Integrating an Automated Prover for Projective Geometry as a New Tactic in the Coq Proof Assistant

Nicolas Magaud
ICube UMR 7357 CNRS - Université de Strasbourg
magaud@unistra.fr

Recently, we developed an automated theorem prover for projective incidence geometry. This prover, based on a combinatorial approach using matroids, proceeds by saturation using the matroid rules. It is designed as an independent tool, implemented in C, which takes a geometric configuration as input and produces as output some Coq proof scripts: the statement of the expected theorem, a proof script proving the theorem and possibly some auxiliary lemmas. In this document, we show how to embed such an external tool as a plugin in Coq so that it can be used as a simple tactic.

1 Introduction

The Coq proof assistant [7, 3] is a generic theorem prover, with huge capabilities. One of its main strengths is that it allows carrying out proofs either interactively or under some assumptions automatically. Its standard tactics allow to solve goals in some well-understood fragments of its underlying logic, e.g. first order logic using the tactic `firstorder` or linear arithmetic using the tactic `lia`. When the proofs get more technical or are outside of the scope of these automatic tactics, the user can take control and thus we are not limited any more by the power of automation. Contrary to what happens with SMT provers, like Z3 [12] or Vampire [9], which either succeed or fail on the goal, in Coq, the user can perform some proof steps interactively and then rely again on automated tools to solve the goal.

However, it requires a lot of expertise to be able to develop some new tactics in Coq to help the user proving theorems more easily. Most tactics are written using the Ltac tactic language [8], but writing tactics sometimes also requires to implement some features in OCaml in a Coq plugin.

In the context of geometry, we developed an independent theorem prover, implemented in C, which handles first-order statements. It takes as input a geometric configuration, written as plain text, and returns a Coq proof script, ready to be verified (type-checked) by Coq. We choose to proceed that way in order to make the development as easy as possible for a non-expert in Coq internals. The only requirements are to know the specification and the tactic languages of Coq, and thus to be able to produce some Coq files (actually a simple `.v` file).

In this article, we show how to integrate such an automated prover as a tactic in Coq, without modifying the interface of the automated prover.

In Sect. 2, we briefly present the automated prover, which we shall see as a blackbox in the rest of this document. In Sect. 3, we show how to translate a Coq goal into the input language of our prover. In Sect. 4, we investigate how to use the Coq script produced by our prover to solve the initial Coq goal. In Sect. 5, we discuss our implementation choices and explain how it could be extended to other external provers.

Chantal Keller and Mathias Fleury (Eds.): Seventh Workshop
on Proof eXchange for Theorem Proving (PxTP)
EPTCS ??, 2021, pp. 48–55, doi:10.4204/EPTCS.??.6

© N. Magaud
This work is licensed under the
Creative Commons Attribution License.

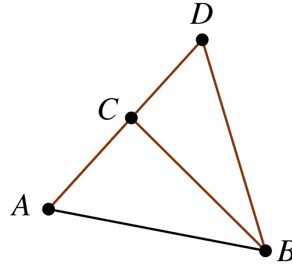

Figure 1: A simple geometric configuration illustrating the statement of Fig. 2

2 The automated prover

We work in the framework of projective incidence geometry, which is one of the simplest theory that can be used to capture most aspects of geometry. It is based on an incidence relation between points and lines. In its projective version, we assume that two coplanar lines always intersect. One of its main advantages is that it can be described using only a few axioms; thus it is a well-suited framework to try and automate proofs of theorems. This geometry can be described either in a synthetic way, writing statements using the incidence relation $\in$, or in a more combinatorial way, using the rank of a set of points [11]. For instance, the property that a line has at least three distinct points can be expressed in a synthetic way as follows :

$$\forall l : Line, \exists ABC : Point, A \neq B \land B \neq C \land A \neq C \land A \in l \land B \in l \land C \in l.$$

It can also be implemented as the following statement using the matroid based description using ranks :

$$\forall AB : Point, rk\{A,B\} = 2 \Rightarrow \exists C : Point, rk\{A,B,C\} = rk\{B,C\} = rk\{A,C\} = 2.$$

In the combinatorial approach, we exclusively deal with points and geometric reasoning is replaced by some computations relying on the combinatorial properties of the underlying matroid on which the rank properties are based [11]. Concretely, the rank of a set of two distinct points A and B is equal to 2. The rank of a set of three collinear points A, B, and C is also equal to 2. The rank of the whole plane in a two dimensional setting is 3. In a space of dimension $n \geq 3$, the maximum rank of a set of points is $n+1$. In the case $n = 3$, a set of points whose rank is 4 is not a plane and actually captures the whole space.

