Automated Discovery of Tactic Libraries for Interactive Theorem Proving

YUTONG XIN, University of Texas at Austin, USA
JIMMY XIN, University of Texas at Austin, USA
GABRIEL POESIA, Stanford University, USA
NOAH D. GOODMAN, Stanford University, USA
QIAOCHU CHEN, New York University, USA
IŞIL DILLIG, University of Texas at Austin, USA

Enabling more concise and modular proofs is essential for advancing formal reasoning using interactive theorem provers (ITPs). Since many ITPs, such as Rocq and Lean, use tactic-style proofs, learning higher-level custom tactics is crucial for proof modularity and automation. This paper presents a novel approach to tactic discovery, which leverages Tactic Dependence Graphs (TDGs) to identify reusable proof strategies across multiple proofs. TDGs capture logical dependencies between tactic applications while abstracting away irrelevant syntactic details, allowing for both the discovery of new tactics and the refactoring of existing proofs into more modular forms. We have implemented this technique in a tool called TacMiner and compare it against an anti-unification-based approach (Peano) to tactic discovery. Our evaluation demonstrates that TacMiner can learn 3× as many tactics as Peano and reduces the size of proofs by 26% across all benchmarks. Furthermore, our evaluation demonstrates the benefits of learning custom tactics for proof automation, allowing a state-of-the-art proof automation tool to achieve a relative increase of 172% in terms of success rate.

CCS Concepts: • **Hardware** → **Theorem proving and SAT solving**; • **Theory of computation** → **Abstraction**; • **Software and its engineering** → *Automatic programming*.

Additional Key Words and Phrases: Interactive Theorem Proving, Machine Learning

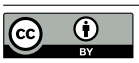
ACM Reference Format:
Yutong Xin, Jimmy Xin, Gabriel Poesia, Noah D. Goodman, Qiaochu Chen, and Işıl Dillig. 2025. Automated Discovery of Tactic Libraries for Interactive Theorem Proving. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 343 (October 2025), 28 pages. https://doi.org/10.1145/3763121

1 Introduction

Tactics are essential in interactive theorem proving, particularly for facilitating reuse and modularity in proof development. By encapsulating common proof strategies into reusable components, tactics allow users to apply previously developed solutions to new problems, reducing the effort required to construct proofs from scratch. Tactics also facilitate proof automation, as shorter proofs are typically easier to discover using automated techniques.

While interactive theorem provers (ITPs) such as Rocq (formerly known as Coq) and Lean provide a rich library of built-in tactics, proof engineers typically need to devise *custom tactics*

Authors' Contact Information: Yutong Xin, maxryeery@utexas.edu, University of Texas at Austin, Austin, Texas, USA; Jimmy Xin, jxin31415@utexas.edu, University of Texas at Austin, Austin, Texas, USA; Gabriel Poesia, poesia@stanford.edu, Stanford University, Stanford, California, USA; Noah D. Goodman, ngoodman@stanford.edu, Stanford University, Stanford, California, USA; Qiaochu Chen, qc1127@cs.nyu.edu, New York University, New York, New York, USA; Işıl Dillig, isil@cs.utexas.edu, University of Texas at Austin, Austin, Texas, USA.



ACM 2475-1421/2025/10-ART343
https://doi.org/10.1145/3763121

```
Proof.
intros. destruct H as [HP HQ].
apply H0 in HP as HP'. apply H2.
rewrite H1 in HQ. split.
- exact HP'.
- exact HQ.
Qed.
```

```
Proof.
intros Hconj HR Heq HW.
destruct Hconj as [H1 H2].
rewrite Heq in H2. apply HW.
split.
- apply HR in H1 as H1'. exact H1'.
- exact H2.
Qed.
```

Fig. 1. Syntactically different proofs w/ same TDG in Figure 2, for Lemma $(P \wedge Q) \rightarrow (P \rightarrow R) \rightarrow (Q = T) \rightarrow (R \wedge T \rightarrow W) \rightarrow W$.

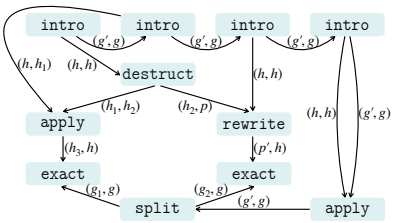


Fig. 2. TDG for Figure 1.

by composing existing tactics into higher-level domain-specific building blocks [14]. For example, the `mathlib` library in Lean contains useful tactics for mathematical proofs, and the `flocq` library in Rocq provides tactics that facilitate proofs about floating point computation. But even if proof assistants enable users to write their own tactics, learning to do so adds an extra layer on top of the already steep learning curve of ITPs.

In this paper, we address the problem of automatically learning *custom tactics* (i.e., compositions of existing tactics) from a given proof corpus. Learning such tactic libraries can uncover reusable patterns in proofs, making it easier to construct similar proofs within the same domain. Furthermore, tactic discovery can enhance proof automation by facilitating a form of curriculum learning [32], where more complex proofs are synthesized using custom tactics discovered in simpler proofs.

While there has been prior work [8, 10, 18] on learning software libraries from a given corpus of *programs*, these methods do not readily apply to tactic-based proofs. Such proofs are written in an imperative style, with frequent reference to implicit, mutating objects such as the current goal, which is typically not present in the source, but rather visualized interactively. However, existing work on library learning typically assumes functional programs and focuses on generalizing concrete program expressions into lambda abstractions. In contrast, tactic discovery requires an understanding of the logical relationships between different proof steps and the ability to construct higher-level tactics that capture these relationships.

In this paper, we address this problem by proposing a new proof abstraction called *tactic dependence graph (TDG)* that hides irrelevant syntactic variations between different proofs, while focusing on important *logical* dependencies. In a TDG, nodes represent tactic *applications*, while edges represent *proof state dependencies* between them. For example, Figure 1 shows two syntactically different proofs of the same theorem, which have exactly the same TDG abstraction shown in Figure 2. In essence, the TDG abstraction hides minor syntactic variations in the proof, such as how sub-goals are named or in what order tactics are applied, and instead focuses on *semantic* dependencies in between different tactic applications.

Given a corpus of proofs from the same domain, our approach leverages the TDG abstraction to discover tactics that maximize compression of the proof corpus—a metric widely used in prior work on library learning [8, 10, 18, 41]. Intuitively, the greater the compression rate of existing proofs, the more broadly applicable the resulting tactics. To achieve this goal, our method identifies *isomorphic subgraphs* within the TDGs that exhibit a special property called *collapsibility*. This property guarantees that (1) each common subgraph can be translated into a valid tactic and (2) the resulting tactics can be applied to refactor the original proofs while preserving their validity.

Our tactic discovery algorithm is inspired by ideas from *top-down* enumerative program synthesis [8, 21, 23, 38]; however, it needs to address additional challenges that are not present in the program synthesis setting. First, since our goal is to discover common isomorphic subgraphs of existing TDGs, we need to perform enumerative search over *graphs* as opposed to abstract syntax *trees*. Second, unlike program synthesis, there is no pre-determined grammar for tactic

dependence graphs. Finally, since our goal is to maximize an optimization objective, the synthesizer cannot stop as soon as it finds an isomorphic embedding but must keep going until it identifies the highest-scoring tactic. Our proposed tactic discovery algorithm effectively addresses these challenges using two key ideas. First, it extracts a graph grammar from the TDGs of existing proofs. Second, it substantially prunes the search space by deriving upper bounds on the compression power of candidate tactics.

We have implemented our proposed tactic discovery method in a tool called TacMiner and evaluate it on several Rocq projects, including CompCert [35] and proofs involving program logics. We compare our approach against an anti-unification baseline from prior work (Peano) [41] and show that TacMiner can learn 3× as many tactics compared to Peano. Furthermore, the tactics learned by TacMiner make proofs 26% shorter across all benchmarks, reducing the corpus size to 63% of its original size in some cases. Finally, our evaluation also demonstrates the benefits of our approach for proof automation, allowing a state-of-the-art tool (Copra) [50] to achieve a relative increase of 172% in the number of theorems proved.

To summarize, this paper makes the following contributions:

- We introduce *tactic dependence graphs (TDG)*, a new abstraction that facilitates proof refactoring and tactic discovery.
- We define the *semantic proof refactoring* problem and show how to use the TDG abstraction to refactor proofs in a way that preserves their validity.
- We propose a new tactic discovery algorithm for assembling a library of tactics from a corpus.
- We implement these ideas in a tool called TacMiner targeting Rocq proofs and compare it against a baseline [41] that uses anti-unification for tactic discovery. Our method learns around 3× as many tactics compared to the baseline, reducing the size of the proof corpus by 26% (compared to 9% for the baseline). Furthermore, the tactics learned by our method enabled Copra, a proof automation tool, to increase the number of theorems it can prove by 172% on a corpus of 50 proofs, providing preliminary evidence of these tactics' utility in automated theorem proving.

