Designing Proof Deautomation for Coq

Jessica Shi
University of Pennsylvania
Philadelphia, PA, USA
jwshi@seas.upenn.edu

Cassia Torczon
University of Pennsylvania
Philadelphia, PA, USA
ctorczon@seas.upenn.edu

Harrison Goldstein
University of Maryland
College Park, MD, USA
University of Pennsylvania
Philadelphia, PA, USA
me@harrisongoldste.in

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@cis.upenn.edu

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, either to understand how the proof works or to pinpoint where it has broken. 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 — and we propose a deautomation algorithm and a proof-of-concept implementation.

1 Introduction

Automation is a powerful weapon in the fight against the overwhelming minutiae of interactive proof. Proofs that rely on automation are written at a higher level of abstraction, improving concision and robustness. In Coq [29], automation constructs include higher-order tactics (tacticals), `auto` and its relatives, and user-defined custom tactics.

However, automation also makes proving less interactive. When working with little or no automation, users can see the proof state at each step. With more automation, single steps become leaps and bounds, eliding many intermediate states that the user might need to understand how their proof works or why it doesn't. Indeed, there is a fundamental tension between making intermediate steps visible and automating them away.

To reconcile this tension, we describe a *proof deautomation* mechanism that reifies the underlying steps taken by 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. We also devote special attention to ensuring that broken proofs can be deautomated and annotated to support rapid recognition of causes of failure.

Our conception of deautomation is similar to the ones embodied in the Tactician tool [1] for HOL Light, which unfolds semicolon-chains to help with proof refactoring and understanding, and in an algorithm for Coq scripts described in Olivier Pons's Ph.D. thesis [24]. Our deautomation procedure goes further, however, handling a wider range of automation constructs and focusing attention on how these constructs affect deautomation of failing proofs.

We study deautomation on a core set of automation features drawn from Coq's `Ltac` language [11]. Concretely, we offer these contributions:

- We characterize users' needs around understanding proof automation, through observations from an interview study with practicing Coq users (§2).
- After introducing a motivating example (§3), we discuss several design considerations for deautomation (§4). In particular, deautomation should be informative even for failing proofs.
- We develop an algorithm for deautomation that captures the execution of the original script and extracts a step-by-step view of the proof, and we establish its fundamental correctness properties (§5).

We describe an implementation of our theory as a proof-of-concept VS Code extension (§6). We close with an extended example (§7), a survey of related work (§8), and a discussion of limitations (in particular, around backtracking and existential variables) and ideas for future work (§9).

2 Need-Finding Study

Prior studies have broadly suggested that users desire greater understanding about proof automation [4, 27], but has 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. In human-oriented programming-languages work, such studies both document the status quo and illuminate opportunities to better help users do what they need [8].

Our study consisted of interviews with eleven participants (a typical sample size for an initial need-finding study of a human-facing programming system [3, 17, 19, 20, 30]). 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 Coq 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 consistent 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 Example

Consider this example of an automated proof, adapted from the exercise solutions to *Logical Foundations* [22]. 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?

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:

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

They can now see in full the proof state that reflexivity is failing to execute on.

```
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 — to figure out what was happening.

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_eval in H, H0. *)
    Fail reflexivity. admit.                       F
  - simpl. intros.
    (* tried and failed to run:
        rewrite aevalR_eval 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

We next describe some of the design considerations that arise when deciding how to support deautomation. §4.1 and §4.2

foreshadow some technical challenges regarding deautomating failing proofs and reasoning about the correctness of deautomation, which we expand on in §5, while §4.3 mentions some interactive features needed in a tool implementation of deautomation.

4.1 Treatment of Failures

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 some kind of *failure recovery* — deautomating a failing proof should provide an informative result.

Informative Failure Recovery. 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 the §3 example, 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. Again, the §3 example is consistent with this aim, where the user could choose to examine either of the failing branches, or both.

After a tactic fails on a branch, we do not necessarily want to continue executing the tactics that follow in, say, a semicolon sequence, as these might be intended for other goals. To navigate these subtleties, we adopt the convention that we do not continue beyond a failure *on the branch that failed*, but we do continue with other branches.

Enriching the Space of Failures. One common situation involving failure is when tactic execution fails on an intermediate step and this also causes the failure of the surrounding tactic expression. As described previously, deautomation should assist with such cases by recording the failure and continuing onto other branches.

But not all tactic execution failures cause the surrounding expression to fail. In particular, consider the tactical 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 thought about how these should interact.

Moreover, even if a top-level tactic script does not fail outright, the user might still benefit from information about internal failures. In §3, the try portion of the script does not fail outright, but the internal failure of the rewrite is extremely pertinent to debugging. To capture this, we chose to have the deautomated output include a trace of what was tried and failed, so that the user can have the relevant details if this behavior was unintended.

Failure recovery affects the entirety of our design, from the motivating examples that we have shown and will show, to our conception of what it means for deautomation to have worked correctly, and to, of course, the technical details of our deautomation algorithm.

4.2 Notions of Correctness

Informally, deautomation is correct when the deautomated script behaves the same as the original. How do we formalize this notion?

We can define the semantics of a proof script as a function from goals to results, where a result is either a list of goals or a failure. But we would not expect deautomation to preserve this semantics, because deautomation must be performed *relative to a specific goal*, so the result of deautomating the same automated script might look different on different goals. For example, induction 1 ; t would be deautomated differently depending on how many goals are generated by induction. Hence, our consideration of what it means for deautomation to preserve semantics must also be relative to a specific goal.

Furthermore, while we do expect deautomation to be semantics preserving for *successful* proof scripts, we should not have this expectation for failing scripts. By supporting failure recovery, deautomation purposefully alters the behavior of failing scripts to allow the user to step through the output.

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 the script from §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 include individual invocations of simpl and intros in the deatuomated script, they may prefer to deselect the corresponding semicolons to opt them out of deautomation:

```
induction 1 ; simpl ; intros ;
```

Then, the deautomated 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 Approach

With these design considerations in mind, we next introduce a formal theory of deautomation.

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 atomic $\rightarrow$ goal $\rightarrow$ (list goal + $\bot_n$). That is, executing an atomic tactic on a goal will either result in a (possibly empty) list of goals or it will return $\bot_n$, representing a failure at what Coq calls *failure level* n.

Tactics t are defined as follows:

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

The variable a ranges over atomic tactics. The idtac tactic 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 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 [16] from these, e.g.:

$$
\begin{aligned}
\text{try } t &:= \text{first } [t \mid \text{idtac}] \\
\text{repeat } t &:= \text{fix } T \; (\text{try } (\text{progress } t \,;\, T))
\end{aligned}
$$

Beyond tactics, we also have sentences s and scripts p:

$$
\begin{aligned}
s &:= \text{all: } t \;\mid\; n\text{: } t \\
p &:= [\,] \;\mid\; s :: p \;\mid\; \{\, p \,\}\, p
\end{aligned}
$$

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. A bare sentence, without annotations, is syntactic sugar for 0: t. (Coq actually 1-indexes goals, but we 0-index here to make some technical details cleaner.) Scripts can be empty, begin with a sentence, or begin with a *focus block*, a script between curly braces. Execution of this block proceeds as if there is only the first goal, which must be solved before the closing brace.

5.2 Deautomation Algorithm

We separate deautomation into 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 explaining how to deautomate simple ;-scripts like this one:

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

Without the complications of non-semicolon tacticals or failure recovery, the process is straightforward. §5.3 adds these refinements.

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. For an atomic tactic a, we define (in pseudocode):

$$
\begin{aligned}
\text{treeify } a \; g \; = \; & \text{node } a \; g \; (\text{map hole } gs) \\
& \text{when } gs := \text{run-atomic } a \; g
\end{aligned}
$$

That is, we record the result of executing a on g as a node whose children are holes representing the yet-unsolved goals gs. In the example andb_true_r above, treeifying the atomic tactic "destruct b" on the initial goal would result in this tree:

node (destruct b) (b && T = b)
hole (T && T = T) ⁄ ＼ hole (F && T = F)

(To save space, we abbreviate true and false as T and F.)

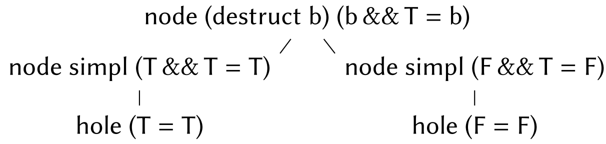
For semicolons, we define:

$$
\begin{aligned}
\text{treeify } (t_1 \,;\, t_2) \; g = \; & \text{let } r := \text{treeify } t_1 \; g \\
& \text{in applyTree (repeat (treeify } t_2) \; |r|) \; r
\end{aligned}
$$

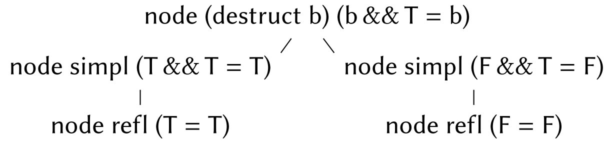
We will explain this in just a moment; conceptually, though, treeification parallels tactic execution. When executing t_1 ; t_2 on a goal g, we execute t_1 on g, resulting in goals gs, then execute t_2 on each goal in gs. When treeifying, we first compute treeify t_1 g, resulting in tree r, then replace each hole g in r with the result of treeify t_2 g.

The function applyTree does this second step; its type is list (goal → tree) → tree → tree. It traverses its argument tree from left to right, applying the first function from the given list of functions to the first hole it encounters, the second function to the second hole, and so on. In the pseudocode above, repeat f n indicates a list of f repeated n times, and $|r|$ is the number of goals in r.

Continuing with the example, treeifying "destruct b ; simpl" would thus result in this tree...



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



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 tactics from each node. For the example, the result is:

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

We show pseudocode for this shortly.

5.3 Deautomation with Failure Recovery

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

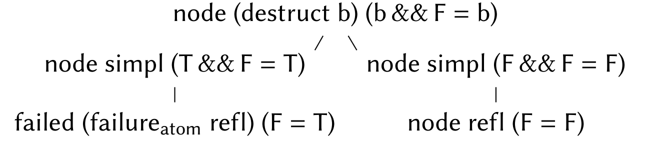
Basics of Recovery. We use failed in a tree to indicate that error e occurred at goal g, where e records the atomic tactic a that failed.

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

For example, suppose we made a mistake in the example above and instead tried to prove this erroneous lemma:

```
Lemma andb_false_r (b : bool) : b && false = b.
Proof. destruct b; simpl; reflexivity.
```

Treeification should construct the following tree:



We extend the definition of the function treeify on atomic tactics with an additional case where run-atomic fails (disregarding the failure level n for now).

$$\text{treeify } a \ g = \text{failed } (\text{failure}_{\text{atom}} \ a) \ g \quad \text{when } \bot_n := \text{run-atomic } a \ g$$

The semicolon case of treeify does not need to be modified, since any failures are handled by the atomic case.

As explained in §4, we support recovering from failures on multiple branches, but we do not continue past a failure on any branch that failed. Concretely, we accomplish this by extending the definition of applyTree to skip over functions in its input list that correspond to goals labeled failed.

As before, 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.

The extract function on a tree outputs a list of atomic tactics and a list of admitted goals. (Keeping track of admitted goals helps state the properties in §5.4.)

$$\text{extract (node } a \ g \ rs) = \ (\text{a :: concat (map extract } rs), [\,])$$
$$\text{extract (hole } g) = (\text{ admit. }, [g]\ )$$
$$\text{extract (failed (atomic}_{\text{fail}} \ a) \ g) = (\text{ Fail } a\text{. admit. }, [g]\ )$$

Here :: is the "cons" operator on lists.

Recording the Initial Failure. We alluded in §4 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_{treeify}, parameterized by a type x:

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

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

The two cases for atomic tactics a — where execution succeeds and where it fails — are rewritten to use the yes and recov constructors of R_{treeify}:

$$\begin{aligned}\text{treeify}\ a\ g\ &= \text{yes (node}\ a\ g\ \text{(map hole}\ gs))\\ &\quad \text{when}\ gs := \text{run-atomic}\ a\ g\\ \text{treeify}\ a\ g\ &= \text{recov (failed (failure}_{\text{atom}}\ a)\ g)\ n\\ &\quad \text{when}\ \bot_n := \text{run-atomic}\ a\ g\end{aligned}$$

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

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

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

$$\begin{array}{ccc} & \text{node split}\ (g \wedge h) & \\ & / \ \backslash & \\ \text{failed (failure}_{\text{atom}}\ \text{(fail 1))}\ g & & \text{failed (failure}_{\text{atom}}\ \text{(fail 0))}\ h\end{array}$$

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.

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

$$\begin{aligned}\text{return}\ x &:= \text{yes}\ x\\ mx \gg\!= k &:= \text{match}\ mx\ \text{with}\\ &\quad |\ \text{yes}\ x \Rightarrow k\ x\\ &\quad |\ \text{recov}\ x\ n \Rightarrow \text{match}\ k\ x\ \text{with}\\ &\qquad\qquad |\ \text{yes}\ x' \Rightarrow \text{recov}\ x'\ n\\ &\qquad\qquad |\ \text{recov}\ x'\ _ \Rightarrow \text{recov}\ x'\ n\\ &\qquad\qquad |\ \text{no}\ _ \Rightarrow \text{no}\ n\\ &\quad |\ \text{no}\ n \Rightarrow \text{no}\ n\end{aligned}$$

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

Using this monad instance, treeifying semicolons now looks like this:

$$\begin{aligned}\text{treeify}\ (t_1\ ;\ t_2)\ g = \text{let}\ r \leftarrow \text{treeify}\ t_1\ g&\\ \text{in applyTree (repeat (treeify}\ t_2)\ |r|)\ r&\end{aligned}$$

The only difference from the previous version is that the := in the let-binding has become a monadic bind. Analogous modifications are needed in applyTree.

For first, we use an auxiliary function $\text{treeify}_{\text{first}}$ to iterate through the list of tactics.

$$\begin{aligned}&\text{treeify}_{\text{first}} : \text{list tactic} \to \text{goal} \to \text{list}\ (R_{\text{treeify}}\ \text{tree})\\ &\text{treeify}_{\text{first}}\ [\,]\ g := [\,]\\ &\text{treeify}_{\text{first}}\ (t :: ts)\ g := \text{match treeify}\ t\ g\ \text{with}\\ &\quad |\ \text{yes}\ r \Rightarrow [\ \text{yes}\ r\ ]\\ &\quad |\ \text{recov}\ r\ 0 \Rightarrow \text{recov}\ r\ 0 :: \text{treeify}_{\text{first}}\ ts\ g\\ &\quad |\ \text{recov}\ r\ S(n) \Rightarrow [\ \text{recov}\ r\ n\ ]\end{aligned}$$

The output of this function is a list of the results of treeifying each tactic in the input. We use the recorded failure level to determine whether or not to continue onto the next tactic or to terminate. The key step is the last one: consistent with first, when we encounter failure $\bot_{S(n)}$, we do not continue onto the next tactic and instead return that the first as a whole has failed at $\bot_n$. (The no cases are analogous to recov.)

In the main treeify function, we need to collapse the list returned from $\text{treeify}_{\text{first}}$ back into a single result, which turns out to be a little tricky. The most straightforward approach would be to take the last element of the list

$$\begin{aligned}\text{treeify (first}\ ts)\ g := \text{match treeify}_{\text{first}}\ ts\ g\ \text{with}&\\ |\ mrs \cdot [mr] \Rightarrow mr&\end{aligned}$$

(where $\cdot$ is list-append). But this is unsatisfying because it throws away the information in mrs. If the tactic t in, for example, first [t | idtac] fails, we might want to know why it failed. To retain this information, we add a new construct to trees:

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

All of the trees in a trace should have the same goal g at their root. Then, if we have function getTrees : list (R_{treeify} tree) $\to$ list tree, we can replace the case above with

$$\begin{aligned}|\ mrs \cdot [mr] \Rightarrow \text{let}\ r \leftarrow mr,\quad rs := \text{getTrees}\ mrs&\\ \text{in return (trace}\ rs\ r)&\end{aligned}$$

One 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, with constructor $\text{failure}_{\text{tac}}\ t$. Then, for first [ ], we construct the tree failed ($\text{failure}_{\text{tac}}$ (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 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.

Lifting to Sentences and Scripts. While it makes sense to talk about tactics being executed on a single goal, intermediate sentences and script chunks are often executed on multiple goals. For treeification, we cannot directly pass in, say, a list of goals, because the tree structure requires that we maintain context about where goals originate and the relationships between them. Instead, treeify on sentences takes a sentence and a tree as inputs, and likewise for scripts. Their output is still a tree.

We treeify sentences by combining two previous definitions: treeify for treeifying tactics and applyTree for applying a list of functions on the unsolved goals in a tree. For sentences starting with all:, this gives:

$$\text{treeify (all: } t\text{) } r = \text{applyTree (repeat (treeify } t\text{) } |r|\text{) } r$$