The above two approaches are shown to be equivalent [5] and the combinatorial one can be successfully used to automatically prove some emblematic theorems of 3D projective incidence geometry [4]. Among them we can cite Desargues' theorem and Dandelin-Gallucci's theorem [6]. The automated prover, named Bip for *matroid **B**ased **I**ncidence **P**rover*, is designed to prove equality between ranks of various sets of points. It is based on rank interval computations. For each subset of the powerset of the geometric configuration, we define the minimum and the maximum rank (in the worst case, when no information is known, the rank of each non-empty subset is between 1 and 4). We then use the matroid axioms, which specify the rank function, and we reformulate them as rewrite rules to incrementally reduce the size of the interval for each subset. This is achieved using a saturation algorithm, which is run on a valuated graph implementing the inclusion lattice of the point powerset, labeled by the minimum and maximum rank. Once the saturation graph is built, it is traversed to build a Coq proof script which

```
Lemma ex2 :
    forall A B C D:Point,
    rk(A :: D :: B :: nil) = 3 ->
    rk(A :: C :: D :: nil) = 2 ->
    rk(C :: A :: nil) = 2 ->
    rk(C :: D :: nil) = 2 ->
    rk(A :: C :: B :: nil) = 3.
Proof.
```

```
context
  dimension 3
  layers 1
endofcontext
layer 0
 points
A B C D
 hypotheses
C D  : 2
C A  : 2
A C D  : 2
A D B  : 3
  conclusion
A C B  : 3
endoflayer
conclusion
A C B  : 3
end
```

Figure 2: The Coq statement of a simple projective geometry theorem (left) and the corresponding input for the automated prover (right)

actually proves the statement at stake. Technical details about the implementation can be found in [4] and the current implementation of the prover as well as several examples of applications are available in the git repository: `https://github.com/pascalschreck/MatroidIncidenceProver`.

In the next sections, we show how to integrate such a tool so that it can be simply used as a tactic in Coq.

3 Translating a Coq statement into the input language of the prover

Provided that we define the theory of ranks in Coq (see Appendix A), as we first did in [10], we can state geometric properties as Coq lemmas using equalities on the ranks of some sets of points.

A Coq statement featuring some rank properties can easily be translated into an input file for the automated prover. It simply consists in traversing the context and the goal, harvesting all variables of type `Point` and all statements of the form rk(e)=k. All other statements are simply dropped, as they could not be used by the automatic prover yet. Fig. 2 shows the correspondence between the Coq statement of a simple theorem in projective geometry, depicted in Fig. 1, and its translation into the input langage of our automated prover.

This translation can be easily implemented inside a Coq plugin, following the examples of the plugin tutorial available in the reference manual of Coq [7]. It produces a simple text file, which can be then used as input by the Bip prover. In the generated input of the Bip prover shown in the right part of Fig.2, layers are a legacy tool which was used to decompose the context at stake. This allows to build smaller proof scripts which can be easily checked by Coq. In the current version of the prover, each deduced statement is stated as a lemma and thus layers are not needed anymore. Once the input file is generated, we simply execute the external prover from Coq using the `Unix.system` primitive of the OCaml library. It produces a Coq file (`.v`), with the appropriate prelude, which is ready to be verified

by Coq.

An implementation of this translation as well as the mechanisms to use the produced proof inside Coq is available in this git repository: `https://github.com/magaud/projective-prover`. It works with Coq 8.12.2 (December 2020) and we are currently updating it to the most recent release of Coq.

