Coq's Prolog and application to defining semi-automatic tactics

Théo Zimmermann Hugo Herbelin

IRIF, Université Paris Diderot, Université Sorbonne Paris-Cité
ΠR^2, INRIA
theo.zimmermann@univ-paris-diderot.fr / hugo.herbelin@inria.fr

Abstract

We report on a work-in-progress to re-implement Coq's `apply` tactic in order to embed some form of simple automation. We design it in a declarative way, relying on `typeclasses eauto`, a tactic which gives access to the proof-search mechanism behind type classes. We qualify this mechanism of "Coq's Prolog" and describe it in a generic way and explain how it can be used to support the construction of automatic and semi-automatic tactics.

Keywords apply, automation, Coq, proof assistant, reflection, type theory, views.

1. Introduction

Since version 8.2 [2], Coq has included a type class mechanism which can intervene during type-inference. The resolution phase of this mechanism is also accessible via the user-side tactic `typeclasses eauto`. In fact `typeclasses eauto` implements a general Prolog-like proof-search mechanism that can be instrumented to do much more. While by default, it uses the specific `typeclass_instances` hint database, one can use it with alternative hint databases. In this case, despite the name of the tactic, it has nothing to do with the type class mechanism anymore (apart from the underlying implementation). It can be viewed as a re-implementation of the former `eauto` tactic, dating back from Coq version 5.10.

Many partially automated tactics have already been built upon Coq's type class mechanism. One example is the re-implementation of the `rewrite` tactic [3], which relies on the mechanism in a second phase of constraint resolution. Another is our `transfer` library [4]. We argue that both of these applications (and many others) abuse the type class mechanism, by polluting the `typeclass_instances` hint database, when they should instead define their own disjoint hint databases and rely on the underlying Prolog-like proof-search mechanism that is made accessible via the `typeclasses eauto` tactic.

We demonstrate the usefulness of this "Coq's Prolog" with a small re-implementation of the `apply` tactic. This tactic is parametrized by a set of "views" à la SSReflect [1] which it is able to apply automatically. It also demonstrates a new way of instrumenting `typeclasses eauto` to progress on a goal instead of solving it ("forward mode", proposed in particular by Michael Soegtrop on the Coq-Club mailing list).

2. Presentation of Coq's Prolog

The `typeclasses eauto` tactic is documented in the chapter on type classes of Coq's user manual [2, Chapter 20]. Like `eauto`, it is possible to specify on which hint database it should operate using the `with myhintdb` clause. By default, the proof-search depth is unbounded and the traversing strategy is depth-first. It is also possible to limit the proof-search depth, and an alternative iterative-deepening traversing strategy exists.

Most hints are lemmas which can be read as Prolog Horn clauses. For instance:

```
Lemma or_introl : forall A B, A -> A \/ B.
```

can be read as the clause[1]:

```
prove(or(A,B)) :- prove(A).
```

Such hints can be introduced with the `Hint Resolve` command:

```
Hint Resolve or_introl : myhintdb.
```

Some more complex hints can be introduced thanks to the `Hint Extern` command. One very special application of such external hints is to put some sub-goals "on the shelf":

```
Hint Extern 0 (solveLater _) =>
  unfold solveLater; shelve : myhintdb.
```

The shelf is a special place in the Coq proof engine which is normally reserved to sub-goals that will be solved by solving other sub-goals that depend on them. It can be instrumented to temporarily store away some sub-goals that we want to keep for later (at which time they will be "unshelved"). The "unshelve" tactical can be used to put back in the list of sub-goals to solve those which had been shelved by the tactic it is applied on. Thus, a combination of the previous external hint plus a call to `unshelve typeclasses eauto with myhintdb` is a way to use this Prolog-like proof-search mechanism in forward mode, instead of the normal solving mode.

Finally, a control operator on the search space is available. It is not comparable to Prolog's cut operator in that it allows to specify regular expressions of search-paths which should be cut but does not restrict backtracking. It is still limited in that regular expressions can only talk about hints declared with `Hint Resolve` and not `Hint Extern`. Additionally, there is no negative expressions and even if regular expressions are known to be closed under complement, the construction requires to know all the alphabet, which is not the case here since new hints can be added dynamically. We can expect to see improvements in control operators for `typeclasses eauto` in the next versions of Coq. Here is an example[2] of using this control operator to forbid any proof using `or_introl` twice:

```
Hint Cut [(_*) or_introl (_*) or_introl] : myhintdb.
```

[1] We introduce the Prolog predicate `prove` to denote the idea that the hints are used when looking for a proof of a statement. When using typeclasses eauto to solve type class goals, there is always a head-constant (the type class) which can serve the same role as `prove` but when using it on non-type-class goals, it is not necessarily the case.

