The Tactician[⋆]

A Seamless, Interactive Tactic Learner and Prover for Coq

Lasse Blaauwbroek[1,2], Josef Urban[1], and Herman Geuvers[2]

[1] Czech Technical University, Prague, Czech Republic
[2] Radboud University, Nijmegen, The Netherlands

lasse@blaauwbroek.eu, josef.urban@gmail.com, herman@cs.ru.nl

Abstract. We present Tactician, a tactic learner and prover for the Coq Proof Assistant. Tactician helps users make tactical proof decisions while they retain control over the general proof strategy. To this end, Tactician learns from previously written tactic scripts and gives users either suggestions about the next tactic to be executed or altogether takes over the burden of proof synthesis. Tactician's goal is to provide users with a seamless, interactive, and intuitive experience together with robust and adaptive proof automation. In this paper, we give an overview of Tactician from the user's point of view, regarding both day-to-day usage and issues of package dependency management while learning in the large. Finally, we give a peek into Tactician's implementation as a Coq plugin and machine learning platform.

Keywords: Interactive Theorem Proving · Tactical Learning · Machine Learning · Coq Proof Assistant · Proof Synthesis · System Overview

1 Introduction

The Coq Proof Assistant [1] is an Interactive Theorem Prover in which one proves lemmas using tactic scripts. Individual tactics in these scripts represent actions that transform the proof state of the lemma currently being proved. A wide range of tactics exist, with a wide range of sophistication. Basic tactics such as `apply lem` and `rewrite lem` use an existing lemma `lem` to perform one specific inference or rewriting step while tactics like `ring` and `tauto` implement entire decision procedures that are guaranteed to succeed within specific domains. Finally, open-ended search procedures are implemented by tactics like `auto` and `firstorder`. They can be used in almost every domain but usually only work on simple proof states or need to be calibrated carefully. Users are also encouraged to define new tactics that represent basic steps, decision procedures, or specialized search procedures within their specific mathematical domain.

When proving a lemma, the challenge for the user is to observe the current proof state and select the appropriate tactic and its arguments to be used. Often the user makes this decision based on experience with previous proofs. If the current proof state is similar to a previously encountered situation, then one

⋆ 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) and by the *AI4REASON* ERC Consolidator grant nr. 649043.

can expect that an effective tactic in that situation might also be effective now. Hence, the user is continuously matching patterns of proof states in their mind and selects the correct tactic based on these matches.

That is not the only task the user performs, however. When working on a mathematical development, the user generally has two roles: (1) As a *strategist*, the user comes up with appropriate lemmas and sometimes decides on the main structure of complicated proofs. (2) As a *tactician*, the user performs the long and somewhat mindless process of mental pattern matching on proof states, applying corresponding tactics until the lemma is proved. Many of the steps in the tactician's role will be considered as "obvious" by a mathematician. Our system is meant to replicate the pattern matching process performed in this role, alleviating the user from this burden. Hence, we have aptly named it Tactician.

To perform its job, Tactician can learn from existing proofs, by looking at how tactics modify the proof state. Then, when proving a new lemma, the user can ask the system to recommend previously used tactics based on the current proof state and even to complete the whole proof using a search procedure based on these tactic recommendations.

In our previous publication, the underlying machine learning and proof search techniques employed by Tactician and how suitable data is extracted from Coq are described [3]. It also contains an evaluation of Tactician's current proof automation performance on Coq's standard library. We will not repeat these details here. Instead, we will focus on the operational aspects and description of Tactician when used as a working and research tool. Section 2 gives a mostly non-technical overview of the system suitable for casual Coq users. That includes Tactician's design principles, its mode of operation, a concrete example and a discussion on using Tactician in large projects. Section 3 briefly discusses some of Tactician's technical implementation issues, and Section 4 describes how Tactician can be used as a machine learning platform. Finally, Section 6 compares Tactician to related work. Installation instructions and an interactive online demo of Tactician can be found at `https://blaauwbroek.eu/papers/cicm2020/`.

2 System Overview

In this section, we give a mostly non-technical overview of Tactician suitable for casual Coq users. Section 2.1 states the guiding design principles of the project, and Section 2.2 describes the resulting user workflow. On the practical side, Section 2.3 gives a simple, concrete example of Tactician's usage, while Section 2.4 discusses how to employ Tactician in large projects.

2.1 Design Principles

For our system, we start with the principal goal of learning from previous proofs to aid the user with proving new lemmas. In Coq, there are essentially two notions of proof: (1) proof terms expressed in the Gallina language (Coq's version of the Calculus of Inductive Constructions [22]); (2) tactic proof scripts written by the user that can then generate a Gallina term. In principle, it is possible to use machine learning on both notions of proof. We have chosen to learn from tactic proof scripts for two reasons:

1. Tactics scripts are a much more high-level and forgiving environment, which is more suitable for machine learning. A Gallina proof term must be generated extremely precisely while a tactic script often still works after minor local mutations have occurred. Gallina terms are also usually much bigger than their corresponding tactic script because individual tactics can represent large steps in a proof.
2. We acknowledge that automation systems within a proof assistant often still need input from the user to fully prove a lemma. Working on the tactic level allows the user to introduce domain-specific information to aid the system. For example, one can write new tactics that represent decision procedures and heuristics that solve problems Tactician could not otherwise solve. One can teach Tactician about such new tactics merely by using them in hand-written proofs a couple of times, after which the system will automatically start to use them.