2 Motivating Example

In this section, we demonstrate our approach through a simple motivating example shown in Figure 3. This figure shows two simple Rocq proofs, `eq_sym` and `eq_trans`, taken from CompCert [35] for establishing the symmetry and transitivity of a predicate defining a suitable notion of equality for dataflow analysis results. As standard in Rocq and many other interactive theorem provers, these are *tactic-style proofs*, where pre-defined tactics such as `intros, destruct, rewrite`, etc. are used to transform proof goals into simpler sub-goals. Our goal in this paper is to learn *higher-level (custom) tactics* that can be used to further simplify proof engineering in a specific domain.

```
Lemma eq_sym: forall x y,
                eq x y -> eq y x.
Proof.
red. intros. destruct x.
- destruct y.
  + red in H.
    unfold EqSet.Equal.
    intros.
    rewrite H.
    reflexivity.
  + auto.
- auto.
Qed.
```

```
Lemma eq_trans: forall x y z: res eqs,
                  eq x y -> eq y z -> eq x z.
Proof.
unfold eq. intros. destruct x.
- destruct y.
  + destruct z.
    * red in H. red in H0.
      unfold EqSet.Equal.intros.
      rewrite H. rewrite H0. tauto.
    * auto.
  + tauto.
- destruct y.
  + destruct z.
    * auto.
    * auto.
  + auto.
Qed.
```

Fig. 3. Input Rocq Proofs.

While the two proofs shown in Figure 3 have salient differences, they also have many similarities, as highlighted in color in the figure. In fact, given these two proofs, we can extract the following custom tactics that may be used to simplify several other proofs:

```
Ltac simplRewrite H := red in H; rewrite H.
Ltac destructUnfold n H := destruct n; [unfold H; intros | auto].
```

The first custom tactic called `simplRewrite` is useful in hypothesis `H` which contains a function or expression that requires some reduction before rewriting. In particular, this tactic first simplifies the hypothesis `H` by performing one step of reduction, and then uses that simplified hypothesis to rewrite the goal. The second tactic, called `destructUnfold`, is useful in scenarios that require handling different cases of an inductive type through unfolding and proof automation, respectively. Figure 4 shows the refactored version of the proofs from Figure 3 using these custom tactics.

Proof Refactoring. As a first step towards tactic discovery, we pose the question *"Can we, and, if so how do we, refactor proofs using a given custom tactic?"*. At first glance, it is unclear how to refactor the proofs from Figure 3 to use our custom tactics. First, the pre-defined tactics comprising the custom tactic do not appear consecutively in either of the proofs. Second, the `destructUnfold` tactic spans multiple cases of the original proof, but its first branch does not exactly correspond to either of the branches of the original tactic. Third, even if a proof uses the same pre-defined tactics as a custom tactic, this does not necessarily mean that a proof can be refactored using that tactic. In particular, because tactic-style proofs are stateful objects with implicit sub-goals, there can be subtle dependencies between tactic applications that make it impossible to refactor the proof using a given tactic *even when* the proof and the tactic are *syntactically* similar. All of these considerations highlight the need for a suitable abstraction that can facilitate proof refactoring and tactic discovery.

```
Lemma eq_sym_compr: forall x y:
res eqs, eq x y -> eq y x.
Proof.
red. intros. destruct x.
destructUnfold y EqSet.Equal.
- simplRewrite H.
reflexivity.
- auto.
Qed.
```

```
Lemma eq_trans_compr: forall x y z:
res eqs, eq x y -> eq y z -> eq x z.
Proof.
unfold eq. intros. destruct x.
- destruct y.
  + destructUnfold z EqSet.Equal.
    simplRewrite H. simplRewrite H0.
    tauto.
  + contradiction.
- destruct y.
  + destruct z.
    * auto.
    * auto.
  + auto.
Qed.
```

Fig. 4. Refactored proofs from Figure 3 using the custom tactics.

Tactic Dependence Graph. To address these problems, we propose representing tactic-style proofs using an abstraction called the *tactic dependence graph (TDG)* that elucidates dependencies between different tactic applications in a proof. Specifically, nodes in a TDG correspond to tactic applications, and edges encode how the "outputs" of one tactic feed as "inputs" to another tactic. For instance, consider the TDG representation of the proof for the `eq_sym` lemma shown in Figure 5. Here, an edge from a node n to n' indicates that the tactic application represented by n' depends on the tactic application represented by n. For example, looking at this TDG, we see that there is no logical dependence between the second application of `red` (node u) and `unfold` (node w) even though they appear consecutively in the syntax of the proof. On the other hand, the TDG also makes it clear that there is an immediate dependency between the second application of `red` (u) and `rewrite` (v) even though they are separated in syntax by two other tactic applications. Also, note that TDG edges elucidate not only *whether* there exists a dependency between two tactic applications but also *how*

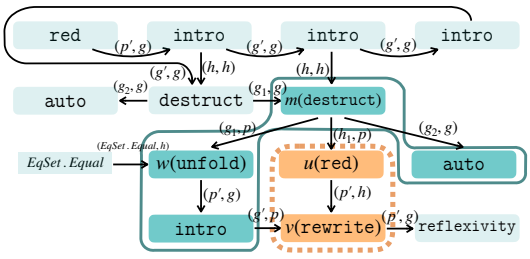


Fig. 5. TDG for `eq_sym`.

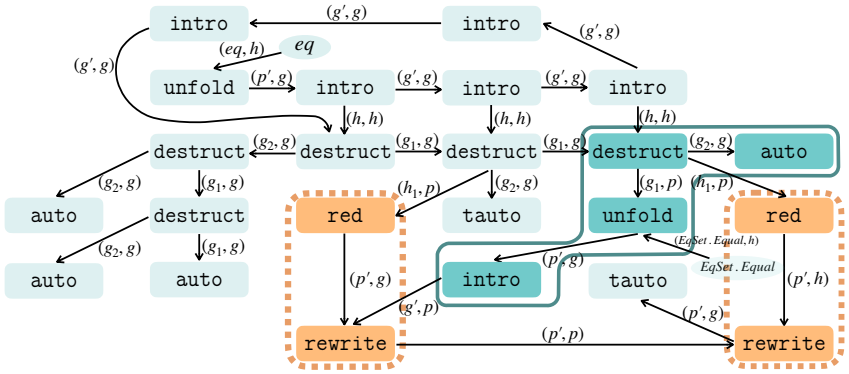


Fig. 6. TDG for `eq_trans`.

they are dependent. In particular, an edge labeled (α, β) indicates that the "formal" output α of the source tactic application is supplied as the "formal" parameter β of the target tactic application. As a matter of convention, we use the symbols h, h_i to denote hypotheses, g, g_i to denote goals, and p, p_i to denote propositions that can be either goals or hypotheses. For instance, looking at the same TDG, we see that `destruct` (node m) produces two goals (labeled g_1, g_2) and a new hypothesis (labeled h_1). We also notice that the formal outputs g_1, g_2 of `destruct` (node m) are used by `unfold` (node w) and `auto` respectively, and that the output hypothesis h_1 is used by `red` (node u).

TDG for Proof Refactoring. Next, let's consider how we might use the TDG for refactoring the `eq_sym` proof from Figure 3 into the form shown in Figure 4. First, observe that the subgraph encircled using solid (green) lines in Figure 5 corresponds to the TDG representation of the `destructUnfold` tactic and the subgraph surrounded by orange dashed lines corresponds to that of `simplRewrite`. Hence, we can refactor the original proof by replacing this sequence of tactic invocations with the `destructUnfold` and `simplRewrite` tactics. Next, there is the question of *where* in the proof these custom tactic invocations belong. Looking at the TDG, we see that an input of `red` (now part of `simplRewrite`) relies on an output of `destruct` (which is now a part of `destructUnfold`); hence, we deduce that `destructUnfold` must occur before `simplRewrite` in the refactored proof. Using this type of reasoning, we can refactor the original proof from Figure 3 to the version shown in Figure 4 while maintaining its syntactic and semantic validity.

Tactic Discovery. In addition to being useful for proof refactoring, the TDG abstraction is also beneficial for *tactic discovery*: Given a corpus of proofs, we can identify common proof patterns by looking for common isomorphic subgraphs of the TDGs. For instance, comparing the TDGs shown in Figure 5 and Figure 6 immediately reveals two common isomorphic subgraphs, highlighted in solid (green) and dashed (orange) lines. Hence, the idea behind our tactic discovery algorithm is to find isomorphic common subgraphs in the corpus and use them to refactor the proofs. However, when learning tactics, there is often a trade-off between the generality of the tactic and its size: On one hand, the more frequently a proof pattern appears, the more generally applicable it is. On the other hand, the larger the tactic, the more useful it is — intuitively, large tactics allow us to skip many steps in the proof. To balance this trade-off, our tactic discovery algorithm looks for common isomorphic subgraphs that maximize the total reduction in corpus size, a metric that has also been used in prior work on library learning [8, 10]. For instance, going back to our running example, the `destructUnfold` tactic reduces the size of the two proofs by 20% whereas the `simplRewrite` tactic reduces size by 10%. Thus, our learning algorithm first refactors the corpus using the `destructUnfold` tactic and then looks for a different tactic that maximizes compression,

such as the `simplRewrite` tactic in our example. This process continues until no new tactics can be discovered.

