Designing Proof Deautomation for Rocq

Jessica Shi [1], **Cassia Torczon** [1], **Harrison Goldstein** [1, 2], **Andrew Head** [1] **and Benjamin C. Pierce** [1]

[1] *University of Pennsylvania, Philadelphia, PA*
[2] *University of Maryland, College Park, MD*

Abstract

Proof assistant users rely on automation to reduce the burden of writing and maintaining proofs. By design, automation hides the details of intermediate proof steps, making proofs both shorter and more robust. However, we observed in a need-finding study that users sometimes do want to examine the details of these intermediate steps, especially to understand how the proof works or why it has failed. To support such activities, we describe a *proof deautomation* procedure that reconstructs the underlying steps of an automated proof. We discuss the design considerations that shaped our approach to deautomation — in particular, the requirement that deautomation should remain informative even for failing proofs.

Keywords: Proof assistants, proof automation, proof refactoring.

1 Introduction

Proof assistants are environments that support the interactive construction of machine-checked proofs. Proof assistants have been used to great effect in domains across computer science and mathematics; in the programming languages community, for example, researchers sometimes formally justify theoretical claims by mechanizing their proofs in a proof assistant. Unfortunately, productive usage of proof assistants requires substantial effort and expertise; indeed, in a survey [1] of users of the Rocq proof assistant [2], 46 percent of respondents had a doctoral degree.

One obstacle to improving proof assistant usability is that two contributing forces are in conflict: interactivity and automation. To see why, consider this example proof script:

```
Lemma andb_true_r (b : bool) : b && true = b.
Proof.
  destruct b.
  - simpl. reflexivity.
  - simpl. reflexivity.
```

A user would typically write such a proof script incrementally, by interacting with the proof assistant at each step to see what the *proof state* looks like. For example, if the user evaluated the proof to just after the `.`, the proof assistant would display this proof state (up to formatting):

```
goal 1 is:                    goal 2 is:
------------------------      ------------------------
(true && true) = true         (false && false) = false
```

Proof states provide useful context about what is happening in the proof: here, the user can see that doing `destruct b` generates two goals, one where `b` is `true` and one where `b` is `false`.

Commands such as `destruct`, `simpl`, and `reflexivity` are built-in *tactics* provided by the Rocq proof assistant. Rocq also provides support for *proof automation*, including higher-order tactics, called *tacticals*. For example, the script above can be written more succinctly as follows:

```
Proof. destruct b; simpl; reflexivity.
```

The semicolon tactical sequences tactics, where `a ; b` applies `b` to every goal generated by `a`. Besides tacticals, other automation constructs include search-based procedures that discharge certain classes of proof obligations entirely, and language features that enable users to write custom tactics.

Automation can greatly reduce the burden of proof assistant usage. However, automation also makes proving less interactive. Whereas in the original script above, the user can see the intermediate proof states at each step, in the more automated script, the proof is completed in just one step, so there are no intermediate states. Indeed, there is a fundamental tension between making intermediate steps visible and automating them away.

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

We think that users should not have to choose between a proof that can be interacted with versus a proof that is automated. To this end, we describe a *proof deautomation* procedure that reconstructs the underlying steps of an automated proof. Intuitively, the deautomation of a proof involves unrolling its automation so that the steps the proof assistant takes are made explicit and individually executable. Such a deautomation procedure enables users to better understand an automated proof, especially if it is not working as expected, by restoring their ability to interact with the proof.

We study deautomation on a core set of automation features drawn from Rocq's `Ltac` language [3]. Concretely, we offer these contributions:

- We characterize users' needs around understanding automation, based on an interview study that we conducted with practicing Rocq users (§2).
- After introducing a motivating example (§3), we discuss several design considerations for making deautomation informative and controllable (§4).
- We develop an algorithm for deautomation that captures the execution of the original script and extracts a step-by-step view of the proof. In particular, we explore how to usefully deautomate failing scripts. We prove (in Rocq) that deautomation preserves semantics up to failures. We also implement a proof-of-concept `VS Code` extension. Some technical details are provided in (§5).

We also compare to related work (§6) and share ideas for future work (§7).

2 Need-Finding Study

Prior studies have broadly suggested that users desire greater understanding about automated proofs [4], [5], but have not deeply explored what, concretely, users want to understand, or where, exactly, are the sources of friction that frustrate such understanding. To gain better insight into these questions, we conducted a need-finding interview study.