Apart from the principal goal described above, the most important objective of Tactician is to be usable and remain usable by actual Coq users. Hence, we prioritize the system's "look and feel" over its hard performance numbers. To achieve this usability, Tactician needs to be pleasant to all parties involved, which we express in four basic "friendliness" tenets.

User Friendly If the system is to be used by actual Coq users, it should function as seamlessly as possible. After installation, it should be ready to go with minimal configuration and without needing to spend countless hours training a sophisticated machine learning model. Instead, there should be a model that can learn on the fly, with a future possibility to add a more sophisticated model that can be trained in batch once a development has been finished. Finally, all interaction with the system should happen within the standard Coq environment, regardless of which editor is used and without the need to execute custom scripts.

Installation Friendly Ease of installation is essential to reach solid user adoption. To facilitate this, the system should be implemented in Ocaml (Coq's implementation language), with no dependencies on machine learning toolkits written in other languages like Python or Matlab. Compilation and installation will then be just as easy as with a regular Coq release.

Integration Friendly The system should not be a fork of the main Coq codebase that has to be maintained separately. A fork would deter users from installing it and risk falling behind the upstream code. Instead, it should function as a plugin that can be compiled separately and then loaded into Coq by the user.

Maintenance Friendly We intend for Tactician not to become abandonware after main development has ceased, and at least remain compatible with the newest release of Coq. As a first step, the plugin should be entered into the Coq Package Index [25], enabling continuous integration with future versions of Coq. Additionally, assuming that Tactician becomes wildly popular, we eventually intend for it to be absorbed into the main Coq codebase.

2.2 Mode of Operation

Analogously to Coq's mode of operation, Tactician can function both in interactive mode and in compilation mode.

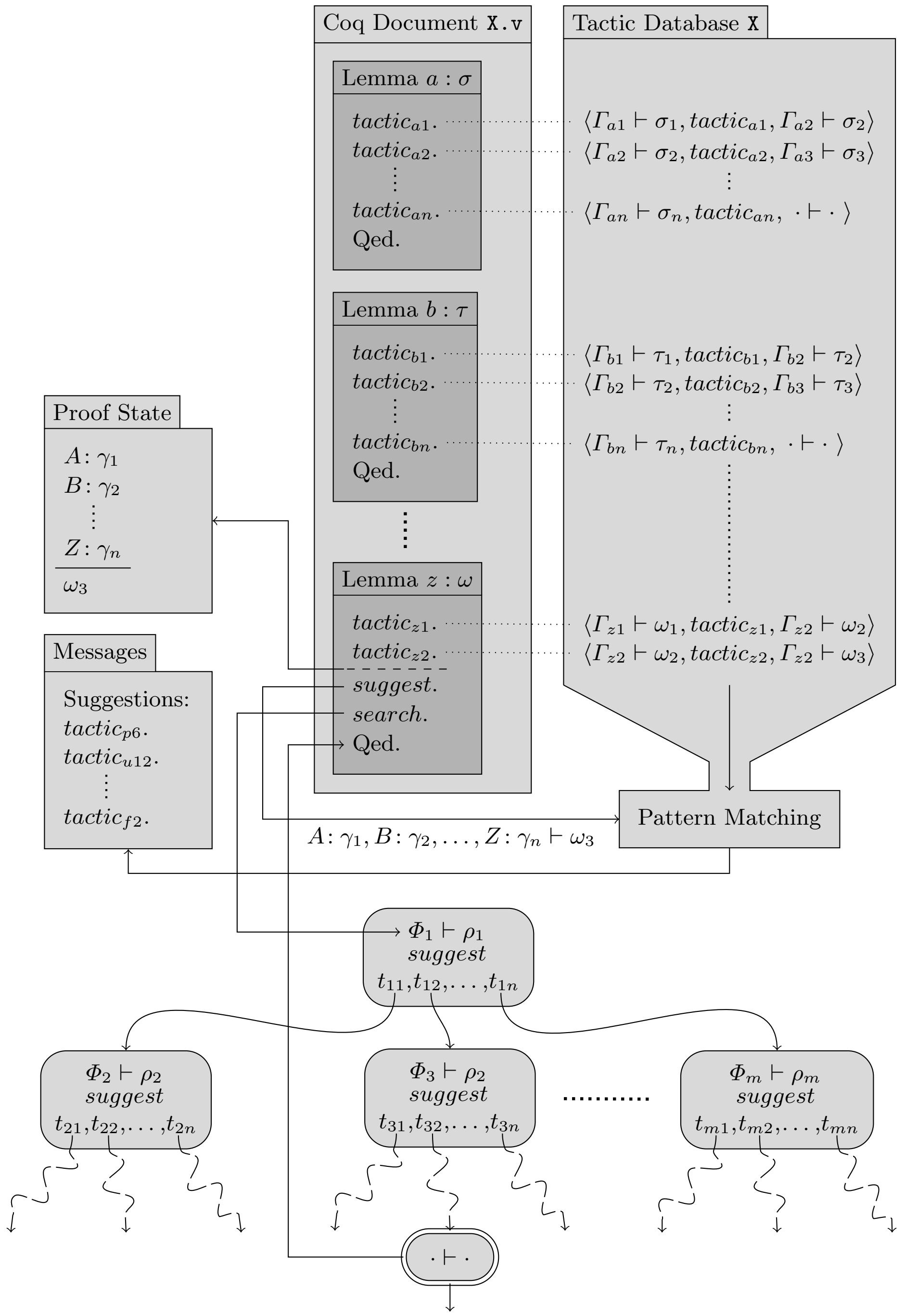
Interactive Mode We illustrate the interactive mode of operation of Tactician using the schematic in Figure 1. When the user starts a new Coq development file—say `X.v`—the first thing Tactician does is create an (in-memory) empty tactic database `X` corresponding to this file. The user then starts to prove lemmas as usual. Behind the scenes, every executed tactic, e.g. tactic_{a1}, is saved into the database accompanied both by the proof states before and after tactic execution, in this case, $\langle \mathit{\Gamma}_{a1} \vdash \sigma_1, \mathit{tactic}_{a1}, \mathit{\Gamma}_{a2} \vdash \sigma_2 \rangle$. The difference between these two states represents the action performed by the tactic, while the state before the tactic represents the context in which it was useful. By recording many such triples for a tactic, we create a dataset representing an approximation of the semantic meaning of that tactic. The database is kept synchronized with the user's movement within the document throughout the entire interactive session.

