Designing Proof Deautomation for Rocq

JESSICA SHI, University of Pennsylvania, USA
CASSIA TORCZON, University of Pennsylvania, USA
HARRISON GOLDSTEIN, University of Maryland, USA and University of Pennsylvania, USA
ANDREW HEAD, University of Pennsylvania, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

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 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.

CCS Concepts: • **Software and its engineering → Software verification and validation**; • **Human-centered computing → Interactive systems and tools**.

Additional Key Words and Phrases: proof assistants, proof automation, proof refactoring

ACM Reference Format:
Jessica Shi, Cassia Torczon, Harrison Goldstein, Andrew Head, and Benjamin C. Pierce. 2025. Designing Proof Deautomation for Rocq. In *Proceedings of the 15th PLATEAU Workshop on Programming Languages and Human-Computer Interaction (PLATEAU '25)*. 18 pages.

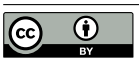
1 Introduction

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 [8] of users of the Rocq proof assistant [25], 46 percent of respondents had a doctoral degree.

Automation can greatly reduce the burden of proof assistant usage. However, automation also makes proving less interactive. In a non-automated proof, users can see the proof state at each intermediate step. In an automated proof, some (or even all) of these intermediate steps are elided, so the user is no longer able to see those intermediate states. Indeed, there is a fundamental tension between making proof steps visible and automating them away.

To reconcile this tension, 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.

Authors' Contact Information: Jessica Shi, University of Pennsylvania, Philadelphia, PA, USA; Cassia Torczon, University of Pennsylvania, Philadelphia, PA, USA; Harrison Goldstein, University of Maryland, College Park, PA, USA and University of Pennsylvania, Philadelphia, PA, USA; Andrew Head, University of Pennsylvania, Philadelphia, PA, USA; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, PA, USA.



PLATEAU '25, Boston, MA

Publication date: March 2025.

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 [9]. Concretely, we offer these contributions:

- We characterize users' needs around understanding automation, based on an interview study that we conducted with practicing Rocq users (§3).
- After introducing a motivating example of deautomation (§4), we discuss several design considerations for making deautomation informative and controllable (§5).
- We develop a deautomation algorithm that captures the execution of the original script and extracts a step-by-step version. We pay particular attention to the nuances of deautomating failing scripts. We prove (in Rocq) that deautomation preserves semantics up to failures. We also have a proof-of-concept implementation. Some technical details are provided in (§6).

We close with another example (§7), a survey of related work (§8), and ideas for future work (§9).

2 Background on Rocq

We begin with some background to orient readers unfamiliar with Rocq. To fully digest the paper's technical content, we recommend an in-depth introduction such as [17].

Tactics. Rocq proofs are written with *tactics*, commands that instruct the proof assistant on what the next step of the proof should be. Consider this proof script:

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

This example has tactics destruct, which does case analysis; simpl, which does simplification; and reflexivity, which solves trivial equalities. In words, this proof says: We proceed by case analysis on the boolean b. When b is true, after simplifying the equality, we see that it holds reflexively. When b is false, after simplifying the equality, we also see that it holds reflexively.

Interaction. We next describe the interaction model between the user and the proof assistant. In the example above, a user would typically write such a proof by checking the *proof state* after each step. 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 important 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.

Automation. Rocq also provides support for proof automation, including *tacticals*, which are higher-order tactics. For example, since both cases of the proof proceed in exactly the same way, the proof above can be written more succinctly using the semicolon tactical:

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

Semicolon sequences tactics, where a ; b applies b to every goal generated by a. Other automation constructs besides tacticals include search-based procedures that discharge certain classes of proof obligations entirely and language features that enable users to write their own tactics. For this work, we focus on deautomating tacticals.

3 Need-Finding Study

Prior studies have broadly suggested that users desire greater understanding about automated proofs [3, 23], 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 11 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 11 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 developments.