3 Preliminaries

This section provides necessary background on interactive theorem proving by developing a formal model of tactic-style proofs.

3.1 Proof States

A central concept in tactic-style proofs is the notion of *proof state*, which represents the current status of an ongoing proof, including the goals that remain to be established and the hypotheses available at each step.

Definition 3.1 (Proof state). A proof state σ is a mapping from identifiers (i.e., names like `H1`) to *proof elements*, which correspond to hypotheses and goals. A *hypothesis* is a proposition φ expressed in the formal language of the prover. A *goal* g is a pair (φ, C) where φ is a proposition to be proven and the *context* C is a set of hypothesis identifiers that are in scope when proving g.

Given an identifier v, we write $g(v)$ to indicate that it represents a proof goal and $h(v)$ to denote that it corresponds to a hypothesis. We use the notation $\text{Goals}(\sigma)$ to denote the set of goals in the proof state σ. Also, given a proof state σ and a set of identifiers I, we write $\sigma \backslash I$ to denote the new proof state obtained by removing identifiers I from σ.

For readability, we omit the context associated with a goal unless necessary. Thus, in most examples, we represent the proof state as a mapping from identifiers to propositions, assuming that all hypotheses are available when proving a goal unless explicitly stated. Additionally, while many ITPs do not explicitly name proof goals—typically operating on a single implicit goal at each step—we choose to explicitly name goals to facilitate disambiguation in multi-goal scenarios.

3.2 Tactics

In interactive theorem provers, theorems are typically proven by applying *tactics* to manipulate proof states. For example, Table 1 lists our representation of some of the built-in tactics in the Rocq theorem prover, along with an (informal) description of their semantics.[1] For instance, consider the `intro` tactic shown in Table 1 which is listed as having "signature" $g \rightarrow h \times g'$. This means that this tactic operates over a specific goal g that is part of the current proof state σ and transforms the proof state if g is of the form $p_1 \rightarrow p_2$. In particular, the new proof state is obtained by removing goal g from the current proof state and introducing p_1 as a new hypothesis and p_2 as a new goal. Note that some tactics, such as `split` and `destruct`, can also introduce multiple new goals, where each goal can have its own unique context (such as in the case of `destruct`).

In addition to using built-in tactics, proof engineers can also define their own *custom* tactics, for example, via the `Ltac` construct in Rocq. We represent both built-in and custom tactics through the following formalization:

Definition 3.2 (Tactic definition). A *tactic definition* is a quadruple $(\eta, I, O, \mathcal{E})$ where η is the name of the tactic, I and O are lists of identifiers representing formal tactic inputs and outputs respectively, and $\mathcal{E}$ is an expression that manipulates a proof state.

Example 3.1. Consider the following simple tactic in Rocq: `Ltac MyTac h := intro h; simpl.` This custom tactic first applies the built-in `intro` tactic to introduce a hypothesis h and then simplifies the resulting proof goal using `simpl`. In our representation, this tactic is formalized as:

$$(\texttt{MyTac}, [g], [h, g''], \texttt{intros}\ [g]\ [h, g']; \texttt{simpl}\ [g']\ [g''])$$

[1] Note that Table 1 only provides examples; our implementation supports many more, as explained in Section 7.

Table 1. Our internal representation of some commonly used Rocq tactics, where h denotes a hypothesis, g denotes a goal, and p is any proposition (either goal or hypothesis).

Tactic name	Signature	Semantics
`intro`	$g \to h \times g'$	If g matches $p_1 \to p_2$, produces p_1 as hypothesis h and p_2 as goal g'
`apply .. (in)`	$h \times p \to p_1(\times ... \times p_n)$	If h matches $p \to p_1$ and p is a hypothesis, produces p_1 as a new hypothesis; if h matches $p_1 \to \ .. \ \to p_n \to p$ and p is a goal, produces $p_1 \times \ .. \ \times p_n$ as new goals
`exact`	$h \times g \to \bot$	If h matches g, discharges the goal (i.e., no new goal is produced)
`split`	$g \to g_1 \times g_2$	If g is of the form $(p_1 \wedge p_2)$, produces p_1 and p_2 as two new goals g_1 and g_2, respectively
`destruct`$_\vee$	$h \times g \to$ $h_1 \times h_2 \times g_1 \times g_2$	If h matches $(h_1 \vee h_2)$, produces two goals g_1 and g_2, derived from g, with corresponding hypotheses h_1 and h_2
`rewrite`	$h \times p \to p'$	If h matches $x = y$ and p is of the form $p(x)$, produces $p(y)$ as the new proposition p'
`left` / `right`	$g \to g'$	If g matches $p_1 \vee p_2$, produces p_1 (resp. p_2 for `right`) as a new goal g'

Here, the formal input I consists of a single goal g, and the formal output is $[h, g'']$, indicating that the resulting proof state includes a new hypothesis h and a new goal g''. This explicit representation captures *which parts* of the proof state are modified by a given tactic. Similarly, when specifying tactic applications (as in the body of the custom tactic), we explicitly define their inputs and outputs. In this case, `intros` takes the initial goal g, introduces a hypothesis h, and produces an updated goal g', which is then further transformed by `simpl` into g''. This formalization clarifies the sequential modifications to the proof state.

In the remainder of this paper, we assume uniqueness of tactic names, and, given a tactic name η, we write $\mathsf{In}(\eta)$ and $\mathsf{Out}(\eta)$ to denote its formal inputs and outputs, and $\mathsf{Body}(\eta)$ to denote its body.

3.3 Tactic Applications

A tactic application modifies the proof state by taking a set of actual proof elements as inputs and producing new proof elements as outputs. Formally, a tactic application is represented as a tuple (η, X, Y) where η is the name of the applied tactic, X is the list of *actual inputs* (e.g., goals, hypotheses), Y is the list of *actual outputs* (e.g., new hypotheses, transformed goals). In our representation, a tactic application is exactly akin to *function invocation* with *call-by-value* semantics: Given a tactic definition $(\eta, I, O, \mathcal{E})$ where I and O denote the formal inputs and outputs, respectively, applying η involves replacing formal inputs with actual proof elements and executing $\mathcal{E}$. More formally, each tactic invocation $t = (\eta, X, Y)$ defines a transition relation between proof states, denoted $[\![t]\!](\sigma) = \sigma'$, where $\sigma' = (\sigma \backslash \mathsf{Goals}(X)) \ \uplus \ [Y \mapsto ([\![\mathcal{E}]\!]([\mathsf{In}(\eta) \mapsto \sigma(X)])(\mathsf{Out}(\eta))]$. In particular, $[\![t]\!](\sigma)$ modifies the proof state σ according to body expression $\mathcal{E}$, subject to the usual formal-to-actual renamings. Note that inputs of t that correspond to proof goals are transformed by the tactic invocation; hence, identifiers that refer to "stale" proof goals are removed when constructing the new input state σ'. Importantly, we assume tactics are deterministic with respect to the inputs (i.e., goals, hypotheses, and hint databases) they consume. That is, tactic invocations will always produce the same outputs as long as their inputs in the initial proof state are the same, even if the rest of the state is different. Given proof state σ and expression $\mathcal{E}$, we write $[\![\mathcal{E}]\!](\sigma)$ to denote the resulting state σ' after applying $\mathcal{E}$ to σ.

Example 3.2. Consider the following tactic application (`apply, [H0, g3], [g4, g5]`) (where the semantics of the built-in `apply` tactic is given in Table 1) and the proof state σ:

$$[\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3}),\ \ \texttt{g3} \mapsto \texttt{P3}]$$

This tactic application results in the following modified proof state σ':

$$[\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3}),\ \ \texttt{g4} \mapsto \texttt{P1},\ \ \texttt{g5} \mapsto \texttt{P2}]$$

Observe that the hypotheses are still the same, but previous goal `g3` is removed from the old proof state and two new goals, `g4` and `g5`, are added to σ'.

3.4 Proof Scripts and Proofs

We conclude this section by defining *proof scripts*: programs defined by a sequence of tactic invocations. A *proof* corresponds to a successful *execution* of that program on some initial state.

Definition 3.3 (Proof script). A *proof script* π is a sequence of tactic invocations.