That is, we replace each unsolved goal in r with the treeification of t for that goal. Atomic- and tactic-level failure recovery now smoothly transfers to sentences: if t fails on a goal in r, the failure is recorded on that branch, but other branches proceed as usual.

For sentences starting with n:, we need to account for the case where the selector n is out of bounds. To ponder how we might handle this failure, observe that each point in a tree (a hole, node, failed, or trace) corresponds to exactly one goal. It would not make sense to try to encode an out-of-bounds error, which is inherently with respect to a list of goals, within the tree. Accordingly, we *do not* recover from sentence- and script-level failures. This decision aligns with our focus on failure recovery *in the context of automation* and the fact that we focus on automation at the level of tactics. So, we have

$$\begin{aligned}\text{treeify (} n\text{: } t\text{) } r = \ & \text{if } n > |r| \\ & \text{then no } 0 \\ & \text{else applyTree ([id, } \dots\text{, treeify } t, \dots\text{, id]) } r\end{aligned}$$

where we pass to applyTree a containing treeify t at the nth position and identity functions elsewhere. In the out-of-bounds case, we use the no constructor from R_{treeify}.

5.4 Correctness

With all the pieces in place, we can now check that deautomation obeys some desirable correctness properties.

Baseline Model of Ltac Semantics. We will want to be able to establish some notion that a deautomated script behaves like the original. In order to conduct such reasoning, we need to have 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 [16] (specifically the "Ltac — The Tactics" chapter). We extend the semantics in [16] to sentences and scripts.

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 $\bot_n$. Execution of sentences and scripts are analogous, but relative to a starting list of goals.

Our model of Ltac semantics is a simplified approximation of the actual Ltac semantics. In particular, we do not model *unification*: goals are opaquely represented, and there is no provision for tactic execution on one goal to affect another goal. The effect on deautomation is that we cannot in general deautomate scripts that, due to existential variable instantiation, rely on goals being solved out of the order they are generated.

Another limitation is that we do not model *backtracking*. Some atomic tactics support backtracking; for example, the "constructor" tactic in a script such as "constructor ; t" may attempt multiple different constructors if t fails. We cannot deautomate proof scripts that rely on backtracking.

These limitations are obvious directions for future work, as we discuss in §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. This includes both how to design informative failure recovery, as we saw above, and how to formalize the properties deautomation should obey, as we shall see next.

Preservation of Meaning. 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 still prove some weaker properties.

It will be useful to distinguish between the *root goal* and the *leaf goals* of a tree, defined as follows:

$$\begin{aligned}&\text{rootGoal (hole } g) = g && \text{leafGoals (hole } g) = [g] \\ &\text{rootGoal (node _ } g \text{ _)} = g && \text{leafGoals (node _ _ } rs) = \\ & && \quad \text{concat (map leafGoals } rs)\end{aligned}$$

(The statements that follow are formulated at the level of tactics. Sentences and scripts are discussed at the end. All the proofs have been mechanized in Coq; however we

were not looking to create a fully verified implementation: the mechanization is not connected to the proof-of-concept implementation described in §6.)

We start by showing 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 equal the root goals of rs. Second, for any failed $e\ g$ in the tree, the error described by e must occur at g. That is, if e is $\text{failure}_{\text{atom}}$ a, then run-atomic $a\ g$ should fail, and if e is $\text{failure}_{\text{tac}}$ t, then execution of t on g should fail. 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 [ ] 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.

To lift these lemmas and the final theorem to scripts, recall that treeification on scripts takes a tree as input. But extraction makes no distinction between tactics already in the tree prior to treeification and tactics added afterwards. So we need to specialize the final theorem to a singleton list of goals $[g]$. In practice, this means we cannot start deautomation "mid-proof." The levers from §4.3 give users more control over what to deautomate.

6 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. A video figure demonstrating this interaction on examples appears in the supplemental material.

How it Works. The implemented deautomation algorithm closely follows the structure of the algorithm from §5. It rests on a few lower-level components.

Parsing. The extension parses the proof script into atomic tactics and tacticals. The current prototype implements a custom parser for a subset of Coq scripts; a future implementation should rely on Coq's own parser to ensure feature parity.

Proof tree construction. The deautomation algorithm relies on a black-box run-atomic function. We implement this by running coqtop and asking it to execute each atomic tactic on the appropriate goal. This has been sufficient for prototyping purposes, although integrating with the Coq API would likely be more robust.

We also support deautomation of auto by replacing it with the output of info_auto, and likewise with eauto and info_eauto.[1] Just as with the tacticals, the user can choose to deselect auto and eauto to opt them out of deautomation.

Traces. When rendering the deautomated script, we extract the traces recorded in the tree from first (and try) as comments on failing branches, to help the user debug.

[1]Readers familiar with the similarly named Info command, which prints the tactics that were executed by some more complex tactic expression, might wonder how its functionality compares with deautomation. While Info aligns with deautomation in limited situations, it does not unpack semicolons, and it does not output information when execution fails.

Pretty printing. Our pretty-printer also adds bullets to delineate the branches of the deautomated script. The resulting extraction produces the examples in §3 and §7.

7 Extended Example

We conclude with another example of deautomation that illustrates some of the features discussed in §5. This example is adapted from *Verified Functional Algorithms* [2], a textbook in the *Software Foundations* series.

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 in the textbook to be

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

Recall that we derive try 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!