Our study consisted of interviews with eleven participants. We opted for interviews rather than a survey or observations, anticipating that this would allow us to investigate users' varying experiences with automation in depth. We recruited participants from personal outreach, mailings, and word of mouth. All eleven were experienced Rocq users. Seven were Ph.D. students, two were undergraduate students, one was an engineer, and one was an academic researcher. The interviews followed a semi-structured guide focusing on participants' experiences writing, reading, and maintaining automation. Participants were asked to share examples from their own proof developments.

After the interviews, we performed a thematic analysis [6] where one of the authors reviewed their notes from all interview sessions. Anecdotes and quotations were checked against recordings of each session. This process led us to the idea of deautomation that we present in this paper.

In the rest of this section, we describe the findings from this study that clarify the value of — and potential designs for — tools for deautomation.

Desires to understand automation. The study participants described a variety of situations where they wanted to know more about their automation, especially if it was not working as expected. For example, participant P7 recounted a time when they needed to fix proofs, originally written by others, due to a version change. Showing us one such proof, written as series of tactics chained by semicolons, P7 said, "Just figuring out where exactly it broke was really hard." They explained,

> "Working with semicolons means if you ever have to look at something you wrote that already has semicolons in it, you probably have to change it back, just because it's not helpful to jump from here [*pointing to the start of a sentence with semicolons*] to here [*pointing to the end*]. There's a lot going on in here, so I would like to understand what it is."

When using automation to operate on multiple goals at once, participants wanted to understand how the behavior of the automation differed on different goals. P5 described how, if a script was applied to all goals, they sometimes needed to know which goals it was failing to solve. P9, who showed examples where they worked with several dozen goals at once, said, "Typically, when I change my tactic, I have no idea how the goals I solved changed compared to my previously written tactic."

P8 expressed dissatisfaction with existing support for operating on multiple goals, remarking, "This is supposed to be an interactive theorem prover." However, they continued, semicolons and the "`all:`"

selector break the interactive process of "run a tactic, see the result, run another one." Several other participants also commented on difficulties with not being able to see proof states at intermediate points of automated proofs (P1, P4, P5, P7).

Participants also sought deeper insight when a tactic such as `auto` surprisingly solved a goal, indicating that a premise might have been false (P4), and when they needed to use automated tactics from other libraries (P2).

Approaches to understanding automation. Many participants described strategies for temporarily undoing automation, as well as choices to avoid automation altogether in certain situations.

In order to find and fix failures in automated proofs, participants would sometimes undo parts of their automation, e.g., by turning semicolons into periods (P3, P4, P5, P7, P9) or inlining the body of a custom tactic (P1, P7, P8, P9). For example, P9 simulated how they debug their custom tactic by copying 20 lines of the tactic into the proof script, where tacticals would then need to be further undone. After repairing the tactic, they would fold the changes back into the original automated style. Sometimes, this process of undoing automation was "easy" (P3), but it could also be "frustrating" (P7).

P1's experiences led them to change their style of automation. Previously, they had written custom tactics to solve all cases of their proof at once. But the difficulty of determining which cases were failing later led them to transition away from these tactics to finer-grained automation.

Other participants also described reasons to opt for less automation. In order to make their proof developments accessible to undergraduate collaborators and to encourage understanding of the "underlying structure" of the proofs, P10 chose to write these proofs with "more primitive tactics." P11 was concerned that, if they automated some but not all of their proof, they would reach a proof state with goals where "it's not clear what the relationship between them is." As a result, P11 said, "I struggle with the decision between trying to get everything automated away versus trying to maintain a structure I can understand looking back." The strategy of writing proofs more simply to support inspection is not unique to our study participants: it also resembles practices employed in engineering high-profile large-scale proofs [7, Section 2.3, for example].

Our study surfaced tensions between automation and interactivity, between automation and debugging, and, more broadly, between automation and understanding. Balancing these competing priorities imposes a significant burden of writing and rewriting proofs. To relieve users of this burden, we wish to transform deautomation from a tedious manual task to a smoother tool-assisted process.

3 Motivating Example

Consider this example of an automated proof[1], adapted from the exercise solutions to *Logical Foundations* [8]. Suppose a user has proved an equivalence between two ways of evaluating expressions, one as an `Inductive` and the other a function. Later, they realize only the forward direction of the equivalence is relevant to them, so to simplify the development, they decide to change the equivalence to an implication. Refactoring the proofs is mostly straightforward — they just need to delete the proofs for the backward direction — but bizarrely, the forward direction now fails!

```
Theorem bevalR_beval :
  ∀ (b : bexp) (bv : bool),
  bevalR b bv → beval b = bv.
Proof.
  induction 1; simpl; intros;
  try (rewrite aevalR_aeval in H, H0;
       rewrite H; rewrite H0);
  reflexivity.
```

If the user evaluates this script, the `reflexivity` tactic triggers an error message saying that `n1 =? n2` and `aeval a1 =? aeval a2` cannot be unified. What is wrong here?