After the interviews, we performed a thematic analysis [4] 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, P7 recounted a time when they needed to fix proofs 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, for example, described how they sometimes needed to know which goals some tactic was failing to solve. P9, who showed proofs 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, automation such as semicolons and the "all:" selector break the interactive process of "run a tactic, see the result, run another one." Several other participants similarly 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 debugging, they would then fold the changes back into the original.) 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 [5, Section 2.3, for example].

Our study surfaced tensions between automation and interactivity, between automation and debugging, and, more broadly, between automation and understanding. To reconcile these tensions, we wish to transform deautomation from a tedious manual task to a smoother tool-assisted process.

4 Motivating Example

Consider this example of an automated proof, adapted from the exercise solutions to *Logical Foundations* [17]. There is no need to know how the proof works. Instead, observe only that the proof uses two kinds of tacticals: the semicolons described earlier, and the try tactical, which tries a tactic and does nothing if the tactic fails. We will discuss other relevant details as we go along.

```
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.
```

Suppose a user initially proved this theorem as an if-and-only-if statement, but they later realize only the forward direction is relevant to them. To simplify the development, they decide to change the theorem to a single implication. Refactoring the proofs is mostly straightforward — they just delete the proofs for the backward direction — but bizarrely, the forward direction now fails! 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?

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 says reflexivity is failing, the user starts by changing the last semicolon into a period, to try and see the place that fails. (We highlight the edit made in each step 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.                                    a1, a2 : aexp
  induction 1; simpl; intros;             n1, n2 : nat
  try (rewrite aevalR_aeval in H, H0;     H : aevalR a1 n1
       rewrite H; rewrite H0).            H0 : aevalR a2 n2
  3: {                                   ------------------------------------
    reflexivity.                          (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 semicolon and try tacticals.

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

```
0  Proof.
1    induction 1.
2    - simpl. intros. reflexivity.
3    - simpl. intros. reflexivity.
4    - simpl. intros.
5      (* tried and failed to run:
6          rewrite aevalR_aeval in H, H0. *)
7      Fail reflexivity. admit.
8    - simpl. intros.
9      (* tried and failed to run:
10         rewrite aevalR_aeval in H, H0. *)
11     Fail reflexivity. admit.
```

The user can immediately jump to the intermediate point in the third case where reflexivity is failing (Line 7). They can also see a trace (Lines 5–6) 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 why

the first two cases (Lines 2–3) succeed, they can readily inspect the proof steps there. Moreover, the user can also inspect the fourth case (Lines 8–11) and see that it is failing for the same reason.

5 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.

5.1 Desirable Outputs

Consider the example proof script from §2 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 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 script should resemble the format of a proof that a user would plausibly write when not using automation.

5.2 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 §4) 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 script preceding a failure and interact with the failing step. In §4, this context included 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 scripts. When users work with proofs where different branches — the cases in a proof by induction, for example — are explicit, they can choose 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 §4 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 $[t_1 \mid \ldots \mid t_n]$ 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.

5.3 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 what is deautomated.

Returning to §4, 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 custom tactics. The user should be able to decide whether to treat a custom tactic opaquely, as they would any built-in tactic, or transparently, which exposes it for deautomation.

6 Technical Details

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

6.1 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 atomic $\to$ goal $\to$ (list goal + $\bot_n$). That is, executing an atomic tactic on a goal returns either a list of goals or $\bot_n$, representing a failure at what Rocq calls *failure level n*.

Tactics t are defined as follows:

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

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 $t\,;[t_1 \mid \ldots \mid t_n]$ 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, with bound tactic variable T, enables recursive tactics. We assume tactics are closed, with variables appearing within corresponding fix binders. Other common tacticals can be derived [14] 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 generally elide explanations about deautomation of sentences and scripts, since most of the interesting details occur at the tactic level.

6.2 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.