After proving a few lemmas by hand, the user can start to reap the fruits of the database. For this, the tactics `suggest` and `search` are available. We illustrate their use in the schematic when "Lemma z : ω" is being proven. The user first executes two normal tactics. After that, Coq's proof state window displays a state for which the user is unsure what tactic to use. Here Tactician's tactics come in.

`suggest` This tactic can be executed to ask Tactician for a list of recommendations. The current proof state $A\colon \gamma_1, B\colon \gamma_2, \ldots, Z\colon \gamma_n \vdash \omega_3$ is fed into the pattern matching engine, which will perform a comparison with the states in the tactic database. From this, an ordered list of recommendations is generated and displayed in Coq's messages window, where the user can select a tactic to execute.

`search` Alternatively, the system can be asked to `search` for a complete proof. We start with the current proof state, which we rename to $\mathit{\Phi}_1 \vdash \rho_1$ for clarity. Then a search tree is formed by repeatedly running `suggest` on the proof state and executing the suggested tactics. This tree can be traversed in various ways, finishing only when a complete proof has been found.

If a proof is found, two things happen. (1) The Gallina proof term that is found is immediately submitted to Coq's proof engine, after which the proof can be closed with `Qed`. (2) Tactician generates a reconstruction tactic `search failing` $\langle \mathtt{t}_{12}, \mathtt{t}_{32}, ...\rangle$ which is displayed to the user (see the bottom of the figure). The purpose of this tactic is to provide a modification resilient proof cache. Its semantics is to first try to use the previously found list of tactics $\langle \mathtt{t}_{12}, \mathtt{t}_{32}, ...\rangle$ to complete the proof immediately. "Failing" that (presumably due to changes in definitions or lemmas), a new search is initiated to recover the proof. In order to use the cache, the user should copy it and replace the original `search` invocation with it in the source file.



Coq Document X.v
Tactic Database X
Lemma $a : \sigma$
$tactic_{a1}$.
$tactic_{a2}$.
$tactic_{an}$.
Qed.
$\langle \Gamma_{a1} \vdash \sigma_1, tactic_{a1}, \Gamma_{a2} \vdash \sigma_2 \rangle$
$\langle \Gamma_{a2} \vdash \sigma_2, tactic_{a2}, \Gamma_{a3} \vdash \sigma_3 \rangle$
$\langle \Gamma_{an} \vdash \sigma_n, tactic_{an}, \cdot \vdash \cdot \rangle$
Lemma $b : \tau$
$tactic_{b1}$.
$tactic_{b2}$.
$tactic_{bn}$.
Qed.
$\langle \Gamma_{b1} \vdash \tau_1, tactic_{b1}, \Gamma_{b2} \vdash \tau_2 \rangle$
$\langle \Gamma_{b2} \vdash \tau_2, tactic_{b2}, \Gamma_{b3} \vdash \tau_3 \rangle$
$\langle \Gamma_{bn} \vdash \tau_n, tactic_{bn}, \cdot \vdash \cdot \rangle$
Proof State
$A\colon \gamma_1$
$B\colon \gamma_2$
$Z\colon \gamma_n$
ω_3
Lemma $z : \omega$
$tactic_{z1}$.
$tactic_{z2}$.
suggest.
search.
Qed.
$\langle \Gamma_{z1} \vdash \omega_1, tactic_{z1}, \Gamma_{z2} \vdash \omega_2 \rangle$
$\langle \Gamma_{z2} \vdash \omega_2, tactic_{z2}, \Gamma_{z2} \vdash \omega_3 \rangle$
Messages
Suggestions:
$tactic_{p6}$.
$tactic_{u12}$.
$tactic_{f2}$.
Pattern Matching
$A\colon \gamma_1, B\colon \gamma_2, \ldots, Z\colon \gamma_n \vdash \omega_3$
$\Phi_1 \vdash \rho_1$ suggest $t_{11}, t_{12}, \ldots, t_{1n}$
$\Phi_2 \vdash \rho_2$ suggest $t_{21}, t_{22}, \ldots, t_{2n}$
$\Phi_3 \vdash \rho_2$ suggest $t_{31}, t_{32}, \ldots, t_{3n}$
$\Phi_m \vdash \rho_m$ suggest $t_{m1}, t_{m2}, \ldots, t_{mn}$
$\cdot \vdash \cdot$
Reconstruction tactic: *search failing* $\langle t_{12}, t_{32}, \ldots \rangle$.

Fig. 1. A schematic overview of Tactician in its interactive mode of operation.

