Improvement of the automation in the `coq-waterproof` library

M1 intership report

Balthazar Patiachvili

10/04/2023 – 23/06/2023

Under the supervision of Jim Portegies

Abstract

Improving logical reasoning is an essential skill that all science students must acquire. To aid in this learning process, some universities have started using proof assistants. However, the syntax of these assistants can be unclear and learning how to use them effectively poses a significant challenge, limiting their utility. Addressing this issue, we present the `coq-waterproof` library, a Coq plugin designed to write proofs in natural language, resembling pen-and-paper style as closely as possible. To streamline the proof development process and avoid wasting time on uninteresting aspects, we improved the existing automation system based on Coq's `auto` tactic. Our primary focus lies on this specific aspect: enhancing automation capabilities in a general sense. Two main components have been scrutinized in detail. First, we have introduced a proof control mechanism during proof searches, allowing for proofs' rejection based on arbitrary criteria. Secondly, we have added automated rewriting steps between proof searches, leveraging current hypotheses, reinforcing the fluidity of the proofs written. These advancements required the development of an extended version of the tactic monad from Coq's OCaml API. Additionally, we have implemented an optimization technique enabling efficient cuts during proof searches, thus reducing the execution time of the automation routine. Overall, this work lays a foundation for further improvements. Future research should build upon our findings, focusing on making the `coq-waterproof` tool more accessible and user-friendly. Bridging the gap between proofs generated by `coq-waterproof` and these crafted manually on paper remains a key goal.

Contents

Internship context

The present document is a report for my M1 short internship that is part of my curriculum at the ENS Paris-Saclay. This internship took place at the CASA team of the Eindhoven University of Technology in the Netherlands under the supervision of Jim Portegies.

I would like to thank Jim Portegies, Jelle Wemmenhove as well as the CASA team for their warm welcome, for the nice atmosphere in which I spent three months and for their sympathy.

Note: As I chose to do a short internship and had to start it before my exams, I left the internship one week to take four exams at the ENS Paris.

1 Introduction

1.1 Context

Acquiring the skill of performing logically coherent reasoning is a vital component during the initial years of a bachelor's degree in science. However, this task can be challenging for students who may encounter difficulties when it comes to writing mathematical proofs. Whether it's the intricacies of structuring a proof or the uncertainty surrounding the validity of an argument, issues can arise at various stages of the proving process.

To facilitate the acquisition of logical reasoning skills, several universities have incorporated proof assistants such as Coq or Lean into their curriculum. Proof assistants, also known as interactive theorem provers, are software tools designed to aid humans in constructing formal proofs. The benefits of utilizing these tools vary depending on the specific proof assistant chosen, but they all share a common feature: ensuring the validity of every written proof. In particular, Coq, the proof assistant utilized during this internship, offers users the ability to track the progress of their proofs at each step, providing immediate feedback on their actions and aiding their understanding of the current state of the proof. This real-time feedback mechanism proves invaluable in guiding students through the process of constructing rigorous and logically coherent arguments. The article [6] shows this point with an experiment where students had to use Coq as a learning tool in an introductory course on logic and data structures. The results show that Coq can be beneficial for undergraduate students in their learning process of formal proofs.

Nevertheless, the ability to write formal proofs in Coq does not always lead to the ability to write paper proofs. Indeed, the same study [6] found that some students using Coq had the worst score at writing proofs in natural language. On the counter part, another study [7] involving students learning Lean and not Coq shows that the students who had attended Lean voluntary classes had better grades in the final examination than the group of students that did not attend these classes regularly. This enlightens the fact that courses using proof assistants are not widespread, and further studies should be done to have a better idea on their potential utility.

Moreover, their use can be arduous for inexperienced users. Indeed, most undergraduate mathematic students have little to zero knowledge about the proof assistants' syntax which can be hard to learn from scratch because the structure and a complete proof can be unclear and confusing.

1.2 Waterproof

To address the issues described above, a team of members of the Eindhoven University of Technology (TU/e) created *Waterproof* [1], a software for educational purpose, based on `coq-waterproof`, a Coq library written in Ltac2, allowing users to make proofs in natural language. The proofs can be constructed with predefined english sentences, making it look like a real pen-and-paper proof. It is expected that by using this library as a complement during the learning process of making sound proofs, the skills obtained with the manipulation of Waterproof will transfer to the handwritten proofs. The figures 1 and 2 present two simple proofs for the same property, one only using Coq, and the other one using the `coq-waterproof` library, showing the focus on the accessibility for non-expert users.

```
Goal forall n m: nat, n = 0 -> S m <> n.
Proof.
  intros n m H H'.
  rewrite H in H'.
  inversion H'.
Qed.
```

Figure 1: Example of a Coq proof of $\forall n, m \in \mathbb{N}, n = 0 \Longrightarrow m + 1 \neq n$

```
Goal forall n m: nat, n = 0 -> S m <> n.
Proof.
  Take n, m: nat.
  Assume that (n = 0).
  We conclude that (S m <> n).
Qed.
```

Figure 2: Example of a `coq-waterproof` proof of $\forall n, m \in \mathbb{N}, n = 0 \Longrightarrow m + 1 \neq n$

Waterproof has been used for several years as an optional complement for some mathematical analysis courses at the TU/e. During the year 2021-2022, while some students that tried Waterproof dropped its use before the final assignment, a non insignifiant part of the class kept it until the end [1], showing that this type of project can really be useful.

1.3 Internship

In this context, this internship focused on the development of Waterproof's automation system called *waterprove*. Indeed, to be interesting as a pedagogical tool, students should focus on proving something that is non-trivial for humans (e.g $\forall \varepsilon \in \mathbb{R}, \varepsilon > 0 \Longrightarrow \frac{\varepsilon}{2} > 0$). This is why very powerful tactics are used, such as `lra` or `nra` that are decision procedure for respectively linear and non-linear real or rational arithmetic. However, having the most powerful automatic prover is not either something that we want to reach: if every lemma/theorem can be proved automatically, there is no point in using this software to learn how to make sound proofs.