Example 3.3. Consider the following lemma:

```
Lemma implication: (P1 ∧ P2) -> (P1 -> P2 -> P3) -> P3.
```

and its corresponding proof script:

```
intros H. destruct H as [h1 h2]. intros H0. apply H0.
- exact h1.
- exact h2.
```

In our representation, this proof script corresponds to the following sequence of tactic invocations:

```
intro [g0] [H, g1];  destruct [H, g1] [h1, h2, g2];
intro [g2] [H0, g3]; apply [H0, g3] [g4, g5];
exact [h1, g4] [];   exact [h2, g5] [].
```

Although the original Rocq script uses bullets ("-") to indicate a structured proof with multiple subgoals, our proof script representation explicitly encodes the logical dependencies between tactics as a linear sequence of invocations. In particular, the `apply H0` tactic introduces two subgoals, labeled `g4` and `g5`, which are subsequently solved by `exact h1` and `exact h2`, respectively.

Finally, we define a *proof* as a successful execution of a proof script on some initial state:

Definition 3.4 (Proof). Let σ_0 be a proof state and $\pi = t_1; \ldots; t_n$ be a proof script. Executing π on σ_0 yields a sequence of states (i.e., *trace*) $\sigma_1, \ldots \sigma_n$ where $\sigma_{i+1} = [\![t_i]\!](\sigma_i)$. We say that π is a *proof* of σ_0 if σ_n does not contain any goal identifiers in its domain.

Example 3.4. Consider the initial proof state, which corresponds to our lemma to be proven from Example 3.3: $\sigma_0 : [\texttt{g0} \mapsto ((\texttt{P1} \wedge \texttt{P2}) \to (\texttt{P1} \to \texttt{P2} \to \texttt{P3}) \to \texttt{P3})]$. Executing the proof script from Example 3.3 results in the following trace:

$$\begin{aligned}
\sigma_1 :&\ [\texttt{H} \mapsto (\texttt{P1} \wedge \texttt{P2}),\ \ \texttt{g1} \mapsto ((\texttt{P1} \to \texttt{P2} \to \texttt{P3}) \to \texttt{P3})] \\
\sigma_2 :&\ [\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{g2} \mapsto ((\texttt{P1} \to \texttt{P2} \to \texttt{P3}) \to \texttt{P3})] \\
\sigma_3 :&\ [\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3}),\ \ \texttt{g3} \mapsto \texttt{P3}] \\
\sigma_4 :&\ [\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3}),\ \ \texttt{g4} \mapsto \texttt{P1},\ \ \texttt{g5} \mapsto \texttt{P2}] \\
\sigma_5 :&\ [\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3}),\ \ \texttt{g5} \mapsto \texttt{P2}] \\
\sigma_6 :&\ [\texttt{h1} \mapsto \texttt{P1},\ \ \texttt{h2} \mapsto \texttt{P2},\ \ \texttt{H0} \mapsto (\texttt{P1} \to \texttt{P2} \to \texttt{P3})]
\end{aligned}$$

Since σ_6 does not contain any goals, this constitutes a proof of our lemma.

4 Semantic Proof Refactoring

To define our tactic discovery problem, we first consider: "What makes a custom tactic useful?" In our setting, a necessary requirement is that the tactic can be applied to refactor existing proofs in a way that yields a desirable outcome—be it reduced proof size, improved maintainability, or some other benefit. In this section, we formally introduce the concept of *proof refactoring*. Instead of relying on brittle, syntactic notions, our approach takes a semantic perspective by representing tactic-style proofs with *tactic dependency graphs (TDG)*.

Definition 4.1 (TDG of proof script). Let π be a proof script. A *tactic dependence graph* for π, denoted $\mathrm{TDG}(\pi)$, is a directed acyclic graph $G = (V, E)$ where each node $v(\eta) \in V$ corresponds to a tactic invocation $t = (\eta, X, Y) \in \pi$ and each arc $(s(\eta), t(\eta'), \beta, \alpha)$ indicates that the formal input α for tactic invocation $t(\eta')$ corresponds to the formal output β for tactic invocation $s(\eta)$.

Example 4.1. Figure 7 shows the TDG for the proof script from Example 3.3. Here, an edge labeled (a, b) indicates the formal output a of the source node corresponds to the formal input b of the target node. We use the signatures shown in Table 1 to refer to the formal inputs and outputs of Rocq's built-in tactics. For example, the edge from v_3 to v_4 labeled (g', p) indicates that formal output g' of the `intro` tactic becomes the formal input p of the `apply` tactic.

In general, multiple syntactically different proof scripts can have the same TDG. Given a graph G, we use $\mathsf{InducedProofs}(G)$ to denote the set of all proof scripts $\Pi = \{\pi_1, \ldots, \pi_k\}$ such that G is isomorphic to $\mathrm{TDG}(\pi_j)$. Thus, we can view a TDG as representing a combinatorially large class of syntactically different, but semantically equivalent, proof scripts. We can obtain all induced proofs of G by considering all topological sorts of G that satisfy branching constraints of the original proof (see Section 7 for more details).

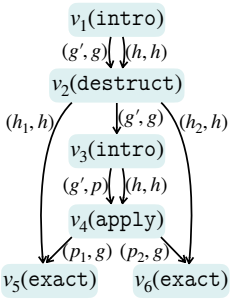


Fig. 7. TDG for Ex 3.3.

Note that we can also represent a tactic definition as a TDG as long as its body expression can be expressed as a sequence of other tactic invocations. However, to simplify the presentation of our algorithms, we augment the TDG for each tactic definition with two special nodes to represent their formal inputs and outputs.

Definition 4.2 (Tactic TDG). Let $\tau = (\eta, I, O, \pi)$ be a tactic definition where π is a sequence of tactic invocations, and let $G_\pi = (V_\pi, E_\pi)$ be the TDG of π. The TDG of τ is a directed acyclic graph $G = (V, E)$ such that $V = V_\pi \cup \{v_{\mathsf{in}}, v_{\mathsf{out}}\}$ where $v_{\mathsf{in}}, v_{\mathsf{out}}$ represent the formal inputs and outputs of τ respectively. Furthermore, let I_{α_i,β_j} denote the set of all tactic invocations of the form $(\eta', \_, \_)$ where the formal input α_i of τ corresponds to formal input β_j of η', and let O_{α_k,β_l} denote the set of all tactic invocations of the form $(\eta', \_, \_)$ where the formal output α_k of τ corresponds to formal output β_l of η'. Then, G contains edges:

$$E = E_\pi \cup \bigcup_{ij} I_{\alpha_i,\beta_j} \cup \bigcup_{kl} O_{\alpha_k,\beta_l}$$

In other words, the TDG for a tactic contains *special nodes and edges* that connect formal inputs and outputs in the tactic definition to the formal inputs and outputs of other tactic invocations.

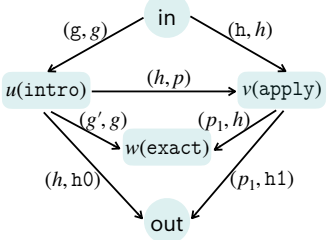


Fig. 8. Tactic TDG for Ex. 4.2.

Example 4.2. Consider the following tactic definition in Rocq:

```
Ltac myTac h h0 h1:= intro h0; apply h0 in h as h1; exact h1.
```

In our internal representation, this tactic takes two formal arguments, namely goal g and hypothesis h, and has two formal outputs h0 and h1. The body expression of this tactic is represented using the following sequence of tactic invocations:

```
intro [g] [h0, g1]; apply [h, h0] [h1]; exact [h1, g1] []
```

Figure 8 shows the TDG representation of this tactic. Note that there is an edge from node u to the special exit node labeled $(h, \mathtt{h}_0)$ since the formal output h of the `intro` tactic becomes the formal output h0 of the custom tactic. The edge between v and the exit node is labeled $(p_1, \mathtt{h}_1)$ similarly.

Intuitively, a proof script π could be refactored using a tactic τ if τ is a subgraph of π modulo isomorphism, subject to some extra restrictions. To make this more precise, we define *isomorphic embedding* in the context of TDG.

Definition 4.3 (Isomorphic embedding). A tactic TDG $G = (V, E)$ is an isomorphic embedding into a proof TDG $G' = (V', E')$ iff there exists an injective function $f : V \backslash \{v_{\text{in}}, v_{\text{out}}\} \rightarrow V'$ such that:

(1) Every vertex in $V \backslash \{v_{\text{in}}, v_{\text{out}}\}$ is mapped to a vertex of G' with the same tactic name, i.e.,

$$\forall v \in V. \exists v' \in V'. f(v(\eta)) = v'(\eta)$$

(2) Every non-special edge in G is mapped to a corresponding edge of G' with the *same edge label*:

$$\forall (s, t, \beta, \alpha) \in E.\ (s \neq v_{\text{in}} \wedge t \neq v_{\text{out}}) \rightarrow \exists (s', t', \beta, \alpha) \in E'. f(s) = f(s') \wedge f(t) = f(t')$$

We refer to this injective function f as the *witness*.