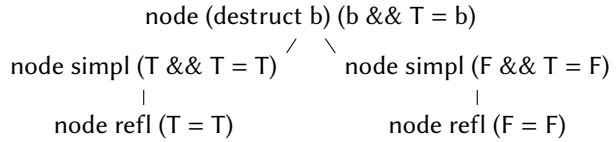
Treeification. Trees are a useful intermediate representation during deautomation. For the moment, a tree r is defined as follows:

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

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" in the context of the example andb_true_r from §2 would eventually result in this tree:



We use some abbreviations for formatting reasons: T for true, F for false, and refl for reflexivity. Appendix A gives a step-by-step construction of 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 tactic from each node.

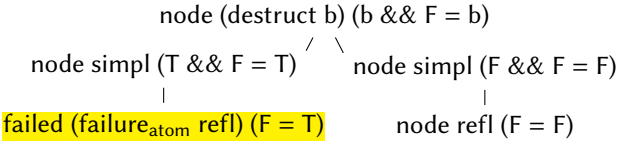
6.3 Deautomation with Failure Recovery

We extend the algorithm to support deautomation of non-semicolon tacticals and 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.

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

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



As described in §5.2, 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 §5.2 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*, which we write as $\bot_n$ for natural numbers n. If executing some tactic t within a first tactical fails at $\bot_0$, then the next tactic in the list provided to first is tried. If t fails at $\bot_{S(n)}$, then first itself fails at $\bot_n$. 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_{treeify}, parameterized by a type x:

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

The new return type of treeify is R_{treeify} tree. That is, there are three cases for what can happen during treeification. 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

$$t \quad := \quad \text{split}\,;[\text{idtac} \mid \text{fail 0}]\,;[\text{fail 1} \mid \text{idtac}]$$

The fail n tactic fails on any goal at $\bot_n$, and idtac does nothing. On goal $g \wedge h$, we construct:

$$\begin{array}{c} \text{node split}\ (g \wedge h) \\ \text{failed}\ (\text{failure}_{\text{atom}}\ (\text{fail 1}))\ g \quad\quad \text{failed}\ (\text{failure}_{\text{atom}}\ (\text{fail 0}))\ h \end{array}$$

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_{treeify} result type.

Recording the initial failure level allows us to deautomate first correctly, but there are also questions of how to deautomate first informatively. For example, suppose we have first $[t_1 \mid t_2]$, and t_1 fails while t_2 succeeds, so the overall tactic behaves like t_2. It would be correct to just return the deautomation of t_2, but it would be much more informative to also retain information about why t_1 failed. So, we add a new tree construct:

$$r \quad := \quad \ldots \mid \text{trace}\ (\text{list}\ r)\ r$$

The second argument to trace contains the tree for the last attempted tactic, while the first argument contains a list of the trees for the rest of the attempted tactics.

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 $t\,;[t_1 \mid \ldots \mid t_n]$, 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:

$$e \;:=\; \ldots \mid \mathsf{failure}_{\mathsf{tac}}\ t$$

Then, for first [ ], we construct the tree failed ($\mathsf{failure}_{\mathsf{tac}}$ (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 indicate in the tree an out-of-fuel error. Incorporating this information into the tree allows us to retain the trace of tactics up until that point instead of failing globally.

Incorporating Sentence- and Script-Level Failures. We mentioned previously that our grammar also includes sentences and scripts. We do *not* recover from sentence- and script-level failures, as they are orthogonal to our goal of recovering from failures that relate to tactic-based automation. We use the no case from the definition of R_{treeify} here.

Further Details. We provide some pseudocode for deautomating atomic tactics, semicolons, and first in Appendix B. The full algorithm (and proof of correctness, described shortly) is here[1].

6.4 Correctness

To reason about the correctness of our deautomation algorithm, we need a formal model of how Ltac scripts behave. For atomic tactics, we rely on a black-box run-atomic function. For other tactics, we use the semantics from [14, Chapter 6]. 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 (§9). 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.