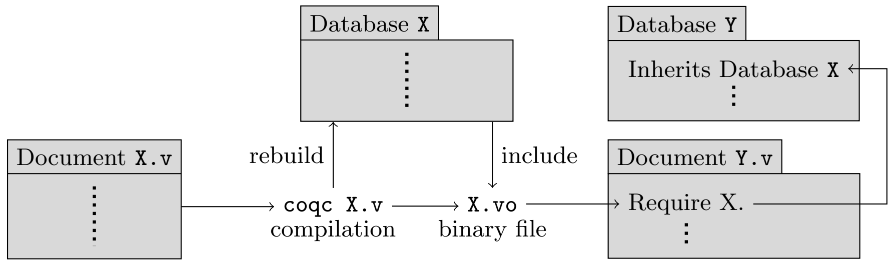


Fig. 2. A schematic overview of Tactician in its compilation mode of operation.

Compilation Mode This mode is visualized in Figure 2. After the file `X.v` has been finished, one might want to depend on it in other files. This requires the file to be compiled into a binary `X.vo` file. The compilation is performed using the command `coqc X.v`. Tactician is integrated into this process. During compilation, the tactic database is rebuilt in the same way as in interactive mode and then included in the `.vo` file. When development `X.v` is then `Require`d by another development file `Y.v`, the tactic database of `X.v` is automatically inherited.

2.3 A Concrete Example

We now give a simple example use-case based on lists. Starting with an empty file, Tactician is immediately ready for action. We proceed as usual by giving a standard inductive definition of lists of numbers with their corresponding notation and a function for concatenation.

```
Inductive list :=
| nil  : list
| cons : nat -> list -> list.
Notation "[]"       := nil.
Notation "x :: ls" := (cons x ls).

Fixpoint concat ls₁ ls₂ :=
  match ls₁ with
  | []       => ls₂
  | x::ls₁' => x::(ls₁' ++ ls₂)
  end where "ls₁ ++ ls₂" := (concat ls₁ ls₂).
```

We wish to prove some standard properties of concatenation. The first is a lemma stating that the empty list `[]` is the right identity of concatenation (the left identity is trivial).

```
Lemma concat_nil_r ls : ls ++ [] = ls.
```

With Tactician installed, we immediately have access to the new tactics `suggest` and `search`. Neither tactic will produce a result when used now since the system has not had a chance to learn from proofs yet. Therefore, we will have to prove this lemma by hand.

```
Proof.
intros. induction ls.
- simpl. reflexivity.
- simpl. f_equal. apply IHls.
Qed.
```

The system has immediately learned from this proof (it was even learning during the proof) and is now ready to help us with a proof of the associativity of concatenation.

```
Lemma concat_assoc ls1 ls2 ls :
  (ls1 ++ ls2) ++ ls3 = ls1 ++ (ls2 ++ ls3).
```

Now, if we execute `suggest`, it outputs the ordered list `intros, simpl, f_equal, reflexivity`. Indeed, using `intros` as our next tactic is not unreasonable. We can repeatedly ask `suggest` for a recommendation after every tactic we input, which sometimes gives us good tactics and sometimes bad tactics. However, we can also eliminate the middle-man and execute the `search` tactic, which immediately finds a proof.

```
Proof. search. Qed.
```

To cache the proof that is found for the future, we can copy-paste the reconstruction tactic that Tactician prints into the source file. This example shows how the system can quickly learn from very little data and with minimal effort from the user. Of course, this also scales to much bigger developments.

We continue our example for more advanced Coq users to showcase how Tactician can learn to use custom domain-specific tactics. We begin by defining an inductive property encoding that one list is a postfix of another.

```
Inductive postfix : list -> list -> Prop :=
| pf_nil : postfix [] []
| pf_cons1 ls1 ls2 n : postfix ls1 ls2 -> postfix ls1 (n::ls2)
| pf_cons2 ls1 ls2 n : postfix ls1 ls2 -> postfix (n::ls1) (n::ls2).
```

We now wish to prove that some lists have the postfix property. For example, `postfix (9::3::[]) (4::7::9::3::[])`. Deciding this is not entirely trivial, because it is not possible to judge from the head of the list whether to apply `pf_cons`$_1$ or `pf_cons`$_2$. Instead of manually writing these proofs, we create a domain-specific, heuristic proving tactic that automatically tries to find a proof.

```
Ltac solve_postfix := solve [match goal with
| |- postfix [] [] => apply pf_nil
| |- postfix (_::_) [] => fail
| |- postfix _ _ =>
       (apply pf_cons1 + apply pf_cons2); solve_postfix
| |- _ => solve [auto]
end].
```

This tactic looks at the current proof state and checks that the goal is of the form `postfix ls`$_1$ `ls`$_2$. If the lists are empty, it can be finished using `pf_nil`.

If ls$_2$ is empty but ls$_1$ nonempty, the postfix is unprovable, and the tactic fails. In all other cases, we initiate a backtracking search where we try to apply either pf_cons$_1$ or pf_cons$_2$ and recurse. Finally, we add a simple catch-all clause that tries to prove any side-conditions using Coq's built-in `auto`. We now teach Tactician about the existence of our previous lemmas and the domain-specific tactic by defining some simple examples. Finally, we ask Tactician to solve a more complicated, compound problem.

```
Lemma ex1 : postfix (9::3::[]) (4::7::9::3::[]).
Proof. solve_postfix. Qed.
Lemma ex2 ls : 1::2::ls ++ [] = 1::2::ls.
Proof. rewrite concat_nil_r. reflexivity. Qed.

Lemma dec2 ls₁ ls₂: postfix ls₁ ls₂ ->
  postfix (7::9::13::ls₁) (8::5::7::[] ++ 9::13::ls₂ ++ []).
Proof. search. Qed.
```

The proof found by Tactician is `rewrite concat_nil_r;intros;solve_postfix`. It has automatically figured out that it needs to introduce the hypothesis, normalize the list and then run the domain-specific prover. This example is somewhat contrived but should illustrate how the user can easily teach Tactician domain-specific knowledge.

