Proof Deautomation: Extended

Abstract

This document is a companion to our PLATEAU paper. It contains extended versions of §5 and §7.1, with more technical details included. It also has an additional example (§Example) covering more advanced features of deautomation.

5 Technical Details

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

5.1 Grammar

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

Atomic tactics are opaque to deautomation. To reason about how they behave, we assume a black-box run-atomic function that determines the result of executing atomic tactics. Its type is atomic $\to$ goal $\to$ (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:

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

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 [2] 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:

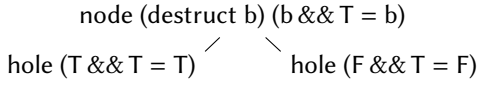
$$r := \text{hole}\ g \mid \text{node}\ a\ g\ r^\star$$

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):

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

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:



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

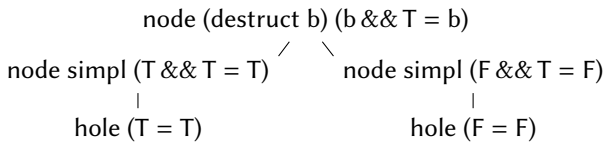
For semicolons, we define:

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

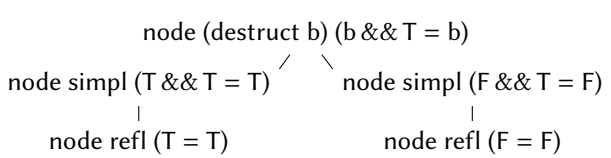
We will explain this in just a moment; conceptually, though, treeification parallels tactic execution. When executing $t_1\ ;\ t_2$ on a goal g, we execute t_1 on g, resulting in goals gs, then execute t_2 on each goal in gs. When treeifying, we first compute treeify t_1 g, resulting in tree r, then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

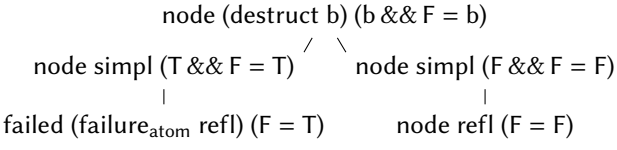
Basics of Recovery. We use failed in a tree to indicate that error e occurred at goal g, where e records the atomic tactic a that failed.

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

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}:

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

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:

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

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

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

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 $\text{let } x \leftarrow mx \text{ in } k\ x$ for $mx \gg\!= k$.

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

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

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.

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

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

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

(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 $[t \mid \text{idtac}]$ fails, we might want to know why it failed. To retain this information, we add a new construct to trees:

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

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

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

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:

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

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

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

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 [2] (specifically the "Ltac — The Tactics" chapter). We extend the semantics in [2] 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 §7. For now, however, our priority is not to model the full complexity of Ltac, but rather to carve out a subset that allows us to explore interesting questions about deautomation. 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 §5.5.)

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.

5.5 Proof-of-Concept Implementation

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

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

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 §Example.

Advanced 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* [1], 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)
```

[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.

```
        (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.

7 Future Work

7.1 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 $x \leq 0$, 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 [3]. 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.

References

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

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

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