Given a simplex α over V, a subset of α will be called a *face* of α (to stress the *geometrical* aspect of simplexes).

Definition 1 An *(abstract) simplicial complex* over V is a set of simplexes $\mathcal{C}$ over V such that it is closed by taking faces (subsets); that is to say, if $\alpha \in \mathcal{C}$ all the faces of α are in $\mathcal{C}$, too.

Example 1 Let us consider $V = \mathbb{N}$.

The small simplicial complex drawn in Figure 1 is mathematically defined as the object:
$C = \{\emptyset, \{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\},$
$\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}, \{4, 5\}, \{4, 6\}, \{5, 6\},$
$\{0, 1, 2\}, \{4, 5, 6\}\}$

facet	subsets
$\{1,3\}$	$\{\emptyset, \{1\}, \{3\}, \{1,3\}\}$
$\{3,4\}$	$\{\emptyset, \{3\}, \{4\}, \{3,4\}\}$
$\{0,3\}$	$\{\emptyset, \{0\}, \{3\}, \{0,3\}\}$
$\{2,3\}$	$\{\emptyset, \{2\}, \{3\}, \{2,3\}\}$
$\{0,1,2\}$	$\{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{0,2\}, \{1,2\}, \{0,1,2\}\}$
$\{4,5,6\}$	$\{\emptyset, \{4\}, \{5\}, \{6\}, \{4,5\}, \{5,6\}, \{4,6\}, \{4,5,6\}\}$

Table 1: Faces of the facets

Definition 2 A *facet* of a simplicial complex $\mathcal{C}$ is a maximal simplex with respect to the subset order $\subseteq$ among the simplexes of $\mathcal{C}$.

Example 2 The facets of the small simplicial complex depicted in Figure 1 are: $\{\{1,3\}, \{3,4\}, \{0,3\}, \{2,3\}, \{0,1,2\}, \{4,5,6\}\}$

Let us note that a *finite* simplicial complex can be generated from its facets taking their faces. Observe also that the notion of facet can be applied to any set of simplexes, and not only to simplicial complexes.

Example 3 Let us show the way of generating the simplicial complex depicted in Figure 1 from its facets. Table 1 shows the faces of facets of Figure 1. If we gather all the faces and remove the duplicate elements, the desired simplicial complex is obtained.

Then, the following algorithm can be defined.

Algorithm 1
Input: a set of simplexes $\mathcal{S}$
Output: the associated simplicial complex from $\mathcal{S}$

Exercise 1 Prove the correctness of algorithm 1.

Let us call `create_sc` to the function defined by algorithm 1 and `facets` to the function computing the facets of a set of simplexes. Then, other related exercises are:

- Given a finite set of simplexes `S`, `create_sc(S) = create_sc(facets(S))`.
- Given a finite set of simplexes `S`, `facets(create_sc(S))= facets(S)`.
- Given a finite simplicial complex `C`, `create_sc(facets(C)) = C`.

An important variant of a simplicial complex is obtained simply by demanding that the vertex set V is endowed with a partial order. Then, the corresponding simplicial complexes are called *ordered (abstract) simplicial complexes.* They are important because, if an order is known on the vertexes, there exists a canonical way for obtaining a *simplicial set* from a simplicial complex. Simplicial sets

are other combinatorial presentation of topological spaces. The Kenzo system works with simplicial sets.

Another consequence of putting an order on vertexes is that simplexes can be represented in a unique way as *ordered lists*. This will be employed in the next subsection.

2.2 ACL2 formalization

The formalization in ACL2 of the exercise suggested in Subsection 2.1 was developed from scratch, without using any ACL2 library. Since our aim was to be close of Kenzo, we deal only with *ordered* simplicial complexes. As a consequence, simplexes were represented as ordered lists of vertexes. In the same vein, simplicial complexes (as sets of simplexes, more generally) were represented as lists of simplexes without duplicates.

In the case of general (non-ordered) simplicial complexes, one could think of using the "ACL2 Finite Set Theory"[1], to make a parallel development.

Let us note that ACL2 is an untyped logic, it uses type information internally to deduce types. ACL2 users provide the prover with type information by specifying type hypotheses on variables in a conjecture. Although ACL2 is syntactically untyped, that does not prevent users from having and using a notion of a type. One cannot create new types in ACL2, in the sense that one cannot create a new non-empty set of values that provably extends the ACL2 value universe. This value universe is divided into 5 kinds of data objects: Numbers, Characters, Strings, Symbols and Conses (Ordered Pairs). Rather, one typically partitions the existing universe in potentially new ways to form 'new' sets. These sets ("types") are presently characterized by just a type predicate.