The two conditions in the above definition ensure agreement between labels of nodes and edges between the tactic TDG G and the proof TDG G' that G is embedded into. However, it turns out that finding an isomorphic embedding between a tactic τ and a proof script π is a necessary but not sufficient condition for refactoring π using τ. To see why this is the case, consider two TDGs, one with edges $\{\mathsf{a} \rightarrow \mathsf{b}, \mathsf{b} \rightarrow \mathsf{c}, \mathsf{a} \rightarrow \mathsf{c}\}$, and another with edges $\{\mathsf{a} \rightarrow \mathsf{d}, \mathsf{d} \rightarrow \mathsf{c}, \mathsf{a} \rightarrow \mathsf{c}\}$. While the subgraph consisting of nodes a and c appears in both proofs, it is not possible to extract a valid tactic that isolates these two steps. This is because node c depends on a node outside the tactic, which in turn relies on an intermediate result produced by the tactic, preventing it from being used as a standalone argument. To avoid this issue, we must additionally ensure that the subgraph of π that is isomorphic to τ is *collapsible* into a single node, meaning that we can replace all incoming edges into the subgraph with the special entry node of the tactic and all outgoing edges with the special exit node. We formalize this using the definition below:

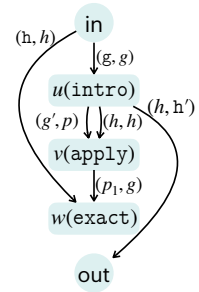


Fig. 9. TDG for Ex 4.3.

Definition 4.4 (Collapsible embedding). Let $G = (V, E)$, $G' = (V', E')$ be the TDG's for a tactic definition τ and proof script π respectively such that G is an isomorphic embedding into G' with witness function f. Let $G \vdash u \rightsquigarrow v$ denote that node v is reachable from node u in graph G. We say that G' is f-collapsible iff both of the following conditions hold:

(1) $\forall u, v \in \mathsf{Range}(f). \forall w \in V'.\ (G' \vdash u \rightsquigarrow w\ \wedge G' \vdash w \rightsquigarrow v) \Rightarrow w \in \mathsf{Range}(f)$

(2) $\forall u, v \in \mathsf{Dom}(f). \forall e \in E'.\ e = (f(u), f(v), \alpha, \beta) \Rightarrow (u, v, \alpha, \beta) \in E$

Here, the first condition states that, if we have $f(a) = u$ and $f(b) = v$ and there is a path from u to v that includes w in the middle, then f must also map some node $c \in V$ to w. On the other hand, the second condition states that if G' includes an edge between any pair of vertices that are in the range of f, then the corresponding edge must also exist in G.

To see why we require the first property, note that any valid proof script induced by G' must include tactic invocations in the order u, w, v since w depends on an output of u and v depends

on an output of w. Thus, if f had only u, v in its range but not w, the refactored proof could not allow the tactic invocations u, v, w in the required order, thereby resulting in an invalid proof script. Additionally, we need the second property because a candidate tactic G is not a valid embedding unless it includes *all* required dependencies between a pair of tactic invocations. Intuitively, these conditions enforce that, if we replace the subgraph of G' that is isomorphic to G with a single node representing a new tactic invocation, then the resulting proof is still valid.

Example 4.3. Consider the proof script and its corresponding TDG from Figure 7. We give examples and non-examples of collapsible isomorphic embeddings.

(1) Consider the tactic from Example 4.2 and its TDG in Figure 8. Note that both the proof script and the tactic contain the sequence of tactic invocations `intro`, `apply`, `exact`. However, the function $[u \mapsto v_2, v \mapsto v_3, w \mapsto v_4]$ does not define an isomorphic embedding because it does not satisfy condition (2) from Definition 4.3.

(2) Now consider the following tactic definition, whose TDG is shown in Figure 9.

```
Ltac myTac2 h0 h := intro h0. apply h0. exact h.
```

In our representation, this tactic has formal inputs g and h and output h'. The TDG from Figure 9 is an isomorphic embedding into Figure 7 with the witness function $[u \mapsto v_3, v \mapsto v_4, w \mapsto v_5]$.

(3) Now, consider a subgraph G_1 of the TDG in Figure 9 that does not contain node v as well as its incoming and outgoing edges. Then, the witness function f from part (2) of this example is an isomorphic embedding but it is not collapsible, as it violates part (1) of Definition 4.4.

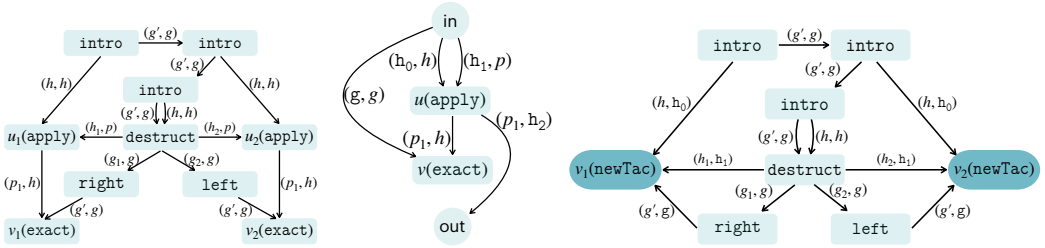
(4) Next, consider a modified version G_2 of Figure 9 that does not contain the edge labeled (h, h) between u, v. Then, the same witness function f still defines an isomorphic embedding but it violates part (2) of Definition 4.4. Intuitively, G_2 is not a valid tactic modulo the original proof because it omits a *required* dependency between `intro` and `apply`.

Our proof refactoring procedure is presented in Algorithm 10: It takes as input a proof script π and a tactic definition τ, and returns a refactored proof script of π. The algorithm first constructs the TDG's G, G_c for the tactic τ and proof script π respectively (line 2) and then it repeatedly contracts G_c in the while loop (lines 3–12) by finding a collapsible isomorphic embedding of G into G_c (line 4). If FindEmbedding does not return a witness (line 5), further refactoring is not possible, so the algorithm returns any one of the induced proofs associated with G_c (line 13). Otherwise, the algorithm proceeds in two steps. First, recall that the identifiers

```
1:  procedure Refactor(τ, π)
    input: A tactic definition τ and a proof script π.
    output: A refactored proof script of π'
2:      G, G_c ← ConstructTDG(τ), ConstructTDG(π)
3:      while true do
4:          f ← FindEmbedding(G, G_c)
5:          if f ≡ ⊥ then break
6:          V ← {v | v ∈ Nodes(G_c) ∧ v ∈ Range(f)}
7:          for all v ∈ V do
8:              for all u ∈ Parents(v) \ Range(f) do
9:                  G_c ← RewireIn(u, v, G, G_c)
10:             for all u ∈ Children(v) \ Range(f) do
11:                 G_c ← RewireOut(u, v, G, G_c)
12:             G_c ← Contract(G_c, V, v(τ.η))
13:     return π' ∈ (InducedProofs(G_c))
```

Fig. 10. Procedure for refactoring a proof script π using tactic τ. This procedure uses auxiliary procedures RewireIn (resp. RewireOut) to change the labels of the incoming (resp. outgoing) edges of v. Given edge (v', v, α, β) in the TDG G_c of π with corresponding edge $(v_{\text{in}}, f^{-1}(v), \gamma, \beta)$ in the TDG G of τ, RewireIn replaces that edge with (v', v, α, γ). Similarly, given edge (v', v, α, β) in G_c with corresponding edge $(f^{-1}(v'), v_{\text{out}}, \alpha, \gamma)$ in G, RewireOut replaces that edge with (v', v, γ, β). Also, FindEmbedding finds a collapsible isomorphic embedding of G into G_c and Contract performs standard graph contraction.

used in the tactic definition τ are different from those used in the original tactic invocation, so we need to change the edge labels to ensure that the refactoring is correct. This is done in lines 8-11 using functions called RewireIn and RewireOut. In particular, given a node v whose parent u is not part of the embedding, we need to change the incoming edges to v to use the correct identifier for τ using the special entry edges in G from v_{in} to $f^{-1}(v)$. RewireOut does something very similar but for "exit" edges whose children are not part of the embedding. Finally, the second step (call to Contract at line 12) replaces the subgraph of G_c induced by vertices V with a single fresh node $v(\tau.\eta)$ where $\tau.\eta$ is the name of tactic τ. Since the replacement of a subgraph with a single node is the standard *graph contraction* operation [16], we do not provide the implementation of Contract.



(a) Original proof TDG (b) A new tactic (c) Refactored proof

Fig. 11. Refactoring a proof using a new tactic.