Thus, the two main axes of research during this internship were to give more control on the automatic search for a proof to be able to reject an unwanted proof (section 3) and to make the automation system more powerful, here by considering rewrites using equalities in the hypothesis (section 4). Furthermore, these new implementations made the execution time greatly increased, hence it was needed to optimize the existing algorithms (section 5). Finally,

to implement these new algorithms into Coq, it was needed to consider the OCaml's tactic system (section 2.2) and to expand it to be able to create typed tactics.

1.4 Tools used

This project is written in Ltac2, in Coq 8.17.0, to which was added OCaml 4.14.1 during this internship. The source code of the project is available on GitHub at: https://github.com/impermeable/coq-waterproof. My repository, on which the development was made, is available at: https://github.com/BalthazarPatiachvili/coq-waterproof. The OCaml API with comments of my repository is available at: https://balthazarpatiachvili.github.io/coq-waterproof/ocaml/.

Before this internship, this library was fully written in Ltac2. However, for various reasons explained later in this report, it was needed to use OCaml pieces of code: a long work has been made to transfer the existing library into a Coq plugin, mainly because of the huge lack of documentation and explanation in the Coq's source code (available in [3]).

2 Coq proofs

In this section, OCaml will be used several times. Thus, `'a`, `'b`, ... represent types such that `'a` means "for all types `'a`".

2.1 Proof state

In Coq, two modes are available: the default environment to, among others, code in Gallina, a functional programming language, and the proof environment to build proofs with tactics (see 2.2) called the *proof mode* [2]. It is possible to enter the proof mode by declaring a new goal, lemma or theorem with respectively `Goal`, `Lemma` or `Theorem`. The proof mode ends with either `Qed` to indicate that the proof is finished and complete (what had to be demonstrated is proved), `Abort` to end the proof before its end or `Admitted` to admit that the property is true without proving it entirely.

The *proof state* is a set of one or several unproved goals. Each goal contains a conclusion, a *local context* with the current local named hypothesis, variables and definitions. It is also possible to use definitions, axioms, proved theorems, ... defined beforehand in the *global environment*. Tactics, more detailed in 2.2, change the current proof state.

```
1 Goal forall n m: nat, n = 0 -> S m <> n.
2 Proof.
3   intros n m H H'.
4   rewrite H in H'.
5   inversion H'.
6 Qed.
```

Figure 3: Example of a proof of $\forall n, m \in \mathbb{N}, n = 0 \implies m + 1 \neq n$ in Coq

For example, the figure 3 shows a simple proof in Coq:

- at the beginning of the proof, there is no hypothesis and the only conclusion is the one declared as the main goal: `forall n m: nat, n = 0 -> S m <> n`;

- after the tactic `intros n m H H'.`, the conclusion became `False`, two variables and two hypothesis have been declared, namely `n: nat` and `m: nat`, and `H: n = 0` and `H': S m = n`;
- after the tactic `rewrite H in H'.`, only the hypothesis `H'` changed, becoming `H': S m = 0`;
- the last tactic, `inversion H'.` concluded the goal, finishing to prove the only conclusion.

2.2 Tactics

2.2.1 General description

Tactics are at the core of Coq's proof mechanism [2]. They transform the current proof state of an incomplete proof in order to make the proof complete. Tactics are more specifically special functions that will manipulate the proofs by either proving one or more goals – or more generally by replacing the current goal with one or more new ones – or by adding or modifying one of the hypothesis.

It is now important to make a distinction between the Coq tactics and the OCaml ones. Indeed, while Coq tactics represent a full operation on the current proof environment, like introducing a new variable bounded by a ∀-quantifier with `intros` or rewriting a subterm with an equal subterm with `rewrite`, OCaml tactics only consist in small monadic operations (see 2.2.2). In this report, whenever it is not specified, it is assumed that "tactic" is referring to an OCaml tactic and not a Coq one.

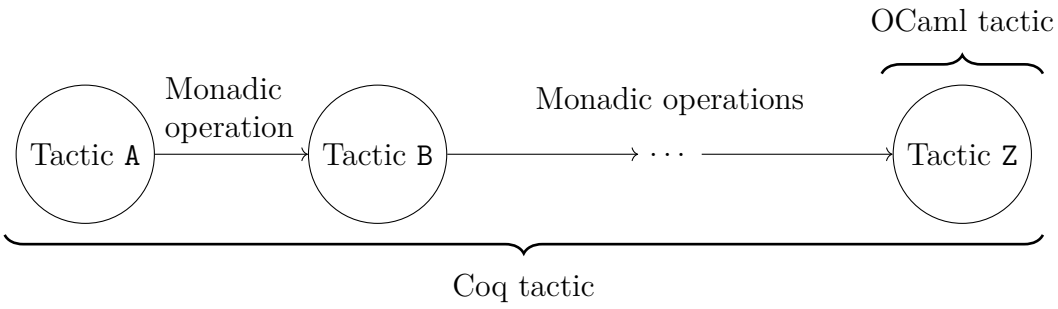


Figure 4: Simplified representation of the tactic system

Coq tactics offer a high level of abstraction, which is really useful when trying to create a proof. Moreover, to create new tactics the Ltac and Ltac2 languages allow the user to easily build their own on the top of Coq tactics. This is why before this internship, all the `coq-waterproof` library was written in Ltac2. However, to have more expressivity in tactics' construction, it is needed to use Coq's OCaml interface: Ltac2 does not provide enough structures and flow control to be able to build a tactic that is not just a juxtaposition of others.