2.4 Learning and Proving in the Large

The examples above are fun to play with and useful for demonstration purposes. However, when using Tactician to develop real projects, three main issues need to be taken care of, namely (1) instrumenting dependencies, (2) instrumenting the standard library and, (3) reproducible builds. Below, we describe how to use Tactician in complex projects and, in particular, how these issues are solved.

Tactician itself is a collection of easily installed Opam [11] packages distributed through the Coq Package Index [25]. The package `coq-tactician` provides the main functionality. It needs to be installed to run the examples of Section 2.3. These examples have no dependencies and make minimal use of Coq's standard library. All learning is done within one file, making them a simple use-case. Things become more complicated when one starts to use the standard library and libraries defined in external dependencies. Although Tactician will keep working normally in those situations, by default, it does not learn from proofs in these libraries. Hence, Tactician's ability to synthesize proofs for lemmas concerning the domain of those libraries will be severely limited.

The main question to remedy this situation is where Tactician should get its learning data from. As explained in Section 2.2, Tactician saves the tactic database of a library in the compiled `.vo` binaries. This database becomes available as soon as the compiled library is loaded. However, this only works if the library was compiled with Tactician enabled, which is the case neither for Coq's standard library nor most external packages. Hence, we need to instrument these libraries by recompiling them while Tactician is loaded. Loading Tactician amounts to finding a way of convincing Coq to pre-load the library `Tactician.Ltac1.Record` before starting the compilation process.

External Dependency Instrumentation An external dependency can be any collection of Coq source files together with an arbitrary build system. Injecting the loading of `Tactician.Ltac1.Record` into the build process automatically is not possible in general. However, we do provide a simple solution for the most common situation. Coq provides package developers with a utility called `coq_makefile` that automatically generates a Makefile to build and install their files. This build system is usually packaged up with Opam to be released on the Coq Package Index. Although this package index does not require packages to use `coq_makefile`, most do this in practice.

Makefiles generated by `coq_makefile` are highly customizable through environment variables. Tactician provides command-line utilities called `tactician enable` and `tactician disable` that configure Opam to automatically inject Tactician through these environment variables. When building packages without Opam, the user can modify the environment by running `eval $(tactician inject)` before building. This solution will suffice to instrument most packages created using `coq_makefile`, as long as authors do not customize the resulting build file too heavily. We will add support for Coq's new Dune build system [15] when it has stabilized. For more stubborn packages, rather aggressive methods of injecting Tactician are also possible but, in general, packages that circumvent instrumentation are always possible. Therefore, we do not provide built-in solutions for those cases.

Standard Library Instrumentation In order to instrument Coq's standard library, it also needs to be recompiled with `Tactician.Ltac1.Record` pre-loaded. We provide the Opam package `coq-tactician-stdlib` for this purpose. This package does not contain any code, but simply takes the source files of the installed standard library and recompiles them. It then promptly commits the Cardinal Sin of Package Management by overwriting the original binary `.vo` files of the standard library. We defend this choice by noting that (1) the original files will be backed up and restored when `coq-tactician-stdlib` is removed and (2) the alternative of installing the recompiled standard library in a secondary location is even worse. This choice would cause a rift in the users local ecosystem of packages, with some packages relying on the original standard library and some on the recompiled one. Coq will refuse to load two packages from the rivaling ecosystems citing "incompatible assumptions over the standard library," forever setting them apart.

Even with our choice of overwriting the standard library, an ecosystem rift still occurs if packages depending on Coq already pre-existed. To resolve this, Tactician ships with the command-line utility `tactician recompile` that helps the user find and recompile these packages.

Tactician Usage within Packages In order to use Tactician's `suggest` and `search` tactics, the library `Tactician.Ltac1.Tactics` needs to be loaded. However, we strongly advise against loading this library directly, for two reasons. (1) If a development `X` that uses Tactician is submitted to the Coq Package Index as `coq-x`, an explicit dependency on `coq-tactician` is needed. This dependency can be undesirable due to users potentially being unwilling to install Tactician.

(2) It would undermine the build reproducibility of the package. Even though `coq-tactician` would be installed as a dependency when `coq-x` is installed, there is no way to ensure that Tactician has instrumented the other dependencies of the package. Hence, it is likely that Tactician will be operating with a smaller tactic database, reducing its ability to prove lemmas automatically.

Instead, the package `coq-x` should depend on `coq-tactician-dummy`. This package is extremely simple, containing one 30-line library called `Tactician.Ltac1Dummy`. It provides alternative versions of Tactician tactics that act as dummies of the real version. Tactics `suggest` and `search` will not perform any action. However, tactic `search failing` $\langle \texttt{...} \rangle$, described in Section 2.2, will still be able to complete a proof using its cache (but without the ability to search for new proofs in case of failure). A released package can thus only employ cached searches. This way, any build will be reproducible.

During development, the real version of Tactician should be loaded to gain its full power. Instead of loading it explicitly through a `Require` in source files, we recommend that users load it through the `coqrc` file. Coq will automatically process any vernacular defined in this file at startup. The command-line utility `tactician enable` will assist in adding the correct vernacular to the `coqrc` file.

3 Technical Implementation

In this section, we provide a peek behind the curtains of Tactician's technical implementation and how it is integrated with Coq. A previous publication already covers the following aspects of Tactician [3]: (1) The machine learning models used to suggest tactics; (2) an explanation of how data extracted from Coq is decomposed and transformed for these models; and (3) the search procedure to synthesize new proofs. These details are therefore omitted here.