4 Running the (generated) Coq script and proving the goal

Assuming the automated prover manages to produce a Coq proof script which proves the goal at stake, we still need to load this Coq file inside Coq and use the externally proven lemma to actually prove the initial goal. Let us check what happens with our simple example. As the automated prover reorders the points in the sets at stake (they are ordered according to the order in which they were introduced, during the construction of the input file), the automatically generated statement and the initial one slightly differ. In our case, the prover produces the following statement in a file named `pprove_ex2.v`[1]:

```
Lemma LABC : forall A B C D ,
  rk(A :: C ::  nil) = 2 ->
  rk(A :: B :: D ::  nil) = 3 ->
  rk(C :: D ::  nil) = 2 ->
  rk(A :: C :: D ::  nil) = 2 ->
  rk(A :: B :: C ::  nil) = 3.
Proof.
```

Proving the initial goal `ex2` from Sect.3 requires compiling all the automatically generated code leading to the proof of lemma `LABC` and loading it in Coq. Then the statement `LABC` can be almost directly applied to prove the statement at stake `ex2`. The slight differences between these two statements lie in the order of the points in each hypothesis or conclusion of the form $rk(e) = n$. Points need to be reordered according to the order in which they are introduced in the context. This is achieved by running the Ltac code `solve_using`[2] which transforms the initial goal until its exactly matches the automatically-proven statement.

```
Require Import pprove_ex2.
solve_using LABC.
```

At this point, even if we can be fairly confident that the whole machinery works properly, we do not have a strong warranty that everything went as planned. The last step of the proof is, as usual in Coq, to type-check the built proof using the concluding command `Qed`. Once this step is completed, we are sure that our prover actually proved the initial statement provided by the user.

5 Discussion

We choose a simple but efficient approach to run the prover inside Coq. One of the main advantages of this approach is that it does not require the designer of the external prover to have any understandings of the internals of Coq. It simply requires the developer to be knowledgeable of the way Coq scripts

[1]The name of the Lemma is automatically generated, starts from the `L` of Lemma, followed by all the names of the points used in the conclusion.

[2]The code of `solve_using` is in the file `Ltac_utils.v`

are written. This is a skill every Coq user has. Such an approach thus makes every Coq user a potential tactic writer, provided we can build an interface linking Coq statements to the expected inputs of such an external prover. Indeed, the critical part of our tool is the translation of the statement from Coq to the input language of the prover. The proof generation does not need to be formally verified. Indeed, Coq will reject the proof script if it does not actually prove the statement at stake (either because statements do not coincide, or because the proof script is incorrect).

The connection from Coq to the automated prover is fairly straightforward. However the embedding of the automated prover into Coq is a bit more technical. It is not fully automated yet because calling a command (to load a compiled `.vo` file) from a tactic requires subtle interactions with the State Transaction Machine (which allows for faster and parallel executions of some parts of the proofs [2]) to maintain consistency of the proof document. For the time being, we make the `pprove` tactic display the command and tactic to be applied to complete solving the goal. In the near future, we plan to integrate these two extra steps directly into the code of the `pprove` tactic.

We think this simple way of embedding the prover could be generalized to other external provers. So far, our automated prover is running from scratch every time we compute the proof again. We plan to add some memoization techniques in order to avoid recomputing the saturation every time. We could aim at a more incremental tool where the saturation mechanism is carried out when a new point is added to the context.

More integrated tools to make automated proofs exist [1], designed to embed SAT and SMT solvers inside Coq and ensure that they provide correct decision procedures. Their approach is safer than ours, but we argue that simply generating a Coq file is much easier to carry out by an average programmer and can be used as a first step towards integrating an automated tool into Coq.

6 Conclusions and Perspectives

We successfully embed the external prover Bip as a simple tactic to be used in Coq using Ltac. We proceed in two steps: we first develop a prover, which is completely independent from the internals of Coq. The second step consists in connecting this external prover to Coq. We call the prover from Coq and retrieve the external proof script it generates. We then simply feed this script to Coq to type-check it. The API of Coq is not intended to make it easy to embed some Coq proof scripts generated externally to prove a goal. We believe it would be a nice feature to ease this process by allowing a tactic to simply run a snippet of Coq code loaded from a file.