Taking into account the previous paragraph, two type predicates were defined to formalize the notions of "simplex" and "list of simplexes". Besides, both the relation "be a face of" between two simplexes and the relation "be in" between a simplex and a list of simplexes were developed. Afterwards, the algorithm to construct the simplicial complex (a list of simplexes with some properties) associated with a simplex (this corresponds with the powerset of a finite set) was defined. Then, two theorems were provided to prove the correctness of the algorithm.

The algorithm computing a simplicial complex of a given list of simplexes, `list`, is split in two steps. The former one, consists of gathering the lists of simplexes coming from the computation of the simplicial complex of each simplex of `list`; as a result a list of simplexes is produced but probably with duplicate elements. The second step is devoted to remove the duplicate elements.

The proof of the correctness of this algorithm was developed using "the Method"[2] that is the recommended procedure by the authors of the system to tackle a proof project.

[1] `http://userweb.cs.utexas.edu/~moore/publications/finite-set-theory/index.html`
[2] `http://userweb.cs.utexas.edu/users/moore/acl2/current/THE-METHOD.html`

2.3 SSReflect formalization

For the problem suggested in Subsection 2.1, we decided to use the library of finite sets as already provided in the SSReflect library, under the name "finset". Of course, this imposes working on a fixed finite set of vertexes.

A first apparent need was to define a notion of *powerset* of a finite set. There is an operator provided in the finset library that gives a more general operator, which we specialized to implement the notion, and then we wrote two lemmas showing the relation between powerset and subset (this is a first draft, and probably the two lemmas should be replaced by a single equality lemma expressing the equivalence between being a subset and being an element of the powerset).

There is a type of set of vertexes, named simplex in our development, and we also manipulate sets of sets. The SSReflect library also provides the construction to view this a type of sets, with various set operations: union, etc.

In this setting, computing the simplicial complex of a given collection of simplexes is simply done by performing the big union of all the powersets of each simplex, thus it can be written in a single formula using a "big operator", so we use a union operation at each step.

In a first try, we described the input of `create_sc` as a sequence. This is given as a module named `first_try` in the file. Working with a sequence gives a simple way to organize proofs by induction, but it makes that the output of `create_sc` (which is a collection of simplexes) does not have the same type as its input (which is also a collection of simplexes). In another try, we described the input of `create_sc` as a set of simplexes. The advantage is the type of sequences of simplexes is not used anymore in our formalization. On the other hand, induction proofs are done on the cardinal of the input set, which implies a few added proof steps. The comparison between the two developments shows where computations about cardinals have to be added, but the proof structure remains basically the same. For instance, the sequence `si::s` is represented by `s'` (with the added knowledge that `si` is in `s'`) in the set-based formalization, and the sequence `s` is represented by `s' :\ si`, the set `s'` from which `si` is removed.

2.4 Comparing the SSReflect and ACL2 formalizations: a first look

An "on surface" comparison of the two formalizations is started here. A more thoughtful analysis would be necessary in the future.

In Figure 2 the powerset definition in ACL2 and SSReflect can be seen. Observe that the ACL2 definition is more verbose, for two reasons: first, it deals with lists instead of sets, and, second, the SSReflect is taking profit of "finset" library.

With respect to proofs, Figure 3 shows the theorems for an equivalent property in ACL2 and SSReflect (namely, an element of the powerset of $\mathcal{S}$ is a subset of $\mathcal{S}$, and viceversa). Please, remark the different styles in ACL2 and SSReflect. In ACL2 the prover is guided by means of some lemmas. On the contrary, SSRe-

ACL2

```
(defun simplex-p (simplex)
  (if (endp simplex)
      (equal simplex nil)
    (if (endp (cdr simplex))
        (and (equal (cdr simplex) nil)
             (natp (car simplex)))
      (and (natp (car simplex))
           (natp (cadr simplex))
           (< (car simplex) (cadr simplex))
           (simplex-p (cdr simplex))))))

(defun map-cons (x s)
  (if (endp s)
      nil
    (cons (cons x (car s))
          (map-cons x (cdr s)))))

(defun powerset (l)
  (if (endp l)
      (list nil)
    (append (powerset (cdr l))
            (map-cons (car l)
                      (powerset (cdr l))))))
```