1 Note that this proof contains two kinds of tacticals: semicolons, described previously, and the `try` tactical, where `try t` tries tactic t and does nothing if t fails.

Without Deautomation We start by walking through a potential set of steps for debugging this proof manually. Of course, different users will have different debugging habits, but we provide this sample walkthrough to demonstrate some sources of tedium that can occur.

Since the error message reports that `reflexivity` is failing, the proof writer starts by changing the last semicolon into a period, so that they can see the place that fails. (We highlight the edit made in each of the steps below.)

```
Proof.
  induction 1; simpl; intros;
  try (rewrite aevalR_aeval in H, H0;
       rewrite H; rewrite H0).
  reflexivity.
```

But now the `reflexivity` succeeds, so where did the error go? The user steps back before the `reflexivity`, where they see there are four goals. Upon closer examination, they realize that the first goal can in fact be solved with `reflexivity`, as can the second. So, to examine the failure, they opt to skip the first two goals, as shown on the left, so that they can now see in full the proof state that `reflexivity` is failing to execute on, as shown on the right:

```
Proof.
  induction 1; simpl; intros;
  try (rewrite aevalR_aeval in H, H0;
       rewrite H; rewrite H0).
  3: {
    reflexivity.
```

```
a1, a2 : aexp
n1, n2 : nat
H : aevalR a1 n1
H0 : aevalR a2 n2
------------------------------------
(aeval a1 =? aeval a2) = (n1 =? n2)
```

Certainly this does not seem solvable by `reflexivity`, but why is the proof in this state? To investigate, the user can remove an earlier semicolon, perhaps before the `try`.

```
Proof.
  induction 1; simpl; intros.
  3: {
    try (rewrite aevalR_aeval in H, H0;
         rewrite H; rewrite H0).
    reflexivity.
```

Stepping back and forth, they find the `try` does nothing, so they know a tactic inside failed. To find the culprit, they start by separating out the first `rewrite`:

```
Proof.
  induction 1; simpl; intros.
  3: {
    rewrite aevalR_aeval in H, H0.
```

Aha! It fails. The user realizes that the `rewrite` worked previously, when `aevalR_aeval` was an if-and-only-if, but now that it is a single implication, they need to use `apply` instead.

Observe that in order to understand why the proof was failing, the user has to maneuver their way around the automation constructs — e.g., the semicolons and the `try` tactical.

With Deautomation Deautomation aims to protect the user from the tedium of this manual maneuvering by returning a version of the proof script free of automation. In this case, deautomating the original proof script outputs:

```
Proof.
  induction 1.
  - simpl. intros. reflexivity.                    U
  - simpl. intros. reflexivity.
  - simpl. intros.
    (* tried and failed to run:                    T
        rewrite aevalR_aeval in H, H0. *)
    Fail reflexivity. admit.                       F
  - simpl. intros.
    (* tried and failed to run:
        rewrite aevalR_aeval in H, H0. *)
    Fail reflexivity. admit.
```

The user can immediately jump to the intermediate point in the third case where `reflexivity` is failing (F). They can also see a trace (T) of the failed tactics within the `try`. They can now use this information to find the bug described earlier.

Beyond assisting the user with pinpointing the location and cause of failure, deautomation also supports their understanding of the proof overall. For example, if the user wants to understand (U) why the first two cases succeed, they can readily see and step through the proof snippets there. Similarly, if the user wants to understand whether the fourth case is failing for the same reason (in this case, it is), they can also easily step there.

4 Design Considerations

Having motivated why we want to deautomate proofs, we next discuss considerations when designing deautomation. Considerations include what deautomation should output, especially on failing proofs, and how users can exercise control over what is deautomated.

4.1 Desirable Outputs

Consider the example proof script from the introduction with `destruct b; simpl; reflexivity.` We could, in theory, deautomate this proof into

```
Proof. destruct b. 1: simpl. 2: simpl. 1: reflexivity. 1: reflexivity.
```

Such a proof mimics the order in which the tactics are executed in the automated proof. But it would be unusual to write a proof in this format; instead, a user is likely to work on one goal at a time:

```
Proof.
  destruct b.
    simpl. reflexivity.
    simpl. reflexivity.
```

In our view, this is what deautomation should output. That is, a deautomated proof should resemble the format of a proof that a user would plausibly write when not using automation.