In the longer run, improving our prover requires to make it able to take synthetic geometry statements as input rather than sets of points and ranks. We carry on developing a way to automatically translate synthetic geometry statements into ranks equalities in Coq, so that the user deals with synthetic geometry, whereas the prover deals with sets of points and their ranks in the back-end. A draft implementation of this translation is available in the file `translate.v` of the git repository.

Finally, we think our example describes an efficient and straight-to-the-point approach to embed an automated prover inside Coq and is useful as a first step towards a more integrated embedding.

References

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In Jean-Pierre Jouannaud & Zhong Shao, editors: *Certified Programs and Proofs - First International Conference, CPP 2011*,

Kenting, Taiwan, December 7-9, 2011. Proceedings, *Lecture Notes in Computer Science* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] Bruno Barras, Carst Tankink & Enrico Tassi (2015): *Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface*. In Christian Urban & Xingyuan Zhang, editors: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, *Lecture Notes in Computer Science* 9236, Springer, pp. 51–66, doi:10.1007/978-3-319-22102-1_4.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin/Heidelberg, doi:10.1007/978-3-662-07964-5. 469 pages.

[4] David Braun (2019): *Approche combinatoire pour l'automatisation en Coq des preuves formelles en géométrie d'incidence projective*. Ph.D. thesis, Université de Strasbourg. Available at `https://tel.archives-ouvertes.fr/tel-02512327`.

[5] David Braun, Nicolas Magaud & Pascal Schreck (2019): *Two Cryptomorphic Formalizations of Projective Incidence Geometry*. *Annals of Mathematics and Artificial Intelligence* 85(2–4), pp. 193–212, doi:10.1007/s10472-018-9604-z. Available at `https://hal.inria.fr/hal-01887000`.

[6] David Braun, Nicolas Magaud & Pascal Schreck (2021): *Two New Ways to Formally Prove Dandelin-Gallucci's Theorem*. In: *Proceedings of the 2021 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2021, Saint Petersbourg (Virtual), Russia, July 18-23, 2021*, ACM, pp. –, doi:10.1145/3452143.3465550.

[7] Coq development team (2021): *The Coq Proof Assistant Reference Manual, Version 8.13.2*. INRIA. Available at `http://coq.inria.fr`.

[8] David Delahaye (2000): *A Tactic Language for the System Coq*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, *Lecture Notes in Computer Science* 1955, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7.

[9] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, *Lecture Notes in Computer Science* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[10] Nicolas Magaud, Julien Narboux & Pascal Schreck (2012): *A Case Study in Formalizing Projective Geometry in Coq: Desargues Theorem*. *Computational Geometry: Theory and Applications* 45(8), pp. 406–424, doi:10.1016/j.comgeo.2010.06.004. Available at `http://hal.inria.fr/inria-00432810/en`.

[11] Dominique Michelucci & Pascal Schreck (2006): *Incidence Constraints: a Combinatorial Approach*. *Int. Journal of Computational Geometry and Applications* 16(5-6), pp. 443–460, doi:10.1142/S0218195906002130.

[12] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of TACAS 2008*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