3.1 Intercepting Tactics

Tactician is implemented as a plugin that provides a new proof mode (tactic language) to Coq. This proof mode contains precisely the same syntactical elements as the Ltac1 tactic language [8]. The purpose of the proof mode is to intercept and decompose executed tactics and save them in Tactician's database. After interception, the tactics are redirected back to the regular Ltac1 engine. By loading the library `Tactician.Ltac1.Record`, this proof mode is activated instead of the regular Ltac1 language.

Note that Ltac1 is the most popular but by no means the only tactic language for Coq [16,21,12,23]. All these languages are compiled into a proof monad implemented on the OCaml level [19]. It would be preferable to instrument the proof monad directly as this would enable us to record tactics from all languages at once. It appears that this is impossible, though, because the structure of the monadic interface does not allow us to recover high-level tactical units such as decision procedures, even when implemented as the most general Free Monad. As such, Tactician only supports Ltac1 at the moment. In the future, we intent to provide improved support for SSreflect [12] and support for recording the new Ltac2 language [23].

3.2 State Synchronization

When recording tactics in interactive mode, it is important to synchronize the tactic database with the undo/redo actions of the user, both from a theoretical and practical perspective. In theory, if a user undoes a proof step, this represents a mistake made by the user, meaning that the recorded information in the database is also a mistake. In practice, keeping such information will lead to problems in compilation mode because the database will be smaller due to the lack of undo/redo actions. Therefore `search`es that succeeded in interactive mode may not succeed in compilation mode. Below, we explain how Coq and Tactician deal state synchronization.

Internally, Coq ships with a state manager that keeps track of all state information generated when vernacular commands are executed. This information includes, for example, definitions, proofs, and custom tactic definitions. This data is automatically synchronized with the user's interactive movement through the document, and saved to the binary `.vo` file during compilation. All data structures registered with the state manager are expected to be persistent (in the functional programming sense [9]). The copy-on-write semantics of such data structures allow the state manager to easily keep a history of previous versions and revert to them on demand.

For Tactician, registering data structures with the state manager to ensure proper synchronization is awkward, because the state manager assumes that tactics have no side-effects outside of modifications to the proof state. Hence, any data registered with the state manager is discarded as soon as the current proof has been finished. Tactician solves this by tricking Coq into thinking that tactics are side-effecting vernacular commands, convincing it to re-execute all tactics at `Qed` time to properly register the side-effects. However, as a consequence, these tactics will modify the proof state a second time, at a time when this is not intended. This is a likely source of future bugs for which a permanent solution is yet to be found.

4 Tactician as a Machine Learning Platform

Apart from serving as a tool for end-users, Tactician also functions as a machine learning platform. A simple OCaml interface to add a new learning model to Tactician is provided. The learning task of the model is to predict a list of tactics for a given proof state. When registering a new model, Tactician will automatically take advantage of it during proof search. Our interface hides Coq's internal complexities while being as general as possible. We encourage everyone to implement their favorite learning technique and try to beat the built-in model. Tactician's performance can easily be benchmarked on the standard library and other packages. The signature of a machine learning model is as follows:

```
type sentence = Node of string * sentence list
type proof_state =
{ hypotheses : (id * sentence) list
; goal       : sentence }
```

```
type tactic
val tactic_sentence : tactic -> sentence
val local_variables : tactic -> id list
val substitute      : tactic -> id_map -> tactic

module type TacticianModelType = sig
  type t
  val create : unit -> t
  val add : t ->  before:proof_state -> tactic -> after:proof_state -> t
  val predict : t -> proof_state -> (float * tactic) list
end
val register_learner : string -> (module TacticianLearnerType) -> unit
```

A `sentence` is a very general tree data type able to encode all of Coq's internal syntax trees, such as those of terms and tactics. Node names of syntax trees are converted into `string`s. This way, most semantic information is preserved using a much simpler data type that is suitable for most machine learning techniques. Proof states are encoded as a list of named hypothesis sentences and a goal sentence. In this case, sentences represent a Gallina term. We abstract from some of Coq's proof state complexities such as the shelf and the unification map.

Tactician represents `tactics` as an abstract type that can be inspected as a sentence using `tactic_sentence`. Since the goal of this interface is to *predict* tactics but not *synthesize* tactics, it is not possible to modify them (this would seriously complicate the interface). There is one exception. We provide a way to extract a list of variables that refer to the local context of a proof. The local variables of a tactic can also be updated using a simultaneous substitution. Such substitutions will allow for a limited form of parameter prediction.

We think that local variable prediction is the only kind of parameter prediction that makes sense in Tactician's context. The only other major classes of parameters are global lemma names and complete Gallina terms. Predicting complete terms is known to be very difficult and would unnecessarily complicate the interface. Predicting names of global lemmas is possible, but does not appear to be very useful because lemma names are almost always directly associated with basic tactics like `apply lem` or `rewrite lem`. Predicting parameters for these tactics is counter-productive because their semantics are mostly dependent on the definition of the lemma. Hence, it is better to view the incarnations of such tactics with different lemmas as arguments as completely separate tactics.

Finally, implementing a learning model entails implementing the module type `TacticianModelType` and registering it with Tactician. This module requires an implementation of a database type `t`. For reasons explained in Section 3.2, this database needs to be persistent. Tactician will `add tactics` to the database, together with the `proof_state` before and after the tactic was applied. The machine learning task of the model is to `predict` a weighted list of tactics that are likely applicable to a previously unseen proof state.