SSReflect

```
Module Type context.

Variable V : finType.

End context.

Module first_try (CTXT : context).
Import CTXT.

Definition simplex := {set V}.

Definition powerset (x : simplex) :
        {set simplex} := setT ::&: x.
```

Figure 2: Definitions about powerset in ACL2 and in SSReflect

flect proofs are led by interactive tactics. This gives a more compact structure in SSReflect, which in addition benefits from the use of libraries. Keeping apart these differences, both proofs can be considered almost “isomorphic”. This suggests that, for a convenient and fruitful comparison, perhaps we should choose a scale of comparison greater than these small definitions and statements.

2.5 Future work

As immediate future work, it is planned to obtain from a 2-dimensional finite simplicial complex (as those associated to $2D$ digital images) its pair of *incidence matrices*. Then, the theorem to be proved is that the product of these two incidence matrices is null. To this aim, it is necessary to start from an *ordered* simplicial complex. This is already made in ACL2 (through the Kenzo style of working). For SSReflect, let us remark that a “finset” always has an implicit order among its elements (induced from an enumeration of them). If it would be enough to this further work, or if it would be better to change to a more “list-oriented” style in SSReflect is still to be decided. Other pending decisions are related to the right degree of abstraction (and, in particular, on whether it would be convenient or not to introduce more concepts in the formalization, as simplicial sets and chain complexes).

ACL2

```
(encapsulate
 ()

 (local (defthm powersetE-lemma1
   (implies (and (consp m)
                 (not (in-p (car m) l)))
            (not (in-p m (powerset l))))))

 (local (defthm powersetE-lemma2
   (iff (in-p x (append l m))
        (or (in-p x l) (in-p x m)))))

 (local (defthm powersetE-lemma3
   (implies (and (consp m)
                 (not (in-p (cdr m) l)))
            (not (in-p m (map-cons x l))))))

 (local (defthm powersetE-lemma4
    (implies (and (consp m)
                  (in-p m (powerset l)))
             (in-p (cdr m) (powerset l)))))

 (defthm powersetE
    (implies (in-p s1 (powerset s2))
             (face-p s1 s2)))

 )

 ;; Intermediary lemmas are not included

(defun in-powerset-schema (s1 s2)
  (declare (xargs :measure (acl2-count s2)))
  (if (or (not (simplex-p s1))
          (not (simplex-p s2))
          (not (face-p s1 s2)))
      nil
    (if (endp s2)
        t
      (in-powerset-schema (cdr s1) (cdr s2)))))

(defthm in-powerset
    (implies (and (simplex-p s1)
                  (simplex-p s2)
                  (face-p s1 s2))
             (in-p s1 (powerset s2)))
  :hints (("Goal" :induct (in-powerset-schema s1 s2))))
```

SSReflect

```
Lemma powersetE :
  forall (x : simplex) (y : simplex),
  y \in powerset x -> y \subset x.
Proof.
rewrite /powerset.  rewrite /ssetI; move => x y.
rewrite [y \in _]inE.
by move/andP => [_].
Qed.

Lemma in_powerset : forall (x y : simplex),
  y \subset x -> y \in powerset x.
Proof.
move => x y xy; rewrite inE.
rewrite xy andbT.
by [].
Qed.

End first_try.
```

Figure 3: Theorems about powerset in ACL2 and in SSReflect

Appendix: SSReflect Exercises

Part I

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat div seq.
Require Import paths fintype tuple finset.

Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.

(* Prove the following propositional tautologies:*)

Section Tauto.
Variables A B C : Prop.

Lemma tauto1 : A -> A.
Proof.
Qed.

Lemma tauto2: (A -> B) -> (B -> C) -> A -> C.
Proof.
...
Qed.

Lemma tauto4 : A /\ B <-> B /\ A.
Proof.
...
Qed.

End Tauto.
(* Your proof script should come in place of the dots, between "Proof."
and "Qed.". Your proof is finished when the system raises a message
saying so. Then the "Qed" command rechecks the proof term
constructed by your script. In the following, we only give the
statements of the lemmas to be proved and no more repeat "Proof" and
"Qed".

The standard Coq section mechanism allows to globally factorize the
abstraction. Here in the section parameters "A B C" are fixed, and
they are discharged after the section "Tauto" is closed by the
command "End Tauto".
*)

(* Prove the following statements:*)
```

```
Section MoreBasics.
Variables A B C : Prop.
Variable P : nat -> Prop.