4.2 Failure Recovery

Users in our need-finding study indicated they especially want to better understand their automation when faced with a failing proof. Automated proofs are often "all or nothing": either they solve the goal completely, or (as in §3) they fail completely. Deautomation, by contrast, needs to support *failure recovery* — deautomating a failing proof should provide an informative result.

What do we mean by "informative"? Our rule of thumb is that, via deautomation, users should be able to access the information they want about their automated proofs as *readily* and *flexibly* as they could if they had written their proofs without automation. This rule shapes what deautomation should look like up to and beyond points of failure.

Before a failure, any number of tactics may have succeeded, providing important context about what progress has been made in the proof. This context should be preserved by deautomation, so that users can step through the deautomated proof preceding a failure and interact with the failing step. In §3, this context included the contents of the two successful cases, the initial successful tactics in the failing case, the trace of the no-op `try`, and the localized report that `reflexivity` failed.

We have several options for how to continue *beyond* a point of failure, if at all. We stated above that users should be able to work *flexibly* with deautomated proofs. When users work with proofs where different branches — the cases in a proof by induction, for example — are explicit, they can choose, with the help of `admits`, what branches they want to address, and in what order. For deautomation to provide the same flexibility, it should support recovering from failures on multiple branches (though not multiple failures on the same branch). Again, the example in §3 is consistent with this aim, where the user could choose to examine either of the failing branches, or both.

Failure recovery becomes even more complex when handling tacticals such as `first`, where `first` [t_1 | ... | t_n] tries each t_i in the list until one succeeds. That is, `first` provides its own internal failure recovery! When t_1 fails, `first` recovers from this failure and continues on to t_2. The presence

of both external failure recovery via deautomation and internal failure recovery via `first` requires careful consideration of how these should interact.

4.3 Levers for Control

Deautomation turns a compact script into a more verbose one. Depending on the needs of the user, details in different parts of this script may be more or less useful. We want our deautomation tool to allow the user to exert some control over exactly what is deautomated.

Returning to §3, by default all of the tacticals in the proof are selected for deautomation:

```
Proof.
  induction 1 ; simpl ; intros ;
    try (rewrite aevalR_aeval in H, H0 ;
         rewrite H ; rewrite H0) ;
    reflexivity.
```

If the user, for example, has no desire to see individual invocations of `simpl` and `intros` in the deautomated script, they may prefer to deselect the corresponding semicolons to opt them out of deautomation, i.e. `induction 1 ; simpl ; intros ;` . Then, the output begins instead with

```
Proof.
  induction 1; simpl; intros.
  - reflexivity.
```

An additional aspect that the user may want to control is whether to deautomate the internals of user-defined tactics. In particular, the user should be able to decide whether to treat a user-defined tactic opaquely, as they would any built-in tactic, or transparently, which exposes it for deautomation.

5 Technical Details

With the design considerations in mind, we next introduce a formal theory of deautomation, as well as a proof-of-concept implementation.

5.1 Grammar

We start by defining the subset of `Ltac` that we support. The grammar is stratified into atomic tactics, tactics, sentences, and scripts.

Atomic tactics are opaque to deautomation. To reason about how they behave, we assume a black-box `run-atomic` function that determines the result of executing atomic tactics. Its type is $\texttt{atomic} \to \texttt{goal} \to (\texttt{list goal} + \bot_n)$. That is, executing an atomic tactic on a goal results in either a (possibly empty) list of goals or $\bot_n$, representing a failure at what Rocq calls *failure level* n.

Tactics t are defined as follows:

$$
\begin{array}{llll}
t := a & \mid t\ ;\ t & \mid \texttt{first}\ [t \mid \cdots \mid t] & \mid T \\
\quad \mid \texttt{idtac} & \mid t\ ;\ [t \mid \cdots \mid t] & \mid \texttt{progress}\ t & \mid \texttt{fix}\ T\ t
\end{array}
$$

The variable a ranges over atomic tactics. The `idtac` does nothing. For semicolons, t_1 ; t_2 executes t_2 on all goals generated by t_1, while t ; [t_1 | ... | t_n] executes t_i on the ith goal generated by t. The tactical `first` behaves like the first tactic from its argument list that succeeds; it fails if they all fail. The `progress` tactical behaves like its tactic argument if it succeeds and changes (progresses) the goal, or fails otherwise. The fixpoint combinator `fix` T t, with bound tactic variable T, provides recursive tactics. We assume tactics are closed, with variables appearing within corresponding `fix` binders. Other useful tacticals can be derived [9] from these, such as `try` and `repeat`.