Our proof begins with properties about treeification and extraction, then glues these together to prove this theorem: deautomating a non-failing proof on a goal g will output a proof that behaves the same as the original proof on g. For failing proofs, failure recovery intentionally outputs a proof that behaves differently from the original; however, we instead prove that the extracted script executes *without* failing. We mechanize these proofs in Rocq. Appendix C contains a proof sketch.

6.5 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 for control" described in §5.3. In particular, they can deselect tacticals to exclude them from deautomation. They can also opt to treat certain custom tactics as transparent, which inlines the body of that tactic during deautomation. (This feature is still quite preliminary: we only support custom tactics

[1] https://github.com/jwshii/deauto-artifact

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.

7 Advanced Example

We walk through a more advanced example of deautomation that exercises features not covered by §4. This proof is adapted from *Verified Functional Algorithms* [2]. Suppose a user is learning about binary-search tree proofs, and they encounter in their textbook this theorem:

```
Theorem lookup_insert_eq :
  ∀ (V : Type) (t : tree V) (d : V) (k : key) (v : V),
  lookup d k (insert k v t) = v.
Proof. induction t; intros; bdall.
```

The bdall tactic is defined to be

```
Ltac bdall := repeat (simpl; bdestructm; try lia; auto).
```

Note that the try tactical can be derived from first, and repeat from a combination of fix, progress, and try, so collectively, this proof script exercises most of our deautomation algorithm.

Given that they did not write this proof themselves, the user is not particularly confident about why it works, so they would like to be able to step through and examine the details. Turning to deautomation, they choose to make bdall transparent, so that they can deautomate its contents. They click "deautomate," and voilà!

```
Proof.
  induction t.
  - intros. simpl. bdestructm.
    + lia.
    + idtac. simpl. bdestructm.
      * lia.
      * simple apply @eq_refl.
  - intros. simpl. bdestructm.
    + idtac. simpl. bdestructm.
      * simple apply IHt1.
      * lia.
    + idtac. simpl. bdestructm.
      * idtac. simpl. bdestructm.
        -- lia.
        -- idtac. simpl. bdestructm.
           ++ simple apply IHt2.
           ++ lia.
      * idtac. simpl. bdestructm.
        -- lia.
        -- idtac. simpl. bdestructm.
           ++ lia.
           ++ simple apply @eq_refl.
```

The deautomated script immediately reveals much more information about the underlying structure of the proof. For example, it is apparent that the repeat in bdall is being put to good use, as the tactics within are invoked many times.

Beyond static information, the user can now step to intermediate goals they wish to inspect. For example, they may wonder what goals lia is solving. In the deautomated script, they can see

precisely the places where lia succeeds; jumping to those locations, they see that these are cases where there are contradictory assumptions (e.g., $k_0 < k$ and $k \geq k_0$).

Note also that, while the custom tactic bdestructm contains automation we do not support, this does not prevent deautomating the surrounding proof by simply continuing to treat it as opaque.

This example shows the complementary strengths of automated and deautomated proof scripts: automated scripts are succinct and powerful; deautomated scripts are flexible and informative.

8 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 PhD thesis [20] and Adams's Tactician tool [1].

Pons presents in [20, Section 4.3] an algorithm for tactic *expansion*. Tactic expansion transforms Rocq proof scripts containing semicolon $t\,;t$ and branching $t\,;[t \mid \ldots \mid t]$ tacticals into individual steps. For example, as per [20, Appendix D], 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 [1] 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 explicitly recovers intermediate proof steps, but there are other ways to improve visibility. In particular, others have developed visual debuggers for tactics [11] and new tactic languages that afford inspection of the flow of subgoals [7, 13]. 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 [22] 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 [6, 18], 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 [16].

Deautomation aims to enable more efficient manipulation of a proof. The kinds of graphical editing [13] and drag-and-drop [10] 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 [15] and for debugging streams [24] 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 [21]. 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.

Deautomation shares some conceptual similarities with work on resugaring [19], which also considers how traversing different levels of abstraction can affect user understanding.