A A few lines of code specifying the rank function and its properties

We outline the context describing the rank function which is loaded before starting the proof in Coq. The axioms describe the properties of the rank function and are used in the automatically built Coq proof script. See files `basic_matroid_list.v` and `basic_rank_space_list.v` for details.

```
Parameter Point : Set.

Parameter eq_dec : forall A B : Point, {A = B} + {~ A = B}.
```

```
Definition equivlist (l l':list Point) :=
  forall x, List.In x l <-> List.In x l'.

Parameter rk : list Point -> nat.
Parameter rk_compat :
  forall x x', equivlist x x' -> rk x = rk x'.

Global Instance rk_morph : Proper (equivlist ==> (@Logic.eq nat)) rk.

Parameter matroid1_a : forall X, rk X >= 0.
Parameter matroid1_b : forall X, rk X <= length X.
Parameter matroid2 :
  forall X Y, incl X Y -> rk X <= rk Y.
Parameter matroid3 :
  forall X Y, rk(X ++ Y) + rk(list_inter X Y) <= rk X + rk Y.

Parameter rk_singleton_ge :
  forall A, rk (A :: nil)  >= 1.
Parameter rk_couple_ge :
  forall A B, ~ A = B -> rk(A :: B :: nil) >= 2.

Parameter rk_three_points_on_lines :
  forall A B, exists C, rk (A :: B :: C :: nil) = 2 /\
                        rk (B :: C :: nil) = 2 /\
                        rk (A :: C :: nil) = 2.

Parameter rk_inter :
  forall A B C D, rk(A :: B :: C :: D :: nil) <= 3 ->
  exists J : Point, rk(A :: B :: J :: nil) = 2 /\
                    rk (C :: D :: J :: nil) = 2.

Parameter rk_lower_dim :
  exists A0 A1 A2 A3, rk( A0 :: A1 :: A2 :: A3 :: nil) = 4.
Parameter rk_upper_dim :
  forall e, rk(e) <= 4.
```

B An example of an automatically generated file

The file `pprove_ex2.v`, which is 163-line long including comments, is automatically generated in the process of proving the statement `ex2`. It is available (or can be regenerated) from `https://github.com/magaud/projective-prover/blob/master/theories/pprove_ex2.v`. All statements, including intermediary statements have exactly the same hypotheses. In addition, there is exactly one statement for each deduction achieved during the saturation phase of the algorithm. The last statement `LABC` corresponds to the actual statement proved automatically. This lemma is then reused to prove the initial Coq goal: `ex2` in our example.

```
Require Import lemmas_automation_g.

Lemma LB : forall A B C D ,
rk(A :: C ::  nil) = 2 -> rk(A :: B :: D ::  nil) = 3 ->
rk(C :: D ::  nil) = 2 -> rk(A :: C :: D ::  nil) = 2 ->
rk(B ::  nil) = 1.
Proof. [...] Qed.

Lemma LAC : forall A B C D ,
rk(A :: C ::  nil) = 2 -> rk(A :: B :: D ::  nil) = 3 ->
rk(C :: D ::  nil) = 2 -> rk(A :: C :: D ::  nil) = 2 ->
rk(A :: C ::  nil) = 2.
Proof. [...] Qed.

[...]

Lemma LABC : forall A B C D ,
rk(A :: C ::  nil) = 2 -> rk(A :: B :: D ::  nil) = 3 ->
rk(C :: D ::  nil) = 2 -> rk(A :: C :: D ::  nil) = 2 ->
rk(A :: B :: C ::  nil) = 3.
Proof.
intros A B C D
HACeq HABDeq HCDeq HACDeq .
assert(HABCm2 : rk(A :: B :: C :: nil) >= 2).
{
try assert(HACeq : rk(A :: C :: nil) = 2)
by (apply LAC with (A := A) (B := B) (C := C) (D := D) ;
try assumption).
assert(HACmtmp : rk(A :: C :: nil) >= 2)
by (solve_hyps_min HACeq HACm2).
assert(Hcomp : 2 <= 2) by (repeat constructor).
assert
  (Hincl : incl (A :: C :: nil) (A :: B :: C :: nil))
        by (repeat clear_all_rk;my_inO).
assert
  (HT := rule_5 (A :: C :: nil) (A :: B :: C :: nil)
         2 2 HACmtmp Hcomp Hincl);apply HT.
}
assert(HABCm3 : rk(A :: B :: C :: nil) >= 3).
[...]
assert(HABCM : rk(A :: B :: C ::  nil) <= 3)
by (solve_hyps_max HABCeq HABCM3).
assert(HABCm : rk(A :: B :: C ::  nil) >= 1)
by (solve_hyps_min HABCeq HABCm1).
intuition.
Qed.
```