Beyond tactics, we also have sentences and scripts. A sentence is a tactic plus an annotation, where `all:`t means t is executed on all goals, and n`:`t means t is executed on the nth goal. Scripts are roughly lists of sentences, except that they can also contain curly braces that focus goals. Due to space constraints, we elide explanations about deautomation of sentences and scripts, since most of the interesting details occur at the tactic level.

5.2 Deautomation Algorithm

The algorithm has two parts: *treeification*, which captures the relevant information about the execution of the script in an intermediate tree representation, and *extraction*, which extracts the deautomated script from the tree. We start by focusing on non-failing scripts with only semicolons.

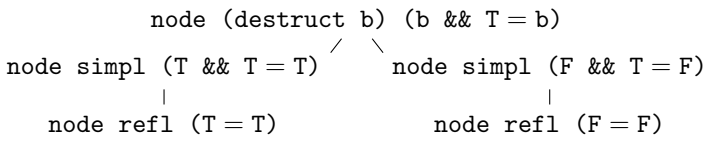
Treeification Trees are a useful intermediate representation during deautomation. For the moment, a tree r is defined as follows:

$$r := \texttt{hole}\ g \mid \texttt{node}\ a\ g\ r^\star$$

A `hole` represents an unsolved goal g, and a `node` represents the execution of an atomic tactic a on a goal g, with children $r^\star$ recursively representing executions on the goals produced by a on g.

The function `treeify` takes two inputs, a tactic t and a goal g, and proceeds by recursion on t. Conceptually, treeification parallels tactic execution. When executing t_1 ; t_2 on g, we execute t_1 on g, resulting in goals gs, then execute t_2 on each goal in gs. When treeifying, we first compute `treeify` t_1 g, resulting in tree r, then replace each `hole` g in r with the result of `treeify` t_2 g.

For example, treeifying the script `destruct b ; simpl ; reflexivity` on the introduction example `andb_true_r` would eventually result in this tree:



Extraction After constructing a tree from an automated script, we extract a deautomated script by traversing it in depth-first order, reading off the atomic tactic from each `node`.

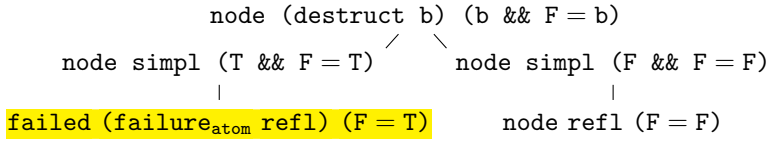
5.3 Deautomation with Failure Recovery

We extend the algorithm to support (1) deautomation of non-semicolon tacticals and (2) failing proofs.

Basics of Recovery We use `failed` in a `tree` to indicate that error e occurred at goal g, where e records the atomic tactic a that failed.

$$r := \ \ldots \mid \texttt{failed}\ e\ g \qquad e := \texttt{failure}_{\texttt{atom}}\ a$$

For example, suppose we made a mistake and instead tried to prove an erroneous lemma claiming `b && false = b`. Treeification will now construct the following tree:



As described in §4.2, we support recovering from failures on multiple branches, so we still reach the `reflexivity` on the right even though the `reflexivity` on the left failed. The final step is to extract an automation-less script. In the example, this is:

```
Proof.
  destruct b.
    simpl. Fail reflexivity. admit.
    simpl. reflexivity.
```

The `Fail` command on a tactic t succeeds if t fails, allowing the extracted script to communicate the failure that occurred without actually failing.

Recording the Initial Failure We alluded in §4.2 to the fact that the internal failure recovery of `first` interacts in complex ways with the external failure recovery of deautomation. In particular, `first` behaves differently depending on *how* the tactics within it fail.

Ltac has multiple *failure levels*, written $\bot_n$. If executing some tactic t within a `first` tactical fails with $\bot_0$, then the next tactic in the list provided to `first` is tried. If t fails with $\bot_{S(n)}$, then `first` itself fails, at level $\bot_n$. Hence, correctly deautomating `first` requires us to correctly handle failure levels throughout the deautomation algorithm.

To do so, we change the return type of `treeify` from `tree` to a new type constructor $R_{\texttt{treeify}}$, parameterized by a type x:

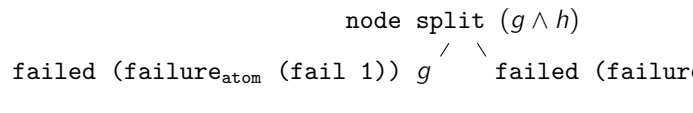
$$R_{\texttt{treeify}}\ x \ :=\ \texttt{yes}\ x \mid \texttt{recov}\ x\ n \mid \texttt{no}\ n$$

The new return type of `treeify` is $R_{\texttt{treeify}}$ `tree`. That is, there are three cases for what can happen during tree construction. The `yes` case says everything succeeded and returns a tree. The `recov` case says one or more failures occurred, but we were able to recover from these failures, so we can still return a tree; it also records the level n of the initial failure encountered. Finally, the `no` case says one or more failures occurred and we could not recover from the last one; it again records the level n of the initial failure.