2.2.2 OCaml's `tactic` monad

A *monad* is a structure combining functions and wrapping their return values in a type enriched by additional computation. A monad can be created by the two following functions (with `'a t` the monadic type wrapping the type `'a`) :

```
val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

`return` is used to create a monadic value of type `'a t` from a value of type `'a`, and `bind` will try to transform a wrapped value of type `'a t` in a wrapped value of type `'b t` by the given function.

Figure 5: Representation of monadic functions `return` and `bind`

In OCaml, tactics are represented as monadic objects, defined in the file `engine/proofview.ml` of the Coq's source code [3]. For a given type `'a`, the corresponding monadic type is `'a tactic`, and the functions `tclUNIT` and `tclBIND`, respectively equivalent to `return` and `bind`, are available, allowing to create all the necessary tactics.

Now, let's give the intuition behind this monad: each Coq tactic corresponds to a `unit tactic`. Therefore, a tactic can be considered as a function that will return a value that can be used in the definition of another tactic thanks to the `tclBIND` function. With this intuition, Coq tactics are functions modifying the environment while returning a `unit` value. Thus, it is possible to use all the functions defined around the tactic monad, we just have to ensure that the final output has the right type, and all the intermediary tactics can have any wanted type. However, as the vast majority of tactics in Coq's source code (so in OCaml) have a type of `unit tactic`, making a lot of useful function to only support `unit tactic` and not `'a tactic`. The need of tactics with a different type from `unit tactic`, which will be called *typed tactics*[1] in this report, is explained in the section 3.

2.2.3 Typed tactics (contribution)

Some functions from the Coq's OCaml API using tactics were needed during the implementation of the algorithms presented in 3. However, to make these algorithms work, it is important to get non-unit values as tactics' output, and these functions could only deal with `unit tactic`. This idea has been found after reading the article [4] presenting Mtac2, a Coq plugin implementing typed tactics for backward reasoning. The introduced features are very similar to the ones wanted (that are described below), nevertheless, this is not usable in our context because we needed the proof flow control offered by OCaml, which is more powerful than the one of Coq.

The figure 6 shows examples of required functions from the `Proofview` module of Coq's OCaml API that were not general enough. During the major part of the internship, these functions were developed only for the needed types: for example, with `trace` a type used in the section 3, the following functions were defined :

[1]Even if all tactics are technically typed, we borrow the name *typed tactics* in opposition to `unit tactic` from the Mtac2 Coq library [4].

Function name	Existing version	General version
`Goal.enter`	`(Goal.t -> unit tactic) -> unit tactic`	`(Goal.t -> 'a tactic) -> 'a tactic`
`tclTHEN`	`unit tactic -> 'a tactic -> 'a tactic`	`'a tactic -> 'a tactic -> 'a tactic`
`tclINDEPENDENT`	`unit tactic -> unit tactic`	`'a tactic -> 'a tactic`

Figure 6: Examples of required functions

```
val trace_goal_enter : (Goal.t -> trace tactic) -> trace tactic
val tclTraceThen : trace tactic -> trace tactic -> trace tactic
val tclTraceIndependent : trace tactic -> trace tactic
```

However, a more general interface could be created under some constraints. Indeed, with the example of `tclTHEN` (see figure 6) that will apply two tactics successively, to be able to define a new version with a given type it is needed to know how to merge the outputs of the two tactics. I noticed that in a more general approach, the vast majority of the tactics defined in the modules `Proofview` and `Tacticals` could be generalized to any type `x` under two constraints: the type can be merged, which is represented by a function of type `x -> x -> x`, and the existence of an empty element of type `x`.

An implementation has been made with a *functor*, a module parametrized by another module. The signature of the module that a type needs to implement to make generalized tactics is the following:

```
module type Mergeable = sig
    type elt
    val empty : elt
    val merge : elt -> elt -> elt
end
```

With this interface, it is possible to define almost all the generalized versions of functions in the modules `Proofview` and `Tacticals` (see the full implementation in annex A). The complete interface has been used both in the algorithms described in sections 3 and 4.

3 Automation control

The first axis of research to improve the automation is to have more control on the proof search flow. Indeed, before this internship, the tactics used for the automation were `auto` and `eauto`. These Coq tactics (whose theory is more described in 3.1) work by searching a proof of the current goal with the help of some statements that had been proved previously (e.g lemmas, theorems, ...) which are named *hints*. Moreover, in addition `eauto` is able to solve goals with ∃ statements whereas `auto` cannot. Nevertheless, in the current implementation of `auto` and `eauto`, it is not possible to specify forbidden hints and/or hints that must be used in the final proof. A first attempt to implement such a control tool is described in 3.2, and a huge improvement made is described in 3.3.

3.1 Prolog and `auto`

Since `auto` *and* `eauto` *are very similar, only* `auto` *will be explained here in details.*

3.1.1 Prolog

To understand `auto`'s operation, it is important to know how Prolog works. Prolog is a logic programming language based on first-order logic used to solve problems involving objects and relationships (see the section 1 of [5]). It contains a system of rules and facts of the form `Head :- Body.` that can be read as "if `Body` is true then `Head` is true". In this case, the first symbol in `Head` is called the *head symbol* of the rule, and more generally the *head symbol* of a formula is the first symbol appearing. A *rule* of the form `Head :- true.` can be written `Head.` and is called a *fact*. It is also possible to query the engine: for example, if we write `?- animal(tom).` with the following rules and facts :

```
cat(tom).              % There is a cat Tom.
animal(X) :- cat(X).   % For all X, if X is a cat then X is an animal.
```

then the output would be `Yes`, meaning that there is an animal Tom. With the same set of rules, the query `?- animal(jerry).` would return `No.` as there is no way to prove that this statement is true with only these two rules. Moreover, the head symbol of the first rule is `cat`, and the one of the second rule is `animal`.

Let's build the *proof search tree* of another example to see the complete flow:

```
mother_child(alice, david). % (1)
father_child(charlie, david). % (2)
mother_child(alice, bob). % (3)

parent_child(X, Y) :- father_child(X, Y). % (4)
parent_child(X, Y) :- mother_child(X, Y). % (5)
child_parent(X, Y) :- parent_child(Y, X). % (6)
```

The proof search tree corresponding to the query `?- child_parent(bob, alice).` is visible on the figure 7. The blue edges indicate the rules applied with success, whereas the red edges indicate the rules tried but that cannot be applied or leading to an incomplete proof. The nodes corresponds to the current goal to prove to complete the proof. This figure shows that the query will return `Yes.`, and the rules applied to find the conclusion are `(6)`, `(5)` and `(3)`. Notice that all rules are not tried at each step of the proof: only the rules with the same head symbol as the current goal are tried. This execution flow is generalizable to any proof search in Prolog [5].

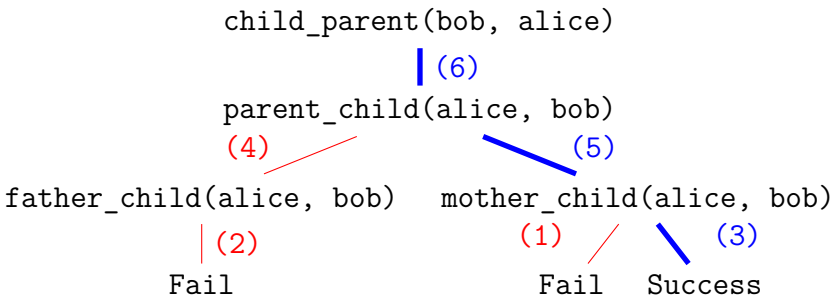


Figure 7: Proof search tree of the Prolog query `?- child_parent(bob, alice).`

3.1.2 `auto`

The Coq tactic `auto` works on the exact same principle. In this context, a rule is called a *hint*, and a set of hints is called a *hint database* [2]. Moreover, we define the

trace of a proof search as the ordered list of tuples containing the tried hints whose parent goal leads to a complete proof, and booleans indicating for each tried hint if it is used for the final proof or not. In the example of the figure 7, the trace would be : `[(6, true); (4, false); (5, true); (1, false); (3, true)]`. In the OCaml implementation done during this internship, more informations are stored in these tuples to make control easier (see the annex B). Furthermore, it is possible to print this trace in Coq with the standard `auto` and `eauto` by adding the keyword `debug` before these tactics.

The order in which the hints are tried at each step is always the same. First, OCaml tactic corresponding to the Coq tactic `assumption` is tried: it checks for every hypothesis if it is equal to the current goal, and ends the proof in this case. This means that the goal can be written as $P \implies P$ for a proposition P, which is a tautology. Secondly, the OCaml tactic corresponding to the Coq tactic `intro` is tried: it checks if the current goal has $\forall$-quantifiers, and introduces a new variable if it is the case, which allows to reduce the number of quantifiers. Then, if nothing above worked, all the hints taken from the local hint database and the hint databases given by the user are tried one after the other, then a recursive call is done on the new goal. To avoid infinite recursions, a maximum depth is set, and the current branch of the proof search fails directly if this depth is reached[2]. The local hint database is a hint database that, unlike the other ones defined before the beginning of the proof search, is built during the search from the current hypothesis and from the lemmas given by the user for the proof. To summarize, the order in which the hints are tried for the call `auto using lemma1, lemma2 with database1, database2.` is the following : the `assumption`, `intro`, the hints built from the hypothesis, `lemma1` and `lemma2`, then the hints contained in `database1` and `database2`. As several proofs can theoretically be found during the search, the search will end directly after founding the first one.

3.2 Control at the end of the proof search (contribution)

The first idea implemented in this internship was to add hints that must be used during the proof search as a simple control of the proof found during the search. This idea is quite simple: retrieve the trace, keep only the hints that are part of the complete proof, then verify for each indicated must-use hint if it is present in this list, and throw an error if not. This has been added to the automation function of the `coq-waterproof` library called `waterprove`, in an OCaml tactic named `wp_auto` (see the annex C for more details).

Here is a possible application: the users are able to use Ltac2 notations in their proofs that will call Coq tactics, which will call OCaml tactics. They can write by example `We conclude that <current goal>.` to try to solve the current goal with `waterprove`. However, this might be insufficient in some cases, and the user should give a lemma needed to prove the goal: it is possible to do so with the notation `By lemma, we conclude that <current goal>.`. Nevertheless, it is not checked in Coq's `auto` that the given lemma is indeed used or not, but this is possible to do so with our tactic, as shown in the figure 8.

On the figure 8, the line 1 declares the goal to prove, here $\forall n \in \mathbb{N}, n = n$, and the lines 2 and 6 are respectively the beginning and the end of the proof. The line 3 introduces a new variable $n \in \mathbb{N}$, the goal to prove after this line is $n = n$ with n fixed in the hypothesis. The line 5 shows that `wp_auto`, called by the notation `We conclude that <current goal>.`, is able to prove this goal. The line 4 is the one interesting here: with a call to `wp_auto`, a check occurres at the end on the trace returned by the proof search: here the complete trace

[2]This limit is set at 5 by default with `auto`.

```
Goal forall n: nat, n = n.
Proof.
    intros n.
    Fail By f_equal we conclude that (n = n).
    We conclude that (n = n).
Qed.
```

Figure 8: Example of a proof rejection because of an unused lemma

is `[(assumption, false); (intro; false); (@eq_refl, true)]`, which does not contain the lemma `f_equal`. Hence, the proof is rejected and an error is thrown, that's why the `Fail` at the beginning of the line succeeds[3].

The implementation of `wp_auto` has been made with mutable global variables that store and complete the trace during the proof search, which seems to be the easiest way of doing this. However, because the order in which the arguments of a function application are evaluated is not specified[4], the update of the variables occurred after the search and the checking. A non-sustainable solution was found, and was replaced with the improvement of the control, as described in 3.3.

3.3 Control during the proof search (contribution)

The solution provided in 3.2 does work, but is not satisfactory: it is possible to imagine a property admitting two distinct proofs with distincts arguments. The figure 9 shows one such example for the property $\forall n \in \mathbb{N}, n+1 = n+1$ (with `S n` meaning "the successor of `n`", namely $n+1$).