9 Future Work

Evaluating Deautomation. An evaluative user study would be useful in verifying our claim that deautomation helps users better understand their automated proofs. Such a user study could quantitatively measure whether, for example, using our deautomation tool decreases the time spent on debugging tasks. It could also qualitatively describe how deautomation fits into users' workflows, and whether it affects the way users use automation.

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 considering backtracking and unification, which we discuss further in Appendix D. Another would be to support non-tactical Ltac machinery such as match goal. A third would be to look at Ltac2, a 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 [1, 20], 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 §4 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.

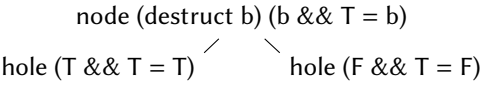
Another challenge is mapping changes in a deautomated script to its automated form. To achieve this mapping, it is likely necessary to maintain a record linking expressions in the original script to the deautomated script. Edits to one script need to be mapped to the other. To do so in a coherent, composable way, it may require bidirectional programming approaches such as lenses [12].

A Treeification Example

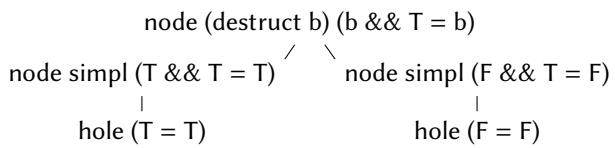
We show how the example tree from §6.2 is constructed incrementally. Recall that the proof is

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

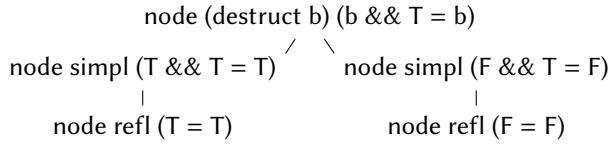
First, treeifying the atomic tactic "destruct b" on the initial goal would result in this tree:



...treeifying "destruct b ; simpl" would result in this tree:



...and treeifying the entire script would result in this tree:



B Pseudocode Algorithm

Recall that trees r and errors e are defined as follows:

$$
\begin{array}{lll}
r := & \mathsf{hole}\ g & \qquad e := \ \mathsf{failure}_{\mathsf{atom}}\ a \\
& \mid \ \mathsf{node}\ a\ g\ r^{\star} & \qquad\quad \mid \ \mathsf{failure}_{\mathsf{tac}}\ t \\
& \mid \ \mathsf{failed}\ e\ g & \qquad\quad \mid \ \text{out-of-fuel} \\
& \mid \ \mathsf{trace}\ (\mathsf{list}\ r)\ r &
\end{array}
$$

Recall that we use R_{treeify} in return types, defined as follows:

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

To sequence computations involving R_{treeify}, we define a monad instance, where

```
return x = yes x
mx >>= k = match mx with
           | yes x ⇒ k x
           | recov x n ⇒ match k x with
                         | yes x' ⇒ recov x' n
                         | recov x' _ ⇒ recov x' n
                         | no _ ⇒ no n
           | no n ⇒ no n
```

In the recov case, where a failure at level n already occurred, that level is retained by threading the initial level through the rest of the computation. We write let $x \leftarrow mx$ in $k\ x$ for $mx \gg\!= k$.