The current interface only allows for models that support online learning because database entries are `add`ed one by one in interactive mode. We justify this by the user-friendliness requirements from Section 2.1. However, we realize that together with the persistence requirement, this places considerable limitations on the kind of learning models that can be employed. In the future, we intent to support a secondary interface that can be used to create offline models employing batch learning on large Coq packages in its entirety.

5 Case Study

The overall performance of our tactical search on the full Coq Standard Library is reported in a previous publication [3], which also reports performance on various parts of the library. The best-performing version of our learning model can prove 34.0% of the library lemmas when using a 40s time limit. Six different versions together prove 39.3%. The union with all CoqHammer methods achieves 56.7%.[3]

Here we show an example of a nontrivial proof found by Tactician. The system was asked to automatically find the proof of the following lemma from the library file `Structures/GenericMinMax.v`,[4] where facts about general definitions of `min` and `max` are proved.

```
Lemma max_min_antimono f :
 Proper (eq==>eq) f -> Proper (le==>flip le) f ->
 forall x y, max (f x) (f y) == f (min x y).
```

Tactician's learning model evaluated the following two lemmas as similar to what has to be proven:

```
Lemma min_mono f :
   (Proper (eq ==> eq) f) -> (Proper (le ==> le) f) ->
   forall x y, min (f x) (f y) == f (min x y).
 intros Eqf Lef x y.
 destruct (min_spec x y) as [(H,E)|(H,E)]; rewrite E;
  destruct (min_spec (f x) (f y)) as [(H',E')|(H',E')]; auto.
 - assert (f x <= f y) by (apply Lef; order). order.
 - assert (f y <= f x) by (apply Lef; order). order.
Qed.

Lemma min_max_modular n m p :
   min n (max m (min n p)) == max (min n m) (min n p).
 intros. rewrite <- min_max_distr.
 destruct (min_spec n p) as [(C,E)|(C,E)]; rewrite E; auto with *.
 destruct (max_spec m n) as [(C',E')|(C',E')]; rewrite E'.
 - rewrite 2 min_l; try order. rewrite max_le_iff; right; order.
 - rewrite 2 min_l; try order. rewrite max_le_iff; auto.
Qed.
```

The trace through the proof search tree that resulted in a proof is as follows:

```
max_min_antimono  .0.0.0.5.5.2.1.0.5.1.5.1
```

[3] CoqHammer's eight methods prove together 40.8%, with the best proving 28.8%.
[4] https://coq.inria.fr/library/Coq.Structures.GenericMinMax.html

This trace represents, for every choice point in the search tree, which of `suggest`'s ranked suggestion was used to reach the proof. The proof search went into depth 12 and the first three tactics used in the final proof are those with the highest score as recommended by the learning model, which most likely followed the proof of `min_mono`. However, after that, it had to diverge from that proof, using only the sixth-best ranked tactic twice in a row. This nontrivial search continued for the next seven tactical steps, combining mostly tactics used in the two lemmas and some other tactics. The search finally yielded the following proof of `max_min_antimono`.

```
intros Eqf Lef x y. destruct (min_spec x y) as [(H, E)|(H, E)]. rewrite E.
destruct (max_spec (f x) (f y)) as [(H', E')| (H', E')].
assert (f y <= f x) by (apply Lef; order). order. auto. rewrite E.
destruct (max_spec (f x) (f y)) as [(H', E')| (H', E')]. auto.
assert (f x <= f y) by (apply Lef; order). order.
```

Note that the original proof of the lemma is quite similar, but shorter and without some redundant steps. Redundant steps are known to happen in systems like Tactician, such as the TacticToe [10] system for HOL4 [24].

```
intros Eqf Lef x y. destruct (min_spec x y) as [(H,E)|(H,E)]; rewrite E;
destruct (max_spec (f x) (f y)) as [(H',E')|(H',E')]; auto.
- assert (f y <= f x) by (apply Lef; order). order.
- assert (f x <= f y) by (apply Lef; order). order.
```

6 Related Work

There exist quite a few machine learning systems for Coq and other interactive theorem provers. The most significant distinguishing factor of Tactician to other systems for Coq is its user-friendliness. There are several other systems that are interesting, but rather challenging to install and use for end-users. They often depend on external tools such as machine learning toolkits and automatic theorem provers. Some systems need a long time to train their machine learning models—preferably on dedicated hardware. Those are often not geared towards end-users at all but rather towards the Artificial Intelligence community.

Tactician takes its main inspiration from the TacticToe [10] system for HOL4 [24] which learns tactics expressed in the Standard ML language. Using this knowledge, it can then automatically search for proofs by predicting tactics and their arguments. Our work is similar both in doing a learning-guided tactic search and by its complete integration in the underlying proof assistant without relying on external machine learning libraries and specialized hardware.

Below is a short list of machine learning systems for the Coq theorem prover.

ML4PG provides tactic suggestions by clustering together various statistics extracted from interactive proofs [20]. It has integrated with the Proof General [2] proof editor and requires connections to Matlab or Weka.

SEPIA provides proof search using tactic predictions and is also integrated with Proof General [13]. Note, however, that its proof search is only based on tactic traces and does not make predictions based on the proof state.

Gamepad is a framework that integrates with the Coq codebase and allows machine learning to be performed in Python [14]. It uses recurrent neural networks to make tactic prediction and to evaluate the quality of a proof state. The system is able to synthesize proofs in the domain of algebraic rewriting. Gamepad is not geared towards end-users.

