General automation in Coq through modular transformations*

Valentin Blot[1,2], Louise Dubois de Prisque[1,2], Chantal Keller[1], and Pierre Vial[1,2]

[1] Université Paris-Saclay, CNRS, ÉNS Paris-Saclay, Laboratoire Méthodes Formelles
[2] Inria

Abstract

In this talk, we will present a methodology to transform a subset of Coq goals into first-order statements that can be automatically discharged by automatic provers. The general idea is to write modular, pairwise independent transformations and combine them. Each of these eliminates a specific aspect of Coq logic towards first-order logic. As a proof of concept, we apply this methodology to a set of simple but crucial transformations which extend the local context with proven first-order assertions that make Coq definitions and algebraic types explicit. They allow users of Coq to solve non-trivial goals automatically. This methodology paves the way towards the definition and combination of more complex transformations, making Coq more accessible.

1 Interaction *vs.* automation.

It is often observed (and regretted) that the lack of automation of Coq is a barrier for it to be widely used by mathematicians or in the industry. Indeed, the rich logical specifications that the underlying **Calculus of Inductive Constructions (CIC)** enables makes it difficult to be automated. This makes a stark contrast with automated provers, which are based on less expressive logics (in particular, **first-order logic**) and lack strongly trusted logical kernels (as Coq has) but have powerful proof-search heuristics.

Automation in presence of higher-order computation, full polymorphism and dependent types is a vast and highly non-trivial topic, but it is quite noticable that **Coq lacks automation even when dealing with first-order goals** on decidable types, whereas the proof would be fully automated in external solvers! Even in this case, the user (particularily a beginner) has to be highly precise in the way they compose their keywords, *e.g.*, they often need to specify how they instantiate variables of lemmas and deal with a lot of trivial details which would be far less tedious on paper, as Example 1 below illustrates.

In this workshop, we will present:

(a) A general methodology to make Coq easily resort to automated provers while dealing with goals which can be reduced to first-order logic. As we will see, (1) we automatically prove various first-order statements in Coq and do a bit of goal transformation (2) we send all these statements to an external automated prover.

(b) An implementation of this methodology based on the Coq plugin SMTCoq. This gives the tactic `snipe` (for sake of the bird known in French as the *Bécassine des Marais*).

Let us examine an example in which automation in Coq would be useful to save user time, as the goal is first-order.

Example 1 (Dealing with datatypes)**.** For example, if want to prove:

```
Goal forall (A: Type) l (a:A), hd_error l = Some a → l <> nil.
```

A typical Coq proof is: `intros A l a H. intro H'. rewrite H' in H. simpl in H. inversion H.`

The statement is straightforward, its proofs relies on the fact the constructors of an inductive type are pairwise disjoint. Yet, in Coq, the user has to be highly precise in the way they compose their keywords, whereas they would not even bother writing the proof on paper.

*This work is funded by a Nomadic Labs-Inria collaboration.

2 Our contribution

We provide a methodology to reconcile Coq goals with the logic of first-order provers. For instance, we get a fully automatic proof of Example 1. This methodology consists of

1. Implementing pairwise independent logical transformations which are *certifying* and use the meta-programming tools Ltac [1] and MetaCoq [3]. *Certifying* means that the transformation produces a proof of its own soundness when it succeeds. This is not surprising as we implement Coq tactics. But our methodology is more general as it is also possible to write *certified* transformations in which the proof of soundness is established previously as a separate Coq proof, once and for all. These transformations are used to state and prove *in Coq* various auxiliary lemmas of first-order logic about the terms that appear in the goal which are then stored in the local context. It may also perfom some transformations on the goal, so that it becomes first-order.

2. Sending this first-order goal and all the auxiliary statements produced in the local context to an external first-order prover. If this prover solves the goal with the information it has been provided, a Coq proof term is built (before being type-checked) from the certificate output by the prover.

We implement this methodology as a new Coq tactic called `snipe`. The two phases above are implemented in the following way:

1. We provide an independent transformation for each of the following use-cases: (1) add the definitions of the Coq functions which appear in the hypotheses or in the goal in the local context (2) transform equalities `f=g` between two higher-order objects into a first-order statement of the form `forall (x1: A1)... (xn: An): f x1 ... xn = g x1 ... xn` (3) eliminate the anonymous function in a fixpoint definition and replace it by the function we are dealing with (4) assert and prove the statements that the constructors of inductive types are pairwise disjoint and injective (5) instantiate all the polymorphic hypotheses (and possible lemmas) by all the subterms of type *Type* in the goal (**monomorphization)**. We provide a single tactic `scope` that suitably combines all these transformations.

2. Then, we use the SMT solver veriT as a back-end to discharge the obtained first-order goal, available through the SMTCoq plugin[1][2], which enables communication between Coq and SMT solvers. An important observation is that in this step, SMTCoq could be replaced by any tactic or tool solving first-order goals. This tool may produce Coq certificates but this is not necessary: it is also possible to use an external solver which is not certified and thus extend our trusted base. This is **the core idea of our proof of concept**.

The proof of Example 1 simply becomes a call to our tactic `snipe`. This tactic proves many more goals, combining fixpoints, datatypes, polymorphism, ... that we will show in our talk.

References

[1] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[2] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq: A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[3] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5):947–999, 2020.

[1] SMTCoq is available at https://smtcoq.github.io.