Treeification. We define the treeify function, which is defined mutually with $\mathsf{treeify}_{\mathsf{first}}$. There are also some auxiliary functions that we sketch at the end.

```
treeify : tactic → goal → R_treeify tree
treeify a g = match run-atomic a g with
              | gs ⇒ yes (node a g (map hole gs))
              | ⊥_n ⇒ recov (failed (failure_atom a) g) n
treeify (t1; t2) g = let r ← treeify t1 g
                     in applyAllTree (treeify t2) r
```

```
treeify (first ts) g = match treeify_first ts g with
                       | mrs · [mr] ⇒ let r ← mr, rs = getTrees mrs
                                      in return (trace rs r)
                       | [ ] ⇒ recov (failed (failure_tac t) g) 0

treeify_first : list tactic → goal → list (R_treeify tree)
treeify_first [ ] g = [ ]
treeify_first (t :: ts) g = match treeify t g with
                            | yes r ⇒ [yes r]
                            | recov r 0 ⇒ recov r 0 :: treeify_first ts g
                            | recov r S(n) ⇒ [recov r n]

applyAllTree : (goal → tree) → tree → tree
applyAllTree f r traverses r and replaces each hole g with the result of f g

getTrees : list (R_treeify tree) → list tree
getTrees mrs returns a list of the trees in mrs
```

Extraction. We next define the extract function, which receives as input the tree constructed by treeify. It outputs a list of tactics (i.e., the deautomated script) and also a list of admitted (unsolved) goals. Keeping track of admitted goals helps us state our correctness properties.

```
extract : tree → (list tactic, list goal)
extract (node a g rs) = let (gs , rs') = extractList rs in (a :: rs' , gs)
extract (hole g) = (admit. , [g])
extract (failed (failure t) g) = (Fail t. admit. , [g])

extractList : list tree → (list tactic, list goal)
extractList rs is essentially concat (map extract rs)
```

C Proof Sketch

We provide a sketch of the proof introduced in §6.4. It will be useful to distinguish between the *root goal* and the *leaf goals* of a tree, defined as follows for hole and node (other cases are analogous):

```
rootGoal (hole g) = g          leafGoals (hole g) = [g]
rootGoal (node _ g _) = g      leafGoals (node _ _ rs) = concat (map leafGoals rs)
```

We first show the result of treeification is *consistent* with the semantics of the original tactic.

Lemma 1. If execution of tactic t on goal g results in goals gs, then treeification of t for g results in yes r, where the leaf goals of r are gs. If execution of t on g results in failure $\bot_n$, then treeification of t for g results in recov r n or no n.

Next, treeification produces only *valid* trees, satisfying two conditions. First, for any node a g rs in the tree, the result of run-atomic a g must match the root goals of rs. Second, for any failed e g in the tree, the tactic that e says fails must indeed fail on g. We do not validate out-of-fuel errors.

Lemma 2. If treeification of t for g results in yes r or recov r n, then r is valid.

For extraction, we might expect that, if we extract script p from a tree r, then execution of p on the root goal of r results in the leaf goals of r. This is almost true, but not quite: since we use admits in the extracted script, we need to instead rely on the record of admitted goals.

Lemma 3. Given a valid tree r, if extracting r results in a script p and admitted goals gs, then execution of p on the root goal of r results in the empty list of goals, and the admitted goals gs are equal to the leaf goals of r.

(We could avoid the issue of admitted goals by, for example, offsetting tactics appearing after an unsolved goal, so "split. admit. reflexivity." would become "split. 2: reflexivity." However, since admit is already commonly used by users to mark unsolved goals, we chose to use it here too.)

We compose all these lemmas into a top-level theorem about deautomation of successful tactics:

Theorem. If execution of t on g results in goals gs, then

A. treeification of t for g results in yes r, and
B. if extraction on r results in a script p' and admitted goals gs', then execution of p' on g results in the empty list of goals, and $gs = gs'$.

Proof. By Lemma 1, treeification does result in yes r, and the leaf goals of r are gs. By Lemma 2, r is valid, so by Lemma 3, given extracted script p' and admitted goals gs', we know p' executes to the empty list of goals and the leaf goals of r are gs'. Transitively, $gs = gs'$. □

When tactic execution fails, if we recover and deautomate into some script p, then we can show this script executes without failing (though with some admits), allowing the user to step through the script to understand what went wrong.

D More Complex Ltac Semantics

Readers familiar with Rocq may be wondering how (a) backtracking and (b) unification of existential variables, which we do not consider in our simplified model of Ltac semantics, might interact with deautomation. We describe some initial thoughts on these issues.