We explicitly record the failure level, and specifically the level of the initial failure, in order to preserve the semantics of `first`. Why? If we have a tactic t where we encounter and recover from multiple failures when constructing a tree r, we cannot determine the initial failure in the original t from r alone. For example, consider the script

$$t\ :=\ \texttt{split ; [idtac | fail 0] ; [fail 1 | idtac]}$$

The `fail` n tactic fails on any goal with $\bot_n$. On goal $g \wedge h$, we construct this tree:



If t appears as an argument to `first`, we will need to know that it fails at $\bot_0$ instead of $\bot_1$, but this information is not apparent from the tree, since construction continued past the initial `fail 0` in the second branch until it encountered the `fail 1` in the first branch. To resolve this issue, we remember the initial failure level in the $R_{\texttt{treeify}}$ result type.

A subtlety remains: what if `first` is applied to an empty list of tactics? The semantics dictates that `first [ ]` should fail. We discuss how to handle this class of failure next.

Incorporating Tactic-Level Failures Up until now, our definition of errors e only included *atomic failure*, which represents an atomic tactic a failing on some goal. But not all failures can be localized to an atomic failure. For example, `first [ ]` fails, but there are no atomic tactics at all in this term. Tactic failures can also occur in t `;` `[`t_1 `|` ... `|` t_n`]`, when n does not match the number of goals generated by t, and `progress` t, when t succeeds but does not change the goal.

We therefore add a second kind of failure, *tactic failure*, to our definition of e, with constructor $\texttt{failure}_{\texttt{tac}}\ t$. Then, for `first [ ]`, we construct the tree $\texttt{failed (failure}_{\texttt{tac}}\ \texttt{(first [ ])) g}$.

One other tactic-level "failure" needs to be considered. Our language supports, through fixpoints, the possibility of non-terminating tactic execution. To ensure treeification terminates, we supply fuel to the algorithm, which decrements with each iteration. If fuel reaches zero, we output the tree `failed out-of-fuel g`. Incorporating this information into the tree allows us to retain the trace of tactics up until that point instead of failing globally.

5.4 Correctness

We can now check that deautomation obeys some desirable correctness properties.

We need a formal model of how Ltac scripts behave. For atomic tactics, we rely on the black-box

`run-atomic` function. For other tactics, we use the semantics from [9] (specifically the "Ltac — The Tactics" chapter). At a high level, execution of a tactic t on a goal g results in either a list of goals gs, which is empty if t solved g, or a failure state $\bot_n$.

This model of `Ltac` semantics is a simplified approximation of the actual `Ltac` semantics. In particular, we do not model *unification* or *backtracking*. These limitations are obvious directions for future work (§7). For now, however, our priority is not to model the full complexity of `Ltac`, but rather to carve out a subset that allows us to explore interesting questions about deautomation.

To reason about the correctness of deautomation, we begin with properties of treeification and extraction, then glue these results together into a theorem about the overall behavior of deautomation: deautomating a non-failing proof on a goal g will result in a proof that behaves the same as the original proof on g. For failing proofs, failure recovery intentionally results in an output that behaves differently from the original; however, we can instead prove that the extracted script executes *without* failing. All our proofs have been mechanized in Rocq.

5.5 Proof-of-Concept Implementation

We have implemented the theory above as a proof-of-concept `VS Code` extension that provides a concrete demonstration of our theoretical contributions and illustrates how deautomation might fit into an interactive programmer workflow.

With this extension, the user's proof is loaded into a side panel, and they see the "levers of control" described in §4.3. In particular, they can deselect tacticals to exclude them from deautomation. They can also opt to treat certain user-defined tactics as transparent, which inlines the body of that tactic during deautomation. (This feature is still quite preliminary: we only support user-defined tactics that are abbreviations — i.e., that do not have arguments — and that fall within our subset of `Ltac`.) After the user adjusts what they want to deautomate, the extension deuatomates the proof.

The implementation also contains some additional features, such as integration with Rocq's built-in `info_auto` for deautomating the `auto` tactic, and pretty-printing with bullets.