Example 4.4. Consider the following Rocq proof and its TDG shown in Figure 11(a):

```
Lemma example:  (A -> D) -> (B -> C) -> ((A \/ B) -> (C \/ D)).
Proof. intro. intro. intro. destruct H1.
    - apply H in H1. right. exact H1.
    - apply H0 in H1. left. exact H1.
```

Also, consider the following tactic whose TDG is shown in Figure 11(b):

```
Ltac newTac h h' := apply h in h'. exact h'.
```

This tactic can be embedded into the proof using the witness functions $f_1 : [u \mapsto u_1, v \mapsto v_1]$ and $f_2 : [u \mapsto u_2, v \mapsto v_2]$, and Figure 11(c) shows the refactored TDG representing the following proof:

```
Proof. intro. intro. intro. destruct H1.
    - right. newTac H H1.
    - left.  newTac H0 H1.
```

Remark. If there are multiple collapsible embeddings of a tactic τ into a proof π such that these embeddings overlap, then the result of Refactor may not be unique, depending on which embedding is discovered first. For simplicity, the rest of the paper assumes that collapsible embeddings do not overlap; however, our implementation handles this situation (see Section 7).

5 Tactic Library Synthesis Problem

In this section, we formalize the *tactic discovery* problem addressed in the rest of this paper.

Definition 5.1 (Tactic discovery problem). Let Π be a corpus of proof scripts, and let $O : (\tau \times \Pi) \to \mathbb{R}$ be an objective function that evaluates the quality of a tactic τ on Π. The goal of the tactic discovery problem is to find a tactic τ that maximizes O (i.e., $\arg\max_\tau O(\tau, \Pi)$).

While the optimization objective O could be defined in a number of ways, we mainly consider *compression power* as our primary objective, even though our learning algorithm can also be adapted to handle other types of measures, as long as $O(\tau, \Pi)$ can be defined as:

$$O(\tau, \Pi) = f\big(\Pi, \textsc{Refactor}(\Pi, \tau)\big),$$

where f is some function $\Pi \times \Pi' \to \mathbb{R}$ and Refactor is the refactoring operation defined in the previous section. In particular, prior work on library learning in the program synthesis literature [8, 10, 18] has argued that reducing the overall size of a corpus—i.e., maximizing how much a learned abstraction "compresses" existing code—is an effective proxy for identifying broadly applicable patterns. Hence, we define an adaptation of compression power to the ITP setting.

Definition 5.2 (Compression power). Given proof corpus and tactic τ, let $\Pi' = \{\pi' \mid \pi \in \Pi \wedge \pi' = \textsc{Refactor}(\pi, \tau)\}$. Then, the compression power of τ modulo Π, denoted $\mathrm{CP}(\tau, \Pi)$, is:

$$\mathrm{CP}(\tau, \Pi) = \left(\frac{\sum_{\pi \in \Pi} \mathrm{Size}(\pi)}{\sum_{\pi' \in \Pi'} \mathrm{Size}(\pi')} \right)$$

Intuitively, the larger the compression power, the more effective the tactic is in reducing the corpus size. Finally, we can define the tactic library discovery problem as follows:

Definition 5.3 (Library synthesis problem). Let Π_1 be a corpus of proof scripts. The *tactic library synthesis* problem is to find a sequence of tactics $\tau_1, \ldots, \tau_n$ such that each τ_i is a solution to the tactic discovery problem for corpus $\Pi_i = \textsc{Refactor}(\tau_{i-1}, \Pi_{i-1})$.

Remark. One might consider defining the library synthesis problem as finding a set of tactics that *collectively* achieve the maximum compression power for the proof corpus. However, the compression power of a set of tactics depends on the *order* in which those tactics are applied. This makes such a definition either ill-formed or necessitates evaluating all possible permutations of tactic application orders, which is computationally infeasible and, in our small-scale experiments, yields no meaningful gain in compression. Consequently, we mirror prior library-learning works [8, 41] and formulate the library synthesis problem as finding a sequence of tactics that achieve maximal compression at each step.

6 Learning Tactic Libraries

We now describe our learning technique for discovering useful tactics from a given corpus.

6.1 Preliminary Definitions

We start this section by presenting some definitions that are useful for describing our algorithm.

Definition 6.1 (Witness set). Let G, G' be the TDG's of a tactic and proof script respectively. A witness set of G and G', denoted $\Upsilon(G, G')$, is the set of all witness functions proving that G is an isomorphic embedding into G'.

In this definition, we deliberately do not require a *collapsible* isomorphic embedding; specifically, f only needs to satisfy Definition 4.3. As we will demonstrate later in this section, our algorithm incrementally constructs the witness set and subsequently filters out non-collapsible candidates. This approach is necessary because the collapsibility criterion cannot be enforced incrementally.

Definition 6.2 (Embedding vector). An embedding vector Λ for a tactic τ and a proof corpus Π is a mapping from each proof $\pi \in \Pi$ to the witness set $\Upsilon(\mathsf{TDG}(\tau), \mathsf{TDG}(\pi))$.

That is, an embedding vector for τ maps each proof π to the witness set between τ and π.

Definition 6.3 (Tactic candidate). A tactic candidate Ψ for a proof corpus Π is a pair (G, Λ) where G is a TDG and Λ is an embedding vector of G for Π.

Given a tactic candidate $\Psi = (G, \Lambda)$, we write $\Psi.G$ to denote G and $\Psi.\Lambda$ to denote Λ. Intuitively, G represents a TDG that *could* be used to refactor some proofs in the corpus, and its embedding vector Λ allows us to efficiently compute witnesses for TDG's that are extensions of G. As we will

see in the next section, our tactic discovery algorithm uses the tactic candidate data structure as a key building block when searching for valid tactics.

Definition 6.4 (Frequency). Let $\Psi = (G, \Lambda)$ be a tactic candidate and Π a proof corpus. The *frequency* of Ψ in Π, denoted $F(\Psi, \Pi)$, is defined as follows:

$$F(\Psi, \Pi) = \sum_{\pi \in \Pi} \sum_{f \in \Upsilon(G, \mathsf{TDG}(\pi))} \mathbb{1}[\mathsf{IsCollapsible}(f, G, \mathsf{TDG}(\pi))]$$

where $\mathbb{1}[\cdot]$ is the standard indicator function and $\mathsf{IsCollapsible}(f, G, G')$ evaluates to true iff f defines a collapsible isomorphic embedding of G' into G.

Intuitively, the frequency of a tactic candidate Ψ counts the number of times that Ψ can be used in refactoring the proofs in the corpus. The higher the frequency of a tactic candidate, the more frequently it can be applied when refactoring proofs in the corpus. However, to evaluate how useful a tactic is in compressing the corpus size, we also need to take into account the size of the tactic. To this end, we define the *effectiveness* of a tactic candidate as follows:

Definition 6.5 (Effectiveness). Let $\Psi = (G, \Lambda)$ be a tactic candidate and Π a proof corpus. The *effectiveness* of Ψ in Π, denoted $\mathscr{E}(\Psi, \Pi)$, is defined as $\mathscr{E}(\Psi, \Pi) = (\mathsf{Size}(G) - 1) \times F(\Psi, \Pi)$.

In other words, effectiveness takes into account both the frequency and the size of the tactic. In this definition, we subtract 1 from the size of G, because when an embedding of the tactic is used for contracting the proof, the size of the proof shrinks by $\mathsf{Size}(G) - 1$. Intuitively, $\mathscr{E}(\Psi, \Pi)$ can be used to accurately characterize the compression power of a tactic – the higher the value of $\mathscr{E}(\Psi, \Pi)$, the more effective Ψ is for compressing the proof corpus Π.

As we will see in the next section, our algorithm uses this metric as a pruning criterion during search.[2]

```
1:  procedure LearnTactic(Π)
    input: A proof corpus Π
    output: A tactic τ that achieves maximum compression of Π
2:      G ← ConstructTDGs(Π)
3:      R ← LearnGraphGrammar(G)
4:      W ← InitWorklist(R, Π)
5:      r ← {Candidate = ⊥, Eff = 0}
6:      while W ≠ ∅ do
7:          Ψ ← W.dequeue()
8:          if UpperBound(Ψ, Π) < r.Eff then continue
9:          if 𝓔(Ψ, Π) > r.Eff then
10:             r ← {Candidate = Ψ, Eff = 𝓔(Ψ, Π)}
11:         W.enqueue(Expand(Ψ, R, Π))
12:     return MakeTactic(r.Candidate)
```

Fig. 12. Procedure for learning a tactic that results in maximum compression of proof corpus Π. Procedure names that are in SmallCaps font are defined in separate algorithms and explained in the rest of this section. On the other hand, procedure names that are written in Sans Serif font are only explained in text. At a high level, this algorithm iteratively explores tactic candidates, pruning those whose expansions cannot yield a higher compression power than a previously encountered tactic.