Lemma foo1 : ~(exists x, P x) -> forall x, ~P x.


Lemma foo2 : (exists x, A -> P x) -> (forall x, ~P x) -> ~A.

End MoreBasics.
(*Hint: Remember that the intuitionistic negation ~A  is a notation
for "A -> False". Also remember that the proof of an existential
statement is pair of the witness and its proof, so you can destruct
this pair by the "case" command.*)

(*
The ssr ssrnat library crucially redefines the comparison
predicates and the operations on natural numbers. In particular,
comparisons are boolean predicates, instead of the inductive versions
proposed by Coq standard library. What is the definition of the
ssr "leq" predicate, denoted "<="? What is the definition of
"<"? Use the "Search" and "Check" commands *)
(*Prove the following statements:*)

Lemma tuto_subnn : forall n : nat, n - n = 0.

Lemma tuto_subn_gt0 : forall m n, (0 < n - m) = (m < n).

Lemma tuto_subnKC : forall m n : nat,
m <= n -> m + (n - m) = n.

Lemma tuto_subn_subA : forall m n p,
  p <= n -> m - (n - p) = m + p - n.



(*
*** Local Variables: ***
*** coq-prog-name: "~/coq/coq-8.2pl1/bin/ssrcoq" ***
*** coq-prog-args: ("-emacs-U" "-I"
"/Users/lrg/coq/MSR/coqfinitgroup/branches/linalg/") ***
*** End: ***
 *)
```

Part II

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat div seq.
Require Import paths fintype tuple finset.

Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.

(*
  State the lemma tuto_orP,  Prove lemmas "tuto_andP" and "tuto_orP".*)

(* Prove the lemma tuto_iffP by case analysis on the boolean
  value "b". Retry the proof, this time by case analysis on the
  hypothesis "reflect P b".*)

(* Sequences *)

(* Program the function tuto_cat  catenating two sequences.*)

(* Prove the lemma:*)
 Lemma tuto_size_cat : forall (A : Type) (s1 s2 : seq A),
     size (tuto_cat s1 s2) = size s1 + size s2.

(* Program the function "tuto_last", such that
"(tuto_last A (x : A)(s : seq A))"
returns the last element of the sequence s if s is not empty
and else returns x. Prove the lemma:*)

 Lemma tuto_last_cat : forall x s1 s2,
   last x (s1 ++ s2) = last (last x s1) s2.

(* Program the functions tuto_take (resp. tuto_drop),
of type  forall A : Type, nat -> seq A -> seq A,
such that (tuto_take n s) (resp. (tuto_drop n s))
compute the prefix of s of size n (resp. the postfix of s skipping
the n first elements), with default value s (resp. the empty
sequence  "[::]"). Prove:*)

  Lemma tuto_cat_take_drop : forall A (s : seq A),
    take n0 s ++ drop n0 s = s.

(*Program the tuto_rot function such that (tuto_rot n s) is
the circular permutation of s of order n. Prove that:*)
```

```
  Lemma tuto_rot_addn : forall A m n (s : seq A),
  m + n <= size s -> rot (m + n) s = rot m (rot n s).

(* For this last proof, you will need more lemmas about the function
programmed in this exercise. Use the \ssr Search command to find
the statements you need.*)

(*Program a function tuto_count which computes the number of
elements of a sequence which satisfy a boolean predicate.*)

Prove that:
  Lemma tuto_count_predUI : forall a1 a2 s,
    count (predU a1 a2) s + count (predI a1 a2) s
   = count a1 s + count a2 s.

(* where predU is the boolean predicate union of its two
arguments, and predI is the boolean predicate intersection of its
two arguments.*)

(* Look for the definition of the filter function. Prove that:*)

  Lemma count_filter : forall s, count s = size (filter s).


(* Prove the lemma tuto_pathP by induction on the path.*)

(* Prove the following lemmas:*)

Lemma tuto_eqxx : forall (T : eqType) (x : T), x == x.

Lemma tuto_predU1l : forall (T : eqType) (x y : T) (b : bool),
   x = y -> (x == y) || b.

Lemma tuto_predD1P : forall (T : eqType) (x y : T) (b : bool),
   reflect (x <> y /\ b) ((x != y) && b).

Lemma tuto_eqVneq : forall (T : eqType) (x : T), {x = y} + {x != y}.

(* Hint: Consider using the view mechanisms for equivalence, goal and
assumption interpretation.*)

(* Remark : try starting the proof of eqVneq  by the tactic:
  move=> T  x y; case: eqP.

What happens?*)
```