6 Related Work

Our goal in this paper has been to expand a proof script so as to provide more points in its execution where its behavior can be inspected. Prior work has addressed related goals. Our closest predecessors are Pons's Ph.D. thesis [10] and Adams's Tactician tool [11].

Pons presents in Section 4.3 of [10] an algorithm for tactic *expansion*. Tactic expansion transforms Rocq proof scripts containing semicolon t `;` t and branching t `; [`t `|` ... `|` t`]` tacticals into individual steps. For example, as per Appendix D of [10], the script `A; B; [C | D | E | F]; G.` would, in the appropriate context, be expanded into:

```
1: A. 1: B. 3: B. 1: C. 1: G. 1. D. 1: E. 1: F. 1. G.
```

Pons generates graphical visualizations of proof trees. He also shows how to modify the expansion algorithm to support failure localization by moving failing tactics to the end of the script.

There are many notable similarities between Pons's expansion and the deautomation explored here. Both achieve the effect of allowing the user to step through the individual tactics in their proof, and both consider the issue of handling failing proofs. Our work goes beyond Pons's in (1) supporting deautomation of several tacticals besides semicolon and branching — for example, as we have seen, tacticals such as `first` require especially careful consideration in the context of failure recovery — and (2) offering a more rigorous treatment of the deautomation procedure and its formal properties.

The Tactician tool [11] supports *unraveling* of HOL Light tactical connectives into a step-by-step proof. We share the broad approach of modeling a proof as a tree and constructing that tree by recording the behavior of tactics as they are applied. The main difference is that Tactician does not appear to support failure localization or recovery. Also, Tactician only discusses how to address HOL Light's equivalent of semicolon and branching tacticals. Conversely, Tactician's implementation is more robust than our current prototype.

Our work is also related to techniques that improve the visibility of intermediate proof states. Our approach is to transform the proof script in a way that introduces execution points, but there are other

ways to improve visibility. In particular, others have developed visual debuggers for tactics [12] and new tactic languages that afford inspection of the flow of subgoals [13], [14]. We see deautomation as a useful way to work with existing tactic languages, and as providing a kind of ready-made trace of what a proof does during a debugging session.

Even when users are shown the state of a proof, they still may need help understanding it. Robert's *PeaCoq* tool [15, Chapter 3] augments displays of proof obligations to highlight how those obligations have changed after the application of a tactic, particularly highlighting which obligations have been addressed and which have been introduced. Furthermore, as some in the proof assistant community have pointed out [16], [17], formal proofs can sometimes helpfully be augmented with diagrammatic notations, as in visualizations of heaps and hydra diagrams. One complement to proof deautomation might be toolkits for creating domain-specific visual descriptions of state, such as those already developed for the Lean proof assistant [18].

Deautomation aims to enable more efficient manipulation of a proof. The kinds of graphical editing [14] and drag and drop [19] interfaces proposed in the proof assistant literature could have a place in helping users reorder and restructure tactics in deautomated proofs. Interfaces from the broader interactive programming tools literature for exposing program state in-situ [20] and for debugging streams [21] could also accelerate inspection of subgoals around sites of automation.

Deautomation might be less necessary if proofs were made more robust to breaking changes that necessitate inspection. For instance, they could be updated with automatically generated patches as their specifications change [22]. We see deautomation as a complement to automated fixes, in the situations where a user by preference or circumstance cannot rely on automation to fix itself.

Our prior work on proof reading [5] loosely inspired our interest in investigating issues around understanding automation, though there is no direct overlap between the two works.

7 Future Work

Expanding the Scope of Deautomation We chose to support a subset of `Ltac`, focusing on a range of tacticals, and to employ a simplified model of `Ltac` semantics that treats atomic tactics and goals as opaque. This tightly defined scope serves as a rich starting point for establishing a core of what effective deautomation looks like, but it certainly should not be the endpoint.

One important avenue for future work would be how to deautomate proofs that contain backtracking, where a tactic failure later in a proof can trigger an alternative path being taken earlier in the proof. Another would be to support non-tactical `Ltac` machinery such as `match goal`. A third would be to look at `Ltac2`, a proposed successor to `Ltac`.

Reautomation The inverse of deautomation is *re*automation — that is, the process of rolling automation back up after the user has inspected and modified it. Notably, this is distinct from general utilities for automating proofs (e.g., [10], [11]), as a user of reautomation may wish that the reautomated proof preserves the design of their original automated script.