6.2 Tactic Discovery Algorithm

In this section, we describe our algorithm for learning a single tactic that maximizes the desired objective. Our top-level algorithm, called LearnTactic, is presented in Figure 12: It takes in a proof corpus Π and returns a single tactic τ that maximizes $\mathrm{CP}(\tau, \Pi)$. The algorithm starts by constructing TDG's for each proof in the corpus and then calls the LearnGraphGrammar procedure (described

[2]To adapt our learning algorithm in Section 6.2 to optimization objectives other than compression power, one needs to define a suitable instantiation of function $\mathscr{E}$ for the corresponding objective.

later), for learning a graph grammar R that can be used to construct TDG's of tactic candidates. Intuitively, R encapsulates recurring patterns within the proof corpus and provides a set of rules to guide the exploration of potential tactic candidates.

Lines 4-11 of LearnTactic systematically explore different tactic candidates. Specifically, the algorithm initializes the worklist $\mathcal{W}$ with all the single-node tactics from the grammar R and initializes a record r to track the best tactic discovered so far. In each iteration of the loop (lines 6–11), the algorithm dequeues an existing tactic candidate Ψ and decides whether or not to continue expanding it. In particular, if it can prove that *no expansion* of Ψ will result in a higher compression power than the best discovered tactic, it discards Ψ from the search space. Otherwise, it proceeds to compute the actual effectiveness of Ψ using the function $\mathscr{E}(\Psi, \Pi)$ from Definition 6.5. If $\mathscr{E}(\Psi, \Pi)$ exceeds that of the previous best tactic, the result r is updated to Ψ. Additionally, since expansions of Ψ might have even higher compression power, the algorithm invokes the Expand procedure to obtain other candidate tactics that can be produced by expanding Ψ using the productions in R. Upon termination, r contains a tactic that maximizes compression power for the entire corpus. In the remainder of this section, we elaborate on the auxiliary procedures used in LearnTactic.

Learning Graph Grammar. While our tactic learning algorithm is inspired by top-down enumerative program synthesis [23, 38], a key difference is that we do not have a context-free grammar defining the space of possible TDGs. Hence, our algorithm first learns a graph grammar that can be used to construct TDG's of possible tactics by analyzing the proof corpus.

$$\frac{G \in \mathcal{G} \quad v(\eta) \in \mathsf{Nodes}(G) \quad v'(\eta') \in \mathsf{Nodes}(G) \qquad \theta = \{(\alpha, \beta) \mid (v(\eta), v'(\eta'), \alpha, \beta) \in \mathsf{Edges}(G)\}\}}{\mathcal{G} \vdash \eta \to (\eta', \theta)}$$

Fig. 13. Inference rule defining the LearnGraphGrammar procedure

Definition 6.6 (TDG grammar). A TDG grammar is defined by a set of production rules of the form $\eta \to (\eta', \theta)$, where each rule specifies that a single-node graph $G = (\{v(\eta)\})$ can be transformed into a new graph $G' = (\{v(\eta), v(\eta')\}, \{(v(\eta), v(\eta'), \alpha, \beta) \mid (\alpha, \beta) \in \theta\})$.

Intuitively, each production $\eta \to (\eta', \theta)$ states that a node labeled η in a TDG can be connected to other nodes labeled η' via arcs whose labels are specified by θ. These productions are mined from the proof corpus based on the following observations:

(1) If a certain tactic never occurs in the proof corpus, it also cannot appear in any learned tactic.
(2) If there is no dependency between a pair of tactics in the corpus, there is no point in learning a tactic that involves a spurious dependency between them.
(3) If there exists a dependency between a pair of tactics in a proof π in the corpus, then the tactic TDG should either not include that dependency or, else, it should include *all* the input-output dependencies — otherwise, we cannot find a *collapsible* isomorphic embedding.

Based on these ideas, Figure 13 presents the graph grammar learning procedure as a single inference rule deriving a judgment of the form: $\mathcal{G} \vdash \eta \to (\eta', \theta)$ The productions R used in the LearnTactic algorithm include the set of all rewrite rules r such that $\mathcal{G} \vdash r$ according to Figure 13.

Worklist Initialization. Next, we consider the InitWorklist procedure presented in Figure 14. The idea behind this procedure is very simple: It constructs a set of single node TDG's based on the non-terminals in graph grammar R. Then, for each such single node TDG, it computes the tactic's embedding vector by going over each proof in the corpus. Hence, the worklist is initialized to tactic candidates that consist of single node TDG's and their corresponding embedding vector.

```
1: procedure InitWorklist(R, Π)
   input: Graph grammar R, proof corpus Π
   output: A worklist of tactic candidates 𝒲
2:     S = {η | (η → (η′, E)) ∈ R}; 𝒲 ← ∅
3:     for all η ∈ S do
4:         G = ({v(η)}, ∅)
5:         Λ ← [π ↦ ⊥ | π ∈ Π]
6:         for all π ∈ Π do
7:             Λ[π] ← GetWitness(G, TDG(π))
8:         𝒲.enqueue((G, Λ))
9:     return 𝒲
```

Fig. 14. The InitWorklist procedure for constructing the initial worklist used in LearnTactic.

```
1: procedure Expand(Ψ, R, Π)
   input: A tactic candidate Ψ = (G, Λ), graph grammar R, corpus Π
   output: A new set of tactic candidates
2:     Θ ← ∅
3:     for all η → (η′, θ) ∈ R do
4:         for all v ∈ FindNodes(G, η) do
5:             v′ ← Fresh(η′)
6:             Θ ← Θ ∪ Apply(Ψ, v, v′, θ, Π)
7:             for all v_i ∈ FindNodes(G, η′) do
8:                 Θ ← Θ ∪ Apply(Ψ, v, v_i, θ, Π)
9:     return Θ
```

Fig. 15. Procedure for expanding a given tactic candidate Ψ using graph grammar R.

Generating New Tactic Candidates. Recall that the LearnTactic procedure generates new tactic candidates by calling the Expand procedure, which is presented in Figure 15. This algorithm takes as input an existing tactic candidate $\Psi = (G, \Lambda)$ and the learned graph grammar R and produces a new set Θ of tactic candidates, where each $\Psi_i \in \Theta$ is an expansion of Ψ. Specifically, let $\Psi_i = (G_i, \Lambda_i)$ be one of the new tactic candidates produced by Expand. Here, every G_i is a strict supergraph of G and always contains additional edges that are not in G. Additionally, some of these G_i's may also contain a *single* additional node that is not in G.

As shown in Figure 15, the Expand procedure considers each production $\eta \rightarrow (\eta', \theta) \in R$ (line 3) and locates all nodes $v \in G$ with η. Then, at line 6, it calls Apply (discussed later) to produce a new tactic candidate G' that includes a fresh node v' labeled η', along with arcs labeled $(\alpha, \beta) \in \theta$ between v and v'. Additionally, since there may be existing nodes labeled η' in G, the inner loop at lines 7–8 also adds additional edges from v to each existing node v_i labeled η'.

We now also briefly explain the auxiliary Apply procedure defined in Algorithm 16. Given a tactic candidate $\Psi = (G, \Lambda)$ where G contains a node v, Apply first constructs a new TDG G' that includes a (possibly new) node v' as well as edges between v and v' with labels θ. However, since a tactic candidate also contains the embedding vector for the tactic, the loop in lines 7–10 constructs a new embedding vector for G' by extending the existing witness functions in Λ. Finally, if the frequency of the resulting tactic candidate is less than two, this means that the new tactic candidate (or any of its future expansions) are *not* useful for compressing the proof corpus; hence, Apply returns the new tactic candidate Ψ' only if $F(\Psi', \Pi)$ is at least 2.

Pruning Non-Optimal Tactic Candidates. Finally, we consider the UpperBound procedure, presented in Figure 17, for deriving an upper bound on the effectiveness of a given tactic candidate Ψ on compressing the size of the proof corpus Π. The idea behind UpperBound is fairly straightforward: For every embedding of G in a proof script π, we compute the *maximum extension* of G that can still be embedded in π:

Definition 6.7. (Maximum extension) The maximum extension of G in G' is a graph G_e with the following properties: (1) G is a subgraph of G_e; (2) G_e is a subgraph of G'; (3) $\text{root}(G_e) = \text{root}(G)$; and (4) For any other graph G'_e satisfying (1) (2) and (3), we have $\text{size}(G'_e) \leq \text{size}(G_e)$.