```
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 has automation we do not support, namely match goal, 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 Ph.D. thesis [24] and Adams's Tactician tool [1].

Pons presents (in Section 4.3 of [24]) an algorithm for tactic *expansion*. Tactic expansion transforms Coq proof scripts containing semicolon $t\,;t$ and branching $t\,;[t\,|\ldots|\,t]$ tacticals into individual steps. For example (in Appendix D of [24])

```
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 (e.g. on p. 63 of [24]). 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 notion of 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 introduces execution points, but there are other ways to improve visibility. In particular, others have developed visual debuggers for tactics [13] and new tactic languages that afford inspection of the flow of subgoals [10, 15]. 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 [26, 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 interactive theorem proving community have pointed out [9, 23], 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 [21]. The CtCoq system [5] supports state visualization and "pseudo natural language" explanation of proofs.

Deautomation aims to enable more efficient manipulation of a proof. The kinds of graphical editing [15] and drag and drop [12] interfaces proposed in the interactive theorem proving literature could have a place in helping users reorder and restructure tactics in deautomated proofs. We anticipate that interfaces from the broader interactive programming tools literature for exposing program state in-situ [18] and for debugging streams [28] could 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 [25]. 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.

9 Future Work

9.1 Expanding the Scope of Deautomation

In this paper, 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. We discuss in this section how we might expand the scope in the future.

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.
```