```
Goal forall n: nat, S n = S n.
Proof.
  intros n.
  apply eq_refl.
Qed.
```

```
Goal forall n: nat, S n = S n.
Proof.
  intros n.
  apply f_equal.
  apply eq_refl.
Qed.
```

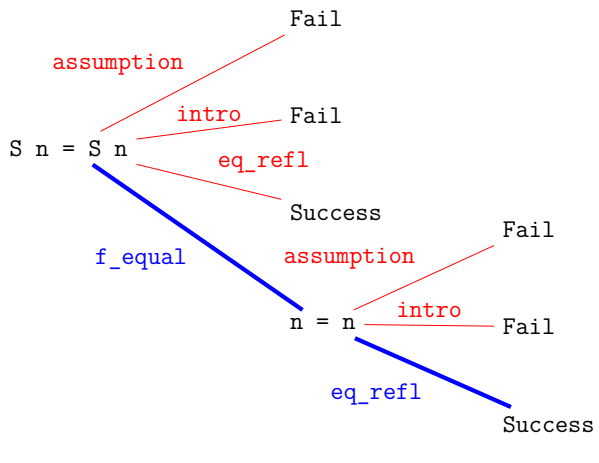
Figure 9: Example of two distinct proofs for a same property

The right proof use the property `f_equal` and not the left one: as the left proof is the "simplest" one, in the sense that the left proof will be found before the right proof during the proof search by `auto` / `wp_auto`. Thus, writing `By f_equal we conclude that (S n = S n).` will be rejected with the current version of `wp_auto` because `f_equal` is not used in the proof. Therefore, if a user indicates `f_equal` as a lemma, the proof will be rejected but it does not seems right to do so : as it exists a proof using this lemma, `wp_auto` should return a proof containing `f_equal`.

Hence there should be a new constraint on `wp_auto` : instead of checking the first proof found, the tactic should return the first proof found such that every wanted hint is used. To

[3] In Coq, `Fail` succeeds if an error is thrown during the execution of the line, and returns an error if not.
[4] https://v2.ocaml.org/manual/expr.html#sss:expr-functions-application

achieve this, it is needed to transfer informations between tactics: continuing the same method as in 3.2 would be too complex to maintain, that's why the solution developed in 2.2.3 was necessary. The idea here is to make the tactics return the trace corresponding to the subtree to which they are the root. Tactics that fail only return an error containing a trace with only themselves, and the ones that succeed merge all traces retrieved from tactics applied after (that are lower in the proof search tree) then return the complete trace of the branch.




```
Failed attempt: [
  (false, assumption);
  (false, intro);
  (true, eq_refl)
]

Final trace: [
  (false, assumption);
  (false, intro);
  (false, eq_refl);
  (true, f_equal);
  (false, assumption);
  (false, intro);
  (true, eq_refl)
]
```

Figure 10: Shorten proof search tree and proof search trace for the goal `S n = S n` requiring the use of `eq_refl` and `f_equal`

On the figure 10, the proof search tree and the proof search trace for the goal `S n = S n` show that a first proof has been found, containing only `eq_refl`. Then, since `eq_refl` and `f_equal` are required to be used, the proof is considered as a success but is invalided, so the proof search continues and a new proof containing both required hints is found, which is the wanted behavior. The exact same process has been done with `wp_eauto`, which is to `eauto` what `wp_auto` is to `auto`. This allowed to change the previous version of `waterprove` that used `auto` and `eauto` with a new one that uses `wp_auto` and `wp_eauto`.

The current version of the implementation is working but some improvements can be done. The most important one is the fact that the check that every must-use lemma is present occurres at the highest depth in the proof search tree. In the case of the figure 10, the checks are done in the edges coming out of the `S n = S n` node. The figure 11 shows an example where this is a problem: as the check is done at the depth 1, if the proof found containing `lemma_A` and `lemma_B` is rejected after the check, the dashed branches will never be tried, including the one that would succeed with `lemma_C`. The solution is not that simple to implement: making the check at the leafs is not possible because one goal can generate several subgoals and a must-use hint is possibly used in only one of these subgoals. A possible solution would be to flatten the proof search tree to do the check only in the last subgoal.

Moreover, the current implementation offers a framework that can be easily extended: it is easy to add other operations during the proof searches. For example, it is possible to give each hint a weight corresponding to its complexity, then to find all the possible proofs and return a proof with the minimal weight if it exists, corresponding to the "simplest" proofs among all the found ones.

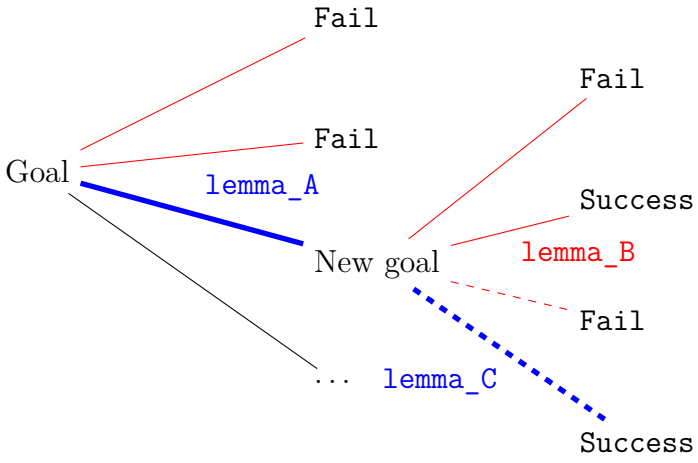


Figure 11: Example of a proof search failing where it should not

4 Automated rewriting

The second axis of research to improve the automation is to improve its strength to make the proofs with Waterproof look like pen-and-paper proofs as much as possible. Two options have been considered: first the resolution of arithmetic equalities and inequalities, and secondly using rewriting based on current hypotheses to the proof search. After reading the Coq manual [2] and Coq's source code [3], it appeared that there was not enough time during this internship to follow this path. Consequently, it was decided to focus on the use of rewrites during the proof search. This idea is based on the automatic use of current hypothesis to generate rewrite hints (see 4.2) for the Coq tactic `autorewrite` (see 4.1).

4.1 Rewriting and `autorewrite`

`rewrite` is one of the most fundamental tactics in Coq. It allows to replace subterms in a given expression with other subterms that have been proven to be equal [2]. For example, if the current goal is `f x = f z`, and the hypothesis `H` states that `x = y`, then the tactic `rewrite H.` will transform the goal into `f y = f z`. Variables can be unified if the used hypothesis contains ∀-quantifiers. With the same idea defined in 3.1.2, let a *rewrite hint* be a registered term typed as an equality, and a *rewrite hint database* a set of rewrite hints [2]. It is to be noted that hint databases and rewrite hint databases are completely distincts: `auto` will never use a rewrite hint. these rewrite hint databases can be used by `autorewrite` that will apply rewritings based on the hints contained in the given rewrite hint databases. Each rewrite hint will be applied until it fails to be used as rewriting. It is possible to give `autorewrite` a Coq tactic that will be applied between each rewriting step. It can be useful to use this possibility with `auto` or `eauto` for example to try to resolve the goals with automated rewritings that are not declared in `auto`'s hint databases.

4.2 Automated use of hypotheses (contribution)

Rewriting hints are really useful to solve a goal, but you have to declare them before using `autorewrite` to be able to use them. Indeed, as explained in 4.1, `auto` (and `wp_auto` as well) cannot use rewrite hints or rewrites based on equalities in the hypotheses. The figure 12 shows that the goal $\forall x \in \mathbb{R}, x = 0 \implies \sin(x) = 0$ cannot be solved directly with `auto` but works with a rewriting step beforehand: the hypothesis `H` (which corresponds to `x = 0`) cannot prove directly the goal as the goal `sin x = 0` is not syntactically equal to `x = 0` nor to `sin_0` (which corresponds to `sin 0 = 0`).