The idea is that, by summing up the sizes of all these extensions across all embeddings into the proof corpus, we can obtain an upper bound $\mathscr{E}(\Psi', \Pi)$ for any Ψ' that is an extension of Ψ. This is precisely what the UpperBound procedure in Figure 17 computes.

```
1:  procedure Apply(Ψ, v, v′, θ, Π)
    input: A tactic candidate Ψ = (G = (V, E), Λ),
    nodes v, v′, edge labels θ, and proof corpus Π
    output: Empty set or singleton tactic candi-
    date
2:      V′ ← (V ∪ {v′})
3:      E′ ← {E ∪ {(v, v′, α, β) | (α, β) ∈ θ})
4:      G′ ← (V′, E′)
5:      Λ′ ← [π ↦ ∅ | π ∈ Dom(Λ)];
6:      Ψ′ ← (G′, Λ′)
7:      for all π ∈ Dom(Λ) do
8:          for all f ∈ Λ[π] do
9:              F ← Extend(f, G′, TDG(π))
10:             Λ′[π] ← Λ′[π] ∪ F
11:     if F(Ψ′, Π) ≥ 2 then return {Ψ′}
12:     else return ∅
```

Fig. 16. Procedure for applying a production from the graph grammar on nodes v, v'.

```
1:  procedure UpperBound(Ψ, Π)
    input: A tactic candidate Ψ = (G, Λ) and a
    proof corpus Π
    output: Upper bound on the effectiveness of
    Ψ
2:      ub ← 0
3:      for all π ∈ Dom(Λ) do
4:          for all f ∈ Λ[π] do
5:              G′ ← ApplyWitness(G, f)
6:              G_e ← MaxExtend(G′, TDG(π))
7:              ub ← ub + size(G_e) − 1
8:      return ub
```

Fig. 17. Computes upper bound on the compression power of any extension of Ψ. ApplyWitness applies function f to G to obtain a graph G'. MaxExtend(G, G') finds a graph G_e such that (1) G_e is subgraph of $TDG(\pi)$ and supergraph of G', (2) root(G_e) = root(G') and (3) G_e has maximum size among all graphs that satisfy (1) and (2).

Tactic Library Synthesis. Finally, to generate a *library* of tactics, our algorithm synthesizes a single tactic by calling LearnTactic, then refactors the corpus using this tactic, and repeats this process until no more tactics can be learned.

Discussion: Learning with Other Objectives. While our algorithm follows standard practice in optimizing *compression power* [8, 10, 18], it naturally generalizes to other objectives. For example, one can consider a broader optimization function of the form $\alpha \times \mathrm{CP}(\tau, \Pi) - \beta \times F(\tau)$, where F penalizes tactics that do not satisfy certain syntactic or semantic criteria, such as requiring too many arguments or being too narrowly applicable. Adapting our method to alternative objectives requires modifying two aspects of the learning algorithm: (1) the objective function itself, captured by $\mathscr{E}$, and (2) the UpperBound procedure, which estimates an upper bound on the objective value. For instance, for an objective of the form $\alpha \times \mathrm{CP}(\tau, \Pi) - \beta \times F(\tau)$, a straightforward implementation of UpperBound could reuse our existing technique for over-approximating compression power while conservatively assuming that the penalty term is zero.

7 Implementation

We have implemented the proposed algorithm as a new tool called TacMiner, which consists of about 5000 lines of Java code and utilizes the external Coq-SerAPI library [3] to parse proof scripts and extract any information necessary for constructing TDGs. The current version of TacMiner only supports Ltac [15], as it remains the dominant tactic language in existing Rocq developments.

Handling Overlapping Embeddings. In the technical presentation, we assume that, if there are multiple isomorphic collapsible embeddings of a tactic into a proof, then they are non-overlapping; however, our implementation does not make this assumption. In particular, when computing an upper bound on effectiveness during the tactic discovery process, we assume that all embeddings can be used for refactoring, giving a conservative upper bound on the effectiveness of a given tactic candidate. Similarly, when computing the actual effectiveness of a tactic, we perform backtracking search to find a disjoint subset of embeddings that maximizes the effectiveness score.

Converting TDGs to Proof Scripts. Since our approach refactors a proof at the TDG level, we need to convert it back to a valid Rocq proof script. As discussed in Section 4, we can do this by performing a topological sort of the TDG. However, if a tactic produces multiple sub-goals as its output, we need to ensure that each sub-goal is processed as a separate branch. Our algorithm for converting TDGs to proof scripts performs additional analysis to keep track of this information and ensures that tactics that are used for discharging the same sub-goal appear in the same branch, subject to the topological sorting constraints within that branch.

Supported Tactics. Our method places no restrictions on the tactics used in the input proof scripts. Any tactic that can be parsed using SerAPI, including advanced tactics like `match goal` and `eapply`, as well as user-defined custom tactics, is supported. Constructs that bundle combinators (e.g., `try`, `first`, `repeat`) with tactics are treated an atomic tactics, whereas tactics connected by the sequencing operator `;` are explicitly decomposed into their individual invocations. There are, however, restrictions on what our system can *learn*. We do not synthesize tactics whose *control flow* depends on inspecting the current proof state (e.g., `match goal`)[3], because a TDG records only the executed trace and omits unexecuted branches. Similarly, while our approach can learn tactics involving existential variables (`evars`), it does so only when their effects remain localized within the proof states that arise from their instantiation.

Table 2. Summary of Benchmarks. "Total size" reports total number of pre-defined tactic applications within the domain, and "Avg. size / proof" reports average proof size in terms of the number of tactic applications.

Domain	Topic	Description	# of Proof	Total size	Avg size / proof
CoqArt	IndPred	Inductive Predicates	61	620	10
	SearchTree	Search Trees	48	781	16
	Reflection	Proof by Reflection	33	668	20
Program Logics	Hoare	Hoare logic	65	1479	23
	Separation	Heap properties	58	592	10
	Seplog	Separation logic	70	1111	16
	CSL	Concurrent separation logic	47	1282	27
Comp-Cert	RegAlloc	Register allocation with validation	31	467	15
	LiveRange	Proofs for live ranges computation	32	903	28
	Needness	Abstract domain for needness analysis	103	1759	17
	RTLSpec	Abstract specification for RTL generation	55	1413	26
BigNums	NMake	Big natural numbers	105	1465	14
	ZMake	Big integers	43	801	19
	QMake	Big rational numbers	68	1392	20

8 Evaluation

We now describe our experimental evaluation that is designed to answer the following questions:

- **RQ1:** How effective is our proposed technique in learning tactics compared to a prior approach [41] that learns tactics using anti-unification?
- **RQ2:** How useful are the learned tactics in terms of compression rate?
- **RQ3:** Can our learned tactics help with proof automation?
- **RQ4:** How data efficient is our learning algorithm?
- **RQ5:** What impact do the key ingredients of our learning algorithm have on running time?

[3]In contrast, tactics such as `intros`, `simpl`, or `auto` are supported: although their effects depend on the proof state, their execution does not require branching on it.

8.1 Experimental Setup

Benchmarks. To evaluate our approach, we collected 918 proofs across four different sources: a textbook called "Interactive Theorem Proving and Program Development"[5] (also known as CoqArt[12]), the formally verified C compiler CompCert [35], the BigNums arbitrary-precision arithmetic library [52], and formalizations of various program logics [34]. For each of these sources, we choose these sub-domains that satisfy certain criteria, such as containing a threshold number of proofs ($\geq$ 30 across all sources). Table 2 provides a summary of our experimental benchmarks, including a description of the sub-domain that the proofs pertain to and the number of proofs in each sub-domain. The last two columns in the table show the total number of tactic applications in each domain and the average number of tactics per proof. Overall, these benchmarks span a wide spectrum of proof domains, ranging from introductory proofs in CoqArt to expert-level developments in large-scale systems like CompCert. Furthermore, they include a mix of programming languages (PL)-specific benchmarks—such as those involving program logics—and formalizations with broader mathematical utility, such as those for infinite-precision arithmetic.

Baseline. Because no prior work exists on tactic library learning for Rocq proofs, we will address our first two research questions and compare our approach against a baseline by adapting the closest relevant prior work: tactic discovery via anti-unification, proposed in Peano, a system designed to automate K12 math proofs. In particular, Peano represents proofs as sequences of

Table 3. Results for evaluating the effectiveness of the tactic learning algorithm

	Topic	Tool	# Tactics Learned	Avg Tactic Size	Max Tactic Size	Tactic Usage Count
CoqArt	IndPred	TacMiner	31	2.8	14	83
		Peano	16	2	2	48
	SearchTree	TacMiner	54	3.4	17	161
		Peano	20	2.4	6	31
	Reflection	TacMiner	38	3.3	11	100
		Peano	20	2.6	9	62
Program Logics	Hoare	TacMiner	87	3.2	15	231
		Peano	18	2.2	4	55
	Separation	TacMiner	36	3.1	8	93
		Peano	26	2.5	7	68
	Seplog	TacMiner	69	3.4	13	188
		Peano	16	2.3	4	56
	CSL	TacMiner	82	3.6	19	214
		Peano	14	2.1	3	44
CompCert	RegAlloc	TacMiner	18	8.8	34	50
		Peano	8	7	18	13
	LiveRange	TacMiner	61	3.8	16	162
		Peano	17	2.2	5	42
	AbsDomain	TacMiner	100	4.3	