Part III

```
Require Import ssreflect eqtype ssrbool ssrnat.
Set Implicit Arguments.
Unset Strict Implicit.
Import Prenex Implicits.

Fixpoint eb (n:nat) : bool :=
  match n with 0 => true | 1 => false | S (S p) => eb p end.

Lemma ebSn : forall n, eb n.+1 = ~~ eb n.
Proof.
move => n; elim: n => [ | p Hp]; first by [].
by rewrite Hp negb_involutive.
Qed.

Lemma neg_ebS : forall n, eb n = ~~eb (n.+1).
by move => n; rewrite ebSn negb_involutive.
Qed.

Lemma ebP : forall n, reflect (exists k, n = 2 * k) (eb n).
move => n; have tmp : reflect (exists k, n = 2 * k) (eb n) *
                 reflect (exists k, n.+1 = 2 * k) (eb n.+1); last by case: tmp.
elim: n => [ | p Hp].
  split.
    by apply: ReflectT; exists 0.
  apply: ReflectF; move => [[ | v]]; first by [].
  by rewrite !mulSn mul0n addSn addnS.
move: Hp => [Hp1 Hp2] /=; split; first by [].
  case: Hp1 => [ x | nx ]; [apply: ReflectT | apply: ReflectF].
  by case: x => [k q]; rewrite q; exists k.+1; rewrite mulnS.
move => [[ | k]]; first by rewrite muln0.
by rewrite mulnS; move => [[q]]; case: nx; exists k.
Qed.

Lemma ebmul : forall n m, eb n -> eb (n*m).
Proof.
  move => n m p; move/ebP: p => [k p]; apply/ebP; exists (k * m).
  by rewrite mulnA p.
Qed.

Lemma ebmulr : forall n m, eb (n*m) -> eb n || eb m.
move => n m; case hn: (eb n); first by [].
```

```
rewrite orb_false_l; case hm: (eb m); first by [].
have {hn} hn:= negbT hn; have {hm} hm:= negbT hm.
rewrite -ebSn in hn; rewrite -ebSn in hm.
move/ebP: (hn) => [kn qn].
move/ebP: (hm) => [km qm].
move/ebP=> [kmn qmn].
have :eb (n.+1 * m.+1) by apply: ebmul.
rewrite mulnS mulSn neg_ebS -addnS -addSn qn qm qmn; case/negP.
by rewrite -!muln_addr; apply/ebP; exists (kn + (km + kmn)).
Qed.

Lemma mmnn2: forall m n, m * m = 2 * (n * n) -> n = 0.


move => m; elim: m {-2} (m) (leqnn m).
move => m; rewrite leqn0; move/eqP => ->.
  rewrite muln0; move => [ | n]; first by [].
  by rewrite !mulSn.
move => m' Hm' m mm' n q.
have evm : eb m.
  have evmm : eb (m * m) by apply/ebP; exists (n * n).
  by have := ebmulr evmm; case/orP.
case nn0 : (n == 0); first by apply/eqP.
have {nn0} nn0:= neq0_lt0n nn0.
have cmp : n * n < m * m.
  rewrite q; apply: ltn_Pmull; first by [].
  by rewrite muln_gt0 nn0.
have cmp' : n < m.
  case cmp': (n < m); first by [].
  have {cmp'} := negbT cmp'; rewrite -ltnNge ltnS => cmp'.
  move: (cmp'); rewrite - (@leq_pmul2l m).
    move => cmp2; move: (cmp').
    rewrite -(@leq_pmul2r n) => //.
    move => cmp3; move: cmp; rewrite ltnNge; move/negP => h; case: h.
    by apply: leq_trans cmp3.
  rewrite lt0n; apply/negP; move/eqP => q'; move: cmp.
  by rewrite q' muln0 ltn0.
move/ebP: evm => [km qm].
move/eqP: q; rewrite qm -mulnA eqn_mul2l /= (mulnC km) -mulnA eq_sym => q.
have nm : n <= m'.
  by rewrite -ltnS; apply: leq_trans  mm'.
have := (Hm' n nm km (eqP q)).
by move => q'; move : cmp'; rewrite qm q' muln0 ltn0.
Qed.
```