```
Goal forall x: R, x = 0 -> sin x = 0.
Proof.
  intros x H.
  Fail progress (auto using sin_0).
  rewrite H; auto using sin_0.
Qed.
```

Figure 12: Example of a proof where `auto` fails but `rewrite` succeeds

Thus, the idea implemented in this internship is to use the current hypotheses as rewrite hints during the proof search, then to use `autorewrite` with `wp_auto` using the rewrite hint database composed of hints made by the hypotheses. To make this work, it was needed to retrieve the current hypotheses, then change the `autorewrite` implementation in Coq core to be able to use locally defined rewrite hints.

```
Goal forall A: Set, forall x y z: A, forall f: A -> A, x = y -> f y = f z
↪  -> f x = f z.
Proof.
  intros A x y z f H1 H2.
  Fail progress auto.
  waterprove.
Qed.
```

Figure 13: Example of a proof where `auto` fails but `waterprove` succeeds

The figure 13 shows that our tactic `waterprove` works in a case where `auto` does not because of the automatic add of rewrite hints and the apply of a custom `autorewrite`. Here, we have to show that, given A a set, $x, y, z \in A$ and $f : A \longrightarrow A$ such that $x = y$ (`H1`) and $f(y) = f(z)$ (`H2`), then $f(x) = f(z)$: it is not possible to directly prove this only with auto because neither `H1` (`x = y`) nor `H2` (`f x = f z`) are syntactically equal to the goal (`f y = f z`). However, adding a rewriting step based on `H1` before doing a proof search returns a complete proof as `x` is substituted to `y` in the goal.

A possible way of improving the current version of `wp_autorewrite`, the tactic that is to `autorewrite` what `wp_auto` is to `auto`, is to add the control of used hints as in 3.3 for the same purpose: controlling that each must-use hint (that can be a hypothesis) is used either during the proof search with `wp_auto`, or during the rewriting steps with `wp_autorewrite`. This is currently not implemented, but it is doable by using the trace returned by the `wp_auto` tactic using the typed tactics interface presented in 2.2.3.

5 Optimization

During the implementation of the algorithms shown in 3, the existing hint databases used in `coq-waterproof` have been slightly modified to adapt to the new automation system. This lead to a great increase in the number of tried hints: for some proof searches, the number of tried hints skyrocketed to hundreds of thousands to millions, which made the automation system unusable for the users as the proof search time was too long to be used. With other

minor corrections, the number of tried hints drastically decreased, but the effects are still visible.

5.1 Internal proof state

In the Coq core, the goals (see 2.1) are represented by the type `Proofview.Goal.t` (available in the file `engine/proofview.ml` in [3]). This type stores several elements where the important ones are: the current environment, which is the merge of the local environment and the global context, and the conclusion. That means that to check if progress has been made, only these fields are relevant, which can be seen in the definition of `Proofview.tclProgress` (in the same file) as only these fields are used.

5.2 Branch skipping in the proof search tree (contribution)

The principle behind this optimization is to skip branches leading to proof states already visited. Indeed, as described in 5.1, the environment and the conclusions are enough to entirely determine the proof state. Thus, it is not necessary to continue to search in a branch if the current state has already been visited. It is particularly useful for hints that make the conclusions switch in a cycle. For example, the hint `eq_sym` of type `forall [A : Type] [x y : A], x = y -> y = x` can be applied twice in a row on the same conclusion that will return to the initial state.

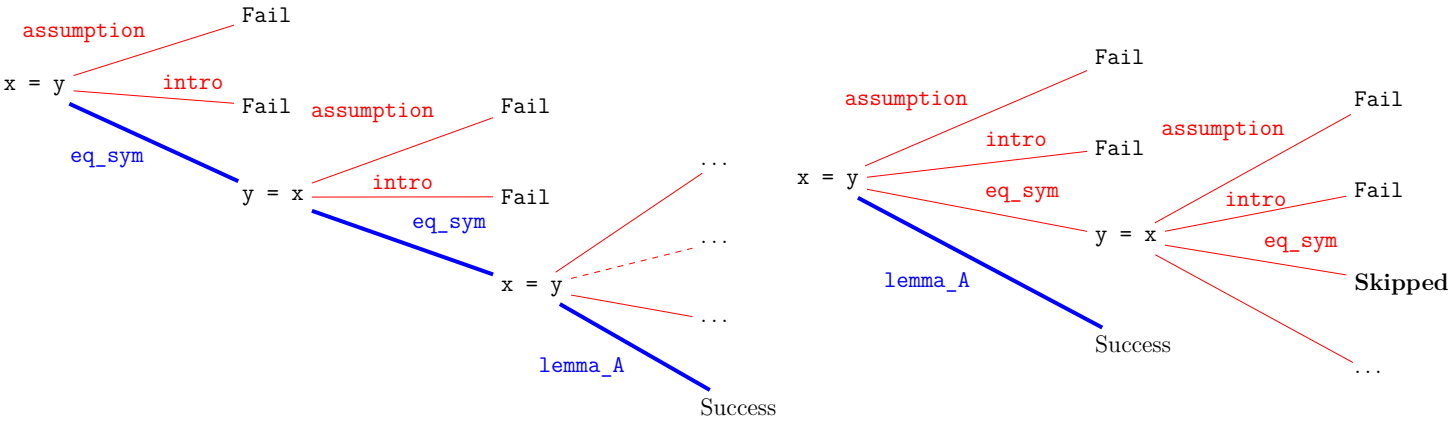


Figure 14: Comparaison of proof search tree with and without the optimization

The figure 14 shows that for a proof search on the goal `x = y` (with `x` and `y` any variables), it is possible to apply twice in a row the lemma `eq_sym` that uses the commutativity of the equality to make the goal become `y = x`. Hence, this hint can be applied an infinite amount of times, but as there is a maximal search depth (see 3.1.2), in this case set at 3, the application of this hint is limited and thus `lemma_A` solves the goal at depth 3, which is visible on the left proof search tree. When the optimization is turned on, the algorithm detects that the environment and the conclusion have not changed: consequently, it is possible to find a strictly simpler proof (in the sense of the number of hints applied), and so the branch is skipped, as indicated on the right proof search tree. A complete example showing the improvement is available in annex D.

