Project Proposal: Relieving User Effort for the Auto Tactic in Coq with Machine Learning

Lasse Blaauwbroek*

Czech Institute for Informatics, Robotics and Cybernetics, Czech Republic
Radboud University Nijmegen, the Netherlands
lasse@blaauwbroek.eu

We propose to enhance the `auto` tactic in Coq with machine learning, aiming to reduce the effort the user has to put in designing hint databases for `auto`. We seek ideas and advice regarding the specific type of ML that would be appropriate in this context.

Proof Styles in Coq The Coq Proof Assistant supports many methods of proving a theorem. One can either directly write proof terms, or choose one of the tactical languages like Ltac [2] or Mtac [3]. Then there are custom sets of tactics for Ltac like SSReflect [5]. However, even within one of these paradigms there are still different styles of proving available. Some people advocate structured proofs using Coq's built-in bullet points, writing every step of the proof explicitly in hopes of increasing readability. Other people try to write very compact and tailored tactic scripts that prove a lemma in one step. This usually results in shorter and easier to maintain proofs, often at the cost of readability. All of these styles have their place, depending on the mathematical domain one is trying to formalize.

In this proposal we will focus on one specific proving style described and popularized by Adam Chlipala [1]. The concept is to provide as little proof information as possible within the tactic script of a lemma. Usually this means that one critical step of the proof is explicitly stated in the script, while the rest of the proof is pieced together by automation. For example, the critical step can be to use induction on a specific variable of the lemma. The resulting cases of the induction principle then have to be solved by the built-in `auto` tactic. This tactic is a generic prover that uses hints previously provided by the user to guide proof search. These hints usually consist of a recipe on how to use a previously declared sublemma of the proof. However, it is also possible to teach the `auto` tactic how and when to use custom tactics and complete decision procedures.

This approach has two main advantages. (1) It keeps the actual proof scripts short and therefore maintainable. If something in the development changes it should be easy to go through the development and fix the hints and proofs, if necessary at all. (2) By only using the `auto` tactic the user is forced to tease out important information about the proof and refactor this into a lemma, hint or tactical procedure. In this way, all the crucial steps will be explicitly declared and can be easily understood by readers without bogging them down with the straightforward details of the proof. The truth of a lemma would ideally be evident to a reader simply by thinking about previously provided hints for a bit, just like it is to Coq.

The `auto` Tactic Fundamentally, the auto tactic is a simple search procedure. For a proof state it can compile a list of possible actions to take, together with a priority for these actions. The resulting search space is traversed in BFS or DFS fashion until either a full proof is found or a limit is reached. The interesting part is that the list of possible actions is compiled from so-called hint-databases. These databases are meant to contain usage information for lemmas and tactics in the current development. Users can add and remove information from a database on the fly by using variations of the `Hint` vernacular. We give some examples that add hints to a database.

- `Hint Resolve thmx` will tell `auto` to try and unify the current goal with the conclusion of theorem `thmx`. On success, auto will replace the current goal with the assumptions of `thmx`.

*This work was supported by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466)

- Let `thmy` be a theorem that has an equality as its conclusion. `Hint Rewrite -> thmy` will tell auto to rewrite the goal using `thmy` if the goal unifies with the left-hand side of the equality. Any possible assumptions of `thmy` are added to the proof state.
- `Hint Extern tac` can be used to register any tactic `tac` to be run by auto. This can be useful for making `auto` do things like normalize terms or other simple steps that never go wrong. Also, since this vernacular gives us access to the full power of the tactical language, it allows us to encode much more complicated hints, as we will elaborate below.

Hint databases have to be designed with great care. Adding the wrong lemmas to a database can lead to a very large search space and even infinite loops. The larger and more complicated a development is, the more problematic this becomes. To reduce the branching factor in such developments, a hint can have a gate specifying the conditions that need to be met before the hint is used. In its simplest form, this can be a pattern that must be matched to the goal before the hint applies. It is, however, possible to write arbitrarily complicated gates using Ltac as a programming language. This way, the hint can be accepted or rejected based on the full contents of the proof state. Philosophically speaking, the goal of writing a gate is to capture the domain specific knowledge and intuition that the user has on how and when to use a lemma or tactic. A simple example is a gate for the lemma $a < b \to b < c \to a < c$. We want to apply this lemma to a goal $x < z$ only if we can expect to find a suitable y. Therefore, the gate will be a pattern on the proof state: `?x < ?y,...,?y < ?z` $\vdash$ `?x < ?z`. Note that this gate is very strict, and a much more complicated one might be required in practice.

Experience tells us that for a decently sized development, the branching factor of the search performed by `auto` has to be kept well below 1.5 to keep the system usable.[1] The gating required to reach this can be quite laborious. Conversely, it tends to not be very difficult to achieve a branching factor smaller than five. Our proposal is to bridge the gap between these factors using machine learning, bringing together the best of human intuition and the computers ability to do the grunt work.

Machine Learning for `auto` Our proposal to incorporate machine learning into auto consists of gathering information on previous runs of the auto tactic. The idea is that at the beginning of a development, hints and proofs are usually much simpler, allowing auto to find a proof easily. We can then record which hints ended up being fruitful in the context of which proof state. As the development progresses, the system can then start to leverage this information to prioritize the list of available actions to `auto` in a proof state. Actions will be more important if they have been used previously in similar states. The amount of actions the machine learning has to choose from will be quite limited because the gating of the user has already weeded out most inapplicable actions.

One fundamental challenge is that the system will not have a lot of data to learn from. This is because within a development a hint associated with a lemma would normally be used tens or at most hundreds of times. The system needs to learn quickly in terms of data. On the other hand, because there will be very few choices to be made at each point, there will be quite a lot of time to consider each choice. For these reasons, most traditional learning techniques, like neural networks, will not be immediately applicable. The simplest approach is to extract features from proof states, and perform a direct comparison with previous states. However, more symbolic methods such as approximate substring matching between goal and lemma may also be applicable [4]. During AITP we would like to gather feedback about other techniques that may suit this setting.

[1] This is partially due to the fact that Coq users generally do not, can not, and often do not want to replace the auto tactic with the found solution like is common in Isabelle. Therefore, to get a good experience, the search has to be completed within seconds.

References

[1] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[2] 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.

[3] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018.

[4] Jiaying Wang, Xiaochun Yang, Bin Wang, and Chengfei Liu. An adaptive approach of approximate substring matching. In *Database Systems for Advanced Applications - 21st International Conference, DASFAA 2016, Dallas, TX, USA, April 16-19, 2016, Proceedings, Part I*, pages 501–516, 2016.

[5] Iain Whiteside, David Aspinall, and Gudmund Grov. An essence of ssreflect. In *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, pages 186–201, 2012.