Backtracking. The tactical first, which we do deautomate, can be thought of as providing a limited, local form of backtracking, where failures can cause additional tactics to be tried. As future work, we would want to incorporate explicit backtracking tacticals like +. Consider this example:

```
Inductive example_ind : Prop :=
| bad  : False → example_ind
| good : True  → example_ind.

Goal example_ind.
Proof. (apply bad + apply good); easy.
```

In the script above, bad is applied, which leads to a goal where easy fails. This failure triggers backtracking, so now good is applied, leading to a goal where easy succeeds.

This script behaves the same as

```
first [apply bad; easy | apply good; easy].
```

which we could deautomate into:

```
(* tried and failed to run: apply bad. easy. *)
apply good. easy.
```

Although backtracking tacticals would add a new layer of complexity to our deautomation theory, we have already built useful foundations around how to deautomate first.

Backtracking is also an effect that can be implemented internally in a tactic such as constructor. For example, suppose we have the same goal as above but with this proof:

```
Proof. constructor; easy.
```

The same general sequence of steps as above occurs, but now the backtracking is internal to constructor. Our current algorithm would erroneously output a script that behaves differently from the original. In fact, we cannot deautomate this proof — that is, we cannot get rid of the semicolon — without also unraveling the internal tactics tried by constructor.

But in our conception of deautomation, we do not peer inside of atomic, built-in tactics, so we may not actually want to deautomate such a proof. One approach to handling such situations is to dynamically *detect* when the deautomated script has in fact diverged in behavior from the original and inform the user. This detection should not preclude us from deautomating scripts with tactics like constructor in general, only those that rely on invisible backtracking.

Unification. Our model of Ltac semantics does not consider unification. However, we can still deautomate many proofs containing e* tactics that create existential variables. For example, we have no problem deautomating this proof

```
Goal ∃ x, x ≤ 0 ∧ x ≤ 1.
Proof.
  (* can be deautomated *)
  eexists. split; eauto.

  (* into *)
  eexists. split.
  - simple apply le_n.
  - simple apply le_S. simple apply le_n.
```

However, we have made the simplifying assumption that we can output the deautomated steps in "linear" order, so that tactics are applied on goals in the order the goals are generated. This causes us to incorrectly deautomate proofs such as this one, where the inequalities are swapped.

```
Goal ∃ x, x ≤ 1 ∧ x ≤ 0.
Proof.
  (* cannot be deautomated *)
  eexists. split; [ | eauto ]; eauto.
```

In this second proof, the ; [| eauto] not only solves the goal for the second inequality $x \leq 0$, but it also correctly instantiates the existential variable corresponding to x to be 0. In our current algorithm, the deautomated output would instead solve the goal for the first inequality $x \leq 1$ before the second, which incorrectly instantiates x to be 1, causing the second inequality to be unsolvable.

An alternative approach to deautomation might preserve the order of the automated script:

```
1: eexists. 1: split. 2: eauto. 1: eauto.
```

In fact this output resembles that of Pons [20]. While this approach would assist the particular issue of out-of-order existential variable unification, it may negatively impact the readability of deautomated scripts in general, as discussed in §5.1.

We would be interested to examine in future work how to balance these challenges of deautomated scripts being maximally useful versus handling out-of-order unification.