To provide a more precise evaluation of the impact of the optimization, a comparison was conducted on the total number of hints attempted during the compilation process of the `coq-waterproof` library. Without the optimization, a total of 1,261,341 hints were tried during compilation. However, with the optimization implemented, this number significantly decreased to 668,513. Upon closer analysis, it was discovered that a specific test

file, `tests/tactics/ItHolds.v`, was responsible for a substantial increase in the number of attempted hints. Upon disabling the compilation of this file, the results showed a decrease in the total number of tried hints to 207,909 without the optimization, and 154,318 with the optimization. This shows that the optimization reduce significantly the number of hints tried, and so the compilation time. This reduction is even greater in some edge cases where more than half of the proof searches is skipped with this optimization.

6 Conclusion

Several improvements have been made to the automation system of `coq-waterproof` during this internship. Notably, the control of automation has been enhanced, allowing for better management of hints to be used in the final proof. Additionally, the automation system's robustness has been strengthened by automatically adding rewriting steps based on current hypotheses. Furthermore, an optimization has been introduced during proof searches, resulting in a significant reduction in the number of attempted hints required. These improvements required the development of a comprehensive interface that expands the tactic monad in Coq's OCaml API.

Moving forward, certain aspects raised in this report require further attention. Specifically, there is a need to create a restricted version of `wp_autorewrite` to complete the improvements initiated in 3.3. Additionally, the algorithm employed for proof searches should be thoroughly reconstructed to prevent unnecessary branch skips arising from the control applied during the search process.

Taking a broader perspective, future research and development on `coq-waterproof` should prioritize enhancing its practicality for both students and teachers. This goal has been the driving force behind this internship and has been shared by other individuals involved in the project throughout the year.

References

[1] J. Wemmenhove, T. Beurskens, S. McCarren, J. Moraal, D. Tuin, and J. Portegies, *Waterproof: Educational software for learning how to write mathematical proofs*, 2022. arXiv: 2211.13513 [math.HO].

[2] Coq Team, *Coq's reference manual*. [Online]. Available: https://coq.inria.fr/distrib/current/refman/.

[3] Coq Team, *Coq's source code*. [Online]. Available: https://github.com/coq/coq.

[4] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer, "Mtac2: Typed tactics for backward reasoning in coq," *Proc. ACM Program. Lang.*, vol. 2, no. ICFP, Jul. 2018. DOI: 10.1145/3236773. [Online]. Available: https://doi.org/10.1145/3236773.

[5] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*. Springer Berlin Heidelberg, 1984. DOI: 10.1007/978-3-642-96661-3. [Online]. Available: https://doi.org/10.1007/978-3-642-96661-3.

[6] M. Knobelsdorf, C. Frede, S. Böhne, and C. Kreitz, "Theorem provers as a learning tool in theory of computation," in *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ser. ICER '17, Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 83–92, ISBN: 9781450349680. DOI: 10.1145/3105726.3106184. [Online]. Available: https://doi.org/10.1145/3105726.3106184.

[7] A. Thoma and P. Iannone, "Learning about proof with the theorem prover LEAN: The abundant numbers task," *International Journal of Research in Undergraduate Mathematics Education*, vol. 8, no. 1, pp. 64–93, Jul. 2021. DOI: 10.1007/s40753-021-00140-1. [Online]. Available: https://doi.org/10.1007/s40753-021-00140-1.

Annexes

A – Typed tactics functor

```
(** Generic mergeable type *)
module type Mergeable = sig

  (** Type of the elements *)
  type elt

  (** Empty value *)
  val empty : elt

  (** How to merge two elements *)
  val merge : elt -> elt -> elt

end

(** Generalization of tactics defined in coq-core for {! Mergeable}-typed tactics *)
module TypedTactics: functor (M: Mergeable) -> sig

  (** Merge of tactics' returned elements *)
  val typedThen :
    M.elt Proofview.tactic ->
    M.elt Proofview.tactic ->
    M.elt Proofview.tactic

  (** Same as {! typedThen} with a list of tactics *)
  val typedLongThen :
    M.elt Proofview.tactic list -> M.elt Proofview.tactic