[2] Be careful when using it that the precedence levels are not what one would expect. They should get fixed in Coq 8.7 but, in the meantime, the best way of writing forward-compatible code is to parenthesize everything.

3. An `apply` tactic with views

Our work is part of a larger effort to make the life of mathematicians using Coq easier. The specific issue we address here is to design an extension of the `apply` tactic which embeds some bits of trivial reasoning, such as reasoning modulo symmetry of equations (identifying `u = t` and `t = u`). There is already a little bit of hard-coded trivial reasoning in the current implementation (automatic decomposition of single-constructor inductive types): for instance it supports applying a theorem of the form `A -> B /\ C` to a goal of the form `B`.

SSReflect defines views which are basically small theorems of the form `A -> B` where `A` is a different way of viewing `B`. Our re-implementation of `apply` tries to solve the problem we were describing by allowing the user to parametrize it with such "views". Without any view, it is less powerful than the current implementation (because of the absence of decomposition of single-constructor inductive types). With many views, it can be much more powerful. The various levels of parametrization could reveal especially useful for teaching (from a level where everything must be done by hand, to a level where most details are handled automatically).

Given the inspiration source we described in the previous paragraph, an obvious application of our work will be to support automatic insertion of views in the context of small scale reflection. In particular, it will be possible to simply register the reflection lemma:

```
Lemma andP : forall b1 b2 : bool, reflect
  (b1 = true /\ b2 = true) ((b1 && b2) = true).
```

in a special hint database and `apply` will know about it and use it when necessary.

There are two main ideas in our implementation. The first is that we are launching a proof search to prove that the theorem we wish to apply implies the current goal:

```
?prove : arrow theorem goal
```

where `arrow` is a relation that is definitionally equal to Coq's implication. We use it because otherwise the proof-search mechanism would introduce the premise `theorem` in the proof context and then try to prove `goal` with it, and this is not what we want.

The second is that although we cannot actually prove this implication most of the time, we can provide an incomplete proof and let the user fill the holes (this is after all the principle of `apply`):

```
solveLater A -> arrow B C -> arrow (A -> B) C
```

where `solveLater` is a dummy constant introduced to call the external hint we described earlier, which will shelve the sub-goal `A` until the full search succeeds and then unshelve it for the user to prove. In fact, because the conclusion `B` might be dependent on the premise `A`, we rather write:

```
forall (t : T), arrow (U t) V ->
  arrow (forall x : T, U x) V
```

and then shelve `t`. A few other generic rules are present, one of them allowing to go under quantifiers and, of course, the reflexivity of arrow:

```
(forall x : A, arrow (f x) (g x)) ->
  arrow (forall x : A, f x) (forall x : A, g x)

arrow T T
```

With these rules only, we can reproduce the behavior of `apply`, except for the built-in handling of single-constructor inductive types. Actually already at that point, we do not reproduce the exact same behavior because our implementation allows applying a theorem `forall x y, P x y` to a goal `forall x, P x 0`.

With this basic infrastructure in place, we can start adding views, such as:

```
arrow P P' -> arrow (P /\ Q) P'
arrow Q Q' -> arrow (P /\ Q) Q'
```

which will be useful in emulating the current behavior of `apply`. We can also add a very simple rule to handle symmetry (one could think of more complex and powerful ways):

```
arrow (u = t) (t = u)
```

and finally, we register some more generic rules like:

```
reflect P b -> arrow P (b = true)
reflect b P -> arrow (b = true) P
```

so that people can easily extend the hint database with existing reflection lemmas such as the one seen above.

4. Conclusion

This work is very preliminary but shows an interesting path to removing some of the hindrances there are in using both SSReflect and vanilla Coq. It is also a demonstration of the power of `typeclasses eauto`, even when not working with type classes. Sometimes, the type class mechanism has been used while what was really wanted was this Coq's Prolog that it gives access to. One such example is the implementation of `rewrite`: it probably could and should be based on a specific hint database instead of the `typeclass_instances` database.

We are planning first to continue to test and improve our re-implementation of `apply`. We would in particular like to have views for applying a theorem modulo commutativity / associativity. We are also planning to merge this work with our previous work on applying theorems modulo isomorphisms [4]. A lot of ideas from this earlier work were reused here and the two implementations could very likely be combined together.

References

[1] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016. URL `https://hal.inria.fr/inria-00258384`.

[2] The Coq development team. The Coq proof assistant reference manual. Technical report, INRIA, 2016. URL `https://coq.inria.fr/`. Version 8.6beta1.

[3] M. Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2010.

[4] T. Zimmermann and H. Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. In *Conference on Intelligent Computer Mathematics, work-in-progress track*, 2015. URL `https://arxiv.org/abs/1505.05028`.