CoqGym extracts tactic and proof state information on a large scale and uses it to construct a deep learning model capable of generating full tactic scripts [26]. CoqGym's evaluation is using a time limit of 600s, which is impractically high for Coq practitioners. Still, it is significantly weaker than CoqHammer. A probable cause is the slowness of deep neural networks which is common to most proving experiments geared towards the deep learning community.

CoqHammer is a machine learning and proving tool in the general *hammers* [5,17,4,18] category designed for Coq [7]. Hammers capitalize on the capabilities of automatic theorem provers to assist ITP's. To this end, learning-based premise selection is used to translate an ITP problem to the ATP's target language (typically First Order Logic). A proof found by the ATP can then be reconstructed in the proof assistant. CoqHammer is a maintained system that is well-integrated into Coq and only requires a connection to an ATP. It has similar performance as Tactician but proves different lemmas, making these systems complementary [3].

7 Further Work and Conclusion

We have presented Tactician, a seamless and interactive tactic learner and prover for Coq. The machine learning perspective has been described in a previous publication [3]. We showed how Tactician is an easy-to-use tool that can be employed by the user with minimal installation effort. A clear approach to using the system in large developments has also been outlined. With its current machine learning capabilities, we expect Tactician to help the user with its proving efforts significantly. Finally, we presented a powerful machine learning interface that will allow researchers to bring their advanced learning models to Coq users while being isolated from Coq's internal complexities. We expect this to be of considerable utility to both the artificial intelligence community and Coq users.

There are many future directions. There is a never-ending quest to improve the built-in learning model. With better features and stronger (but still fast) learners such as boosted trees (used in ENIGMA-NG [6]) we hope to push Tactician's performance over the standard library towards 50%. Apart from this, we expect to improve support for SSreflect and to introduce support for the new Ltac2 language. In the future, the machine learning interface will be expanded to allow for batch learning. Additionally, we would like to incorporate the tactic history (*memory*) of the current lemma into the learning model, similar to SEPIA. Memory will allow Tactician to predict which tactics are often used together.

References

1. The Coq proof assistant, version 8.11.0
2. Aspinall, D.: Proof General: A generic tool for proof development. In: TACAS. Lecture Notes in Computer Science, vol. 1785, pp. 38–42. Springer

3. Blaauwbroek, L., Urban, J., Geuvers, H.: Tactic learning and proving for the Coq proof assistant. CoRR **2003.09140** (2020)
4. Blanchette, J.C., Greenaway, D., Kaliszyk, C., Kühlwein, D., Urban, J.: A learning-based fact selector for Isabelle/HOL. J. Autom. Reasoning **57**(3), 219–244
5. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Form. Reasoning **9**(1), 101–148
6. Chvalovský, K., Jakubuv, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: CADE 27. pp. 197–215 (2019)
7. Czajka, L., Kaliszyk, C.: Hammer for Coq: Automation for dependent type theory. J. Aut. Reasoning **61**(1-4), 423–453
8. Delahaye, D.: A tactic language for the system Coq. In: LPAR. Lecture Notes in Computer Science, vol. 1955, pp. 85–95. Springer
9. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. **38**(1), 86–124
10. Gauthier, T., Kaliszyk, C., Urban, J.: TacticToe: Learning to reason with HOL4 tactics. In: LPAR. EPiC Series in Computing, vol. 46, pp. 125–143. EasyChair
11. Gazagnaire, T., Fessant, F.L., Minsky, Y., Madhavapeddy, A., Tuong, F.: Ocaml package manager
12. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. J. Form. Reasoning **3**(2), 95–152
13. Gransden, T., Walkinshaw, N., Raman, R.: SEPIA: search for proofs using inferred automata. In: CADE. Lecture Notes in Computer Science, vol. 9195, pp. 246–255. Springer
14. Huang, D., Dhariwal, P., Song, D., Sutskever, I.: Gamepad: A learning environment for theorem proving. In: ICLR (Poster). OpenReview.net
15. Jane Street: Dune: A composable build system for OCaml, `https://dune.build`
16. Kaiser, J., Ziliani, B., Krebbers, R., Régis-Gianas, Y., Dreyer, D.: Mtac2: typed tactics for backward reasoning in Coq. PACMPL **2**(ICFP), 78:1–78:31
17. Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. J. Aut. Reasoning **53**(2), 173–213
18. Kaliszyk, C., Urban, J.: MizAR 40 for Mizar 40. J. Autom. Reasoning **55**(3), 245–256
19. Kirchner, F., Muñoz, C.A.: The proof monad. J. Log. Algebr. Program. **79**(3-5), 264–277
20. Komendantskaya, E., Heras, J., Grov, G.: Machine learning in Proof General: Interfacing interfaces. In: UITP. EPTCS, vol. 118, pp. 15–41
21. Malecha, G., Bengtson, J.: Extensible and efficient automation through reflective tactics. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 532–559. Springer
22. Paulin-Mohring, C.: Inductive definitions in the system Coq - rules and properties. In: TLCA. Lecture Notes in Computer Science, vol. 664, pp. 328–345. Springer
23. Pédrot, P.M.: Ltac2: Tactical warfare. In: CoqPL (2019)
24. Slind, K., Norrish, M.: A brief overview of HOL4. In: TPHOLs. Lecture Notes in Computer Science, vol. 5170, pp. 28–32. Springer
25. The Coq Development Team: Coq package index, `https://coq.inria.fr/opam/www`
26. Yang, K., Deng, J.: Learning to prove theorems via interacting with proof assistants. In: ICML. Proceedings of Machine Learning Research, vol. 97, pp. 6984–6994. PMLR