References

[1] Mark Adams. 2015. Refactoring proofs with Tactician. In *Revised Selected Papers of the Colocated Workshops of the International Conference on Software Engineering and Formal Methods*. doi:10.1007/978-3-662-49224-6_6

[2] Andrew W. Appel. 2024. *Verified Functional Algorithms*. Software Foundations, Vol. 3. Electronic Textbook. https://softwarefoundations.cis.upenn.edu/vfa-current/index.html

[3] Bernhard Beckert, Sarah Grebing, and Florian Böhl. 2015. A Usability Evaluation of Interactive Theorem Provers Using Focus Groups. In *Software Engineering and Formal Methods*. doi:10.1007/978-3-319-15201-1_1

[4] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. *Analysing Data: The Editor's Work.* Springer International Publishing, 51–60. doi:10.1007/978-3-031-02217-3_5

[5] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. Challenges and Experiences in Managing Large-Scale Proofs. In *Proceedings of the International Conference on Intelligent Computer Mathematics.* doi:10.1007/978-3-642-31374-5_3

[6] Shardul Chiplunkar and Clément Pit-Claudel. 2023. Diagrammatic Notations for Interactive Theorem Proving. In *The International Workshop on Human Aspects of Types and Reasoning Assistants.* doi:10.5075/epfl-SYSTEMF-305144

[7] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. 2007. Tinycals: Step by Step Tacticals. In *Electronic Notes in Theoretical Computer Science.* doi:10.1016/j.entcs.2006.09.026

[8] Ana de Almeida Borges, Annalí Casanueva Artís, Jean-Rémy Falleri, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann. 2023. Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users. In *Proceedings of the International Conference on Interactive Theorem Proving.* doi:10.4230/LIPIcs.ITP.2023.12

[9] David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning.* doi:10.1007/3-540-44404-1_7

[10] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A Drag-and-Drop Proof Tactic. In *Proceedings of the International Conference on Certified Programs and Proofs.* doi:10.1145/3497775.3503692

[11] Jim Fehrle. 2022. A Visual Ltac Debugger in CoqIDE. In *The International Workshop on Coq for Programming Languages.* https://popl22.sigplan.org/details/CoqPL-2022-papers/1/A-Visual-Ltac-Debugger-in-CoqIDE

[12] John Nathan Foster. 2009. *Bidirectional Programming Languages.* Ph. D. Dissertation. https://www.proquest.com/docview/304986072/abstract/11884B3FBDDB4DCFPQ/1

[13] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. 2013. A Graphical Language for Proof Strategies. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning.* doi:10.1007/978-3-642-45221-5_23

[14] Wojciech Jedynak. 2013. *Operational Semantics of Ltac.* Master's thesis.

[15] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the Conference on Human Factors in Computing Systems.* doi:10.1145/3313831.3376494

[16] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. 2023. An Extensible User Interface for Lean 4. In *Proceedings of the Internaitonal Conference on Interactive Theorem Proving.* doi:10.4230/LIPIcs.ITP.2023.24

[17] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2024. *Logical Foundations.* Software Foundations, Vol. 1. Electronic Textbook. http://softwarefoundations.cis.upenn.edu

[18] Clément Pit-Claudel. 2020. Untangling Mechanized Proofs. In *Proceedings of the International Conference on Software Language Engineering.* doi:10.1145/3426425.3426940

[19] Justin Pombrio. 2018. *Resugaring: Lifting Languages through Syntactic Sugar.* Ph. D. Dissertation. https://github.com/justinpombrio/thesis

[20] Olivier Pons. 1999. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs.* Ph. D. Dissertation. https://cedric.cnam.fr/~pons/PAPERS/these.pdf

[21] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting Proof Automation to Adapt Proofs. In *Proceedings of the International Conference on Certified Programs and Proofs.* doi:10.1145/3167094

[22] Valentin Robert. 2018. *Front-end tooling for building and maintaining dependently-typed functional programs.* Ph. D. Dissertation. https://escholarship.org/uc/item/9q3490fh

[23] Jessica Shi, Benjamin Pierce, and Andrew Head. 2023. Towards a Science of Interactive Proof Reading. In *Plateau Workshop.* doi:10.1184/R1/22277317.v1

[24] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A Fluent Code Explorer for Data Wrangling. In *Proceedings of the Symposium on User Interface Software and Technology.* doi:10.1145/3472749.3474744

[25] Rocq Development Team. 1989–2025. The Rocq Prover. https://rocq-prover.org/