  (** Generalization of {! Proofview.Goal.enter} *)
  val typedGoalEnter :
    (Proofview.Goal.t -> M.elt Proofview.tactic) ->
    M.elt Proofview.tactic

  (** Generalization of {! Proofview.tclINDEPENDENT} *)
  val typedIndependent :
    M.elt Proofview.tactic -> M.elt Proofview.tactic

end = functor (M: Mergeable) -> struct

  let typedThen (tactic1: M.elt tactic) (tactic2: M.elt tactic): M.elt tactic =
    tactic1 >>= fun elt1 ->
    tactic2 >>= fun elt2 ->
    tclUNIT @@ M.merge elt1 elt2

  let typedLongThen (tactics: M.elt tactic list): M.elt tactic =
    List.fold_left typedThen (tclUNIT M.empty) tactics

  let typedGoalEnter (f: Goal.t -> M.elt tactic): M.elt tactic =
    Goal.goals >>= fun goals ->
    let tactics = List.map (fun goal_tactic -> goal_tactic >>= f) goals in
    List.fold_left (fun acc tac -> typedThen acc tac) (tclUNIT M.empty) tactics

  let typedIndependent (tactic: M.elt tactic): M.elt tactic =
    tclINDEPENDENTL tactic >>= fun elts -> tclUNIT @@ List.fold_left M.merge M.empty elts

end
```

This is the full implementation of the typed tactics functor. It is possible to extend it with even more generalization of Coq's OCaml API functions, but only these ones were useful for the algorithms' implementation. This piece of code can be found on the git repository in the file `src/proofutils.ml`.

The symbol `>>=` in the context of monads is a convention to denote the `bind` function. Thus, the two following lines are equivalent:

```
(* Version with the >>= notation *)
wrapped >>= fun value -> result

(* Version without the notation *)
bind wrapped (fun value -> result)
```

B – OCaml trace implementation

```
    (**
    Trace atome type

    Can be read as (is_success, depth, hint_name, hint_db_source)
    *)
    type trace_atom = bool * int * Pp.t * Pp.t

    (**
    Trace type
    *)
    type trace = {
    log: bool; (** Are tried hints printed ? *)
    current_depth: int; (** The current depth of the search *)
    trace: trace_atom list (** The full trace of tried hints *)
    }
```

This is the types corresponding to the trace defined in 3.1.2. The module `Pp` of Coq's OCaml API defines pretty printing functions, and the type `Pp.t` corresponds to an expression that can be printed. It can be considered as an improved version of `string`.

C – Description of `wp_auto`

The description of `wp_auto` made in 3.2 has been simplified not to flood the explanations with useless implementation details. Indeed, in the implementation `wp_auto` only returns the trace of the proof search but never fails: another tactic has been created for this: `rwp_auto` (for *restricted* `wp_auto`). All these functions can be found in the file `src/wp_auto.ml` of the repository. Moreover, `We conclude that <current goal>.` does not call directly `wp_auto` nor `rwp_auto`, but calls `waterprove` or `rwaterprove` defined in `src/waterprove.ml` which will call `wp_auto` / `rwp_auto` and `wp_eauto` / `rwp_eauto`.

D – Example of successful optimization

```
1  Require Import Ltac2.Ltac2.
2
3  Require Import Waterproof.Waterproof.
4  Require Import Waterproof.Waterprove.
5  Require Import Waterproof.Automation.
6
7  Waterproof Clear Automation.
8
9  Local Parameter A: Type.
10 Local Parameter a b: A.
11 Local Parameter a_eq_b: a = b.
12
13 Waterproof Enable Automation Core.
14
15 Local Hint Resolve a_eq_b : core.
16
17 Goal b = a.
18 Proof.
19   pose eq_sym as eq_sym.
20   (* Std.auto Std.Debug (Some 5) [] None. *)
21   (* waterprove 5 false [] Main. *)
22 Qed.
```

- The lines 1-5 are the necessary imports of `coq-waterproof`.
- The lines 7-13 correspond to the definition of parameters that will be used in the proof, and the lemma `a_eq_b` that have to be placed in a hint database different than the local one to be tried in the correct order (`a_eq_b` should be tried after `eq_sym`).
- The lines 17-22 correspond to the proof where the proof search will be tested.

To try the proof search with Coq's auto, the line 20 should be uncommented, and to try the proof search with the optimization presented in 5, it should be the line 21.

The list of applied hints by the proof search of line 20 is:
`[eq_sym, eq_sym, eq_sym, eq_sym, eq_sym, a_eq_b]`
The line 20 will display the follow output upon execution[5]:

```
(* debug auto: *)
* assumption. (*fail*)
* intro. (*fail*)
* simple apply eq_sym. (*success*)
** assumption. (*fail*)
** intro. (*fail*)
** simple apply eq_sym. (*success*)
*** assumption. (*fail*)
*** intro. (*fail*)
*** simple apply eq_sym. (*success*)
**** assumption. (*fail*)
```

[5]The `a_eq_b` hint is hidden behind `trivial` in the output.

```
12 **** intro. (*fail*)
13 **** simple apply eq_sym. (*success*)
14 ***** assumption. (*fail*)
15 ***** intro. (*fail*)
16 ***** simple apply eq_sym. (*success*)
17 ***** exact a_eq_b (in core). (*fail*)
18 ***** simple apply @eq_refl (in core). (*fail*)
19 ***** simple apply f_equal_nat (in core). (*fail*)
20 ***** simple apply eq_sym ; trivial (in core). (*success*)
```

The list of applied hints by the proof search of line 21 is: `[eq_sym, a_eq_b]`. The line 21 will not display anything without enabling an option before the compilation of the library. Here is the output with this option:

```
1 (* info wp_auto: *)
2 Trace:
3 x assumption in ().
4 x intro in ().
5 o eq_sym in ().
6 x assumption in ().
7 x intro in ().
8 x eq_sym in ().
9 o a_eq_b in (core).
```

This shows that the hint `eq_sym` have been applied only once with the optimization, but 5 times without, which demonstrates that the the optimization does work in some cases.