One challenge would be how to infer precisely what a user wants out of reautomation after they have edited a deautomated script. Consider the example from §3 once more. Suppose the user reviews the deautomated script, finds the bug, and fixes it on the third branch, but not the fourth. What is the right outcome of reautomation in this case? We could assume that the user intends to change the fourth branch in the same way. This would lead to a reautomated proof that retains the same structure and makes the modification on all failing branches. We could instead interpret the user's edits literally, where the reautomated proof now behaves differently in the third and fourth branches, perhaps by using the `; [ | | ]` construct. Reautomation would have to be designed in a way that correctly anticipates when a change is meant to be folded into additional branches.

Another challenge is mapping changes in a deautomated script to its automated form. To achieve this mapping, it is likely necessary to maintain a record linking expressions in the original script to the deautomated script. Edits to one script need to be mapped to the other. To do so in a coherent, composable way, it may require bidirectional programming approaches such as lenses [23].

References

[1] A. de Almeida Borges, A. C. Artís, J.-R. Falleri, *et al.*, "Lessons for interactive theorem proving researchers from a survey of coq users," in *14th International Conference on Interactive Theorem Proving (ITP 2023)*, Dagstuhl Publishing, vol. 268, 2023, pp. 1–18.

[2] C. D. Team, *The Coq proof assistant*, 1989-2024. [Online]. Available: http://coq.inria.fr.

[3] D. Delahaye, "A proof dedicated meta-language," *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 2, 2008.

[4] B. Beckert, S. Grebing, and F. Böhl, "A usability evaluation of interactive theorem provers using focus groups," in *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers 12*, Springer, 2015, pp. 3–19.

[5] J. Shi, B. Pierce, and A. Head, "Towards a science of interactive proof reading," Plateau Workshop, 2023.

[6] A. Blandford, D. Furniss, and S. Makri, *Qualitative HCI Research: Going Behind the Scenes* (Synthesis Lectures on Human-Centered Informatics). Morgan & Claypool Publishers, 2016.

[7] T. Bourke, M. Daum, G. Klein, and R. Kolanski, "Challenges and experiences in managing large-scale proofs," in *Intelligent Computer Mathematics: 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings 5*, Springer, 2012, pp. 32–48.

[8] B. C. Pierce, A. A. de Amorim, C. Casinghino, *et al.*, *Logical Foundations* (Software Foundations), B. C. Pierce, Ed. Electronic textbook, 2024, vol. 1, Version 6.7, http://softwarefoundations.cis.upenn.edu.

[9] W. Jedynak, "Operational semantics of Ltac," M.S. thesis, University of Wrocław, 2013.

[10] O. Pons, "Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs," Ph.D. dissertation, 1999.

[11] M. Adams, "Refactoring proofs with Tactician," in *Software Engineering and Formal Methods: SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY* SCART, York, UK, September 7-8, 2015. Revised Selected Papers*, Springer, 2015, pp. 53–67.

[12] J. Fehrle, "A visual Ltac debugger in CoqIDE," in *The Eigth International Workshop on Coq for Programming Languages*, 2022.

[13] C. S. Coen, E. Tassi, and S. Zacchiroli, "Tinycals: Step by step tacticals," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 2, pp. 125–142, 2007.

[14] G. Grov, A. Kissinger, and Y. Lin, "A graphical language for proof strategies," in *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings 19*, Springer, 2013, pp. 324–339.

[15] V. Robert, *Front-end tooling for building and maintaining dependently-typed functional programs*, 2018.

[16] C. Pit-Claudel, "Untangling mechanized proofs," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, 2020, pp. 155–174.

[17] S. Chiplunkar and C. Pit-Claudel, "Diagrammatic notations for interactive theorem proving," in *4th International Workshop on Human Aspects of Types and Reasoning Assistants*, EPFL, 2023.

[18] W. Nawrocki, E. W. Ayers, and G. Ebner, "An extensible user interface for lean 4," in *14th International Conference on Interactive Theorem Proving (ITP 2023)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[19] P. Donato, P.-Y. Strub, and B. Werner, "A drag-and-drop proof tactic," in *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2022, pp. 197–209.

[20] S. Lerner, "Projection boxes: On-the-fly reconfigurable visualization for live programming," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–7.

[21] N. Shrestha, T. Barik, and C. Parnin, "Unravel: A fluent code explorer for data wrangling," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 198–207.

[22] T. Ringer, N. Yazdani, J. Leo, and D. Grossman, "Adapting proof automation to adapt proofs," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018, pp. 115–129.

[23] J. N. Foster, "Bidirectional programming languages," English, ISBN: 9781109710137, Ph.D. University of Pennsylvania, United States – Pennsylvania, 2009. [Online]. Available: https://www.proquest.com/docview/304986072/abstract/11884B3FBDDB4DCFPQ/1 (visited on 11/22/2022).