The + tactical allows cross-semicolon backtracking. In the script above, bad is applied, which leads to a goal where easy fails. This failure triggers backtracking, so that 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 steps of the deautomated script 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 ≤ 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 ≤ 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 [24]. While this approach would assist the particular issue of out-of-order existential variable unification, it may negatively impact the readability of deautomated outputs in general. For example, if we tried to reformat the andb_true_r example this way, we might get:

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

Even in this small example, it is more challenging to see the structure of the deautomated proof — in particular, what the "branches" of the proof are and what tactics are applied to which branch. This is further complicated by the fact that the *n*: selectors are re-indexed as goals get solved.

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.

More of Ltac. In this paper, we support just a subset of Ltac, focusing on tacticals. One important feature to be added is match goal (and variants). We could consider match goal in two parts: the pattern-matching machinery, which determines what branches match, and the failure-recovery machinery, which tries a new branch if one fails. The pattern matching part will be new, but the failure recovery part ties in closely with what we know about deautomating first.

We also described our preliminary support for deautomating a very limited class of user-defined tactics. In the future, we would want to additionally support tactics that take arguments, recursive tactics, and tactics with more advanced functionality, such as generating fresh names.

Ltac2. We worked with Ltac because it is still in widespread use, but we are interested in exploring Ltac2 in future work. In fact, the backtracking primitives zero, plus, and case seem like a compelling starting point for determining how deautomation might compositionally support proofs that rely on backtracking. Besides backtracking, since many of the tactics and tacticals of Ltac were carried over to Ltac2, the portability of our deautomation should also benefit from the strong similarities between the languages.

9.2 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., [1, 24]), as a user of reautomation may wish that the reautomated proof preserves the design of their original automated script.

Why might this be difficult? For one reason, it may be difficult 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? One option is to 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. Another option is to 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 [14].

10 Conclusion

As our need-finding study shows, there is a tension between automation and interactivity: automation comes at the expense of easy access to a proof's intermediate state. But with appropriate tooling, automation and interactivity can coexist harmoniously, enabling users to examine mechanized proofs at a level of detail that suits their needs. This work on deautomation describes one model of such tooling, fit to a core set of tacticals from Coq and robust to failures.

References

[1] Mark Adams. 2015. 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, 53–67.

[2] Andrew W. Appel. 2024. *Verified Functional Algorithms*. Software Foundations, Vol. 3. Electronic textbook. Version 1.5.5, http://softwarefoundations.cis.upenn.edu.

[3] Alex Bäuerle, Ángel Alexander Cabrera, Fred Hohman, Megan Maher, David Koski, Xavier Suau, Titus Barik, and Dominik Moritz. 2022. Symphony: Composing interactive interfaces for machine learning. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–14.

[4] Bernhard Beckert, Sarah Grebing, and Florian Böhl. 2015. 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, 3–19.

[5] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, and Francis Montagnac. 1997. *User Guide to the CTCOQ Proof Environment*. Technical Report. INRIA.

[6] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. *Qualitative HCI Research: Going Behind the Scenes*. Morgan & Claypool Publishers.

[7] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. 2012. 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, 32–48.

[8] Sarah E Chasins, Elena L Glassman, and Joshua Sunshine. 2021. PL and HCI: Better together. *Commun. ACM* 64, 8 (2021), 98–106.

[9] Shardul Chiplunkar and Clément Pit-Claudel. 2023. Diagrammatic notations for interactive theorem proving. In *4th International Workshop on Human Aspects of Types and Reasoning Assistants*. EPFL.

[10] Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. 2007. Tinycals: Step by step tacticals. *Electronic Notes in Theoretical Computer Science* 174, 2 (2007), 125–142.

[11] David Delahaye. 2008. A Proof Dedicated Meta-Language. *Electronic Notes in Theoretical Computer Science* 70, 2 (2008).

[12] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. 2022. A drag-and-drop proof tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 197–209.

[13] Jim Fehrle. 2022. A Visual Ltac Debugger in CoqIDE. In *The Eigth International Workshop on Coq for Programming Languages*.

[14] John Nathan Foster. 2009. *Bidirectional programming languages*. Ph.D. University of Pennsylvania, United States – Pennsylvania. https://www.proquest.com/docview/304986072/abstract/11884B3FBDDB4DCFPQ/1 ISBN: 9781109710137.

[15] Gudmund Grov, Aleks Kissinger, and Yuhui Lin. 2013. 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, 324–339.

[16] Wojciech Jedynak. 2013. *Operational Semantics of Ltac*. Master's thesis. University of Wrocław.

[17] Sam Lau, Sruti Srinivasa Srinivasa Ragavan, Ken Milne, Titus Barik, and Advait Sarkar. 2021. Tweakit: Supporting end-user programmers who transmogrify code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[18] Sorin Lerner. 2020. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[19] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. 2020. How domain experts create conceptual diagrams and implications for tool design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–14.

[20] Alok Mysore and Philip J Guo. 2017. Torta: Generating mixed-media gui and command-line app tutorials using operating-system-wide activity tracing. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 703–714.

[21] Wojciech Nawrocki, Edward W Ayers, and Gabriel Ebner. 2023. An extensible user interface for Lean 4. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[22] 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. Version 6.7, http://softwarefoundations.cis.upenn.edu.

[23] Clément Pit-Claudel. 2020. Untangling mechanized proofs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 155–174.

[24] Olivier Pons. 1999. *Conception et réalisation d'outils d'aide au développement de grosses théories dans les systèmes de preuves interactifs*. Ph. D. Dissertation.

[25] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 115–129.

[26] Valentin Robert. 2018. Front-end tooling for building and maintaining dependently-typed functional programs.

[27] Jessica Shi, Benjamin Pierce, and Andrew Head. 2023. Towards a Science of Interactive Proof Reading. Plateau Workshop.

[28] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 198–207.

[29] Coq Development Team. 1989-2024. The Coq Proof Assistant. http://coq.inria.fr

[30] Xiong Zhang and Philip J Guo. 2017. Ds.js: Turn any webpage into an example-centric live programming environment for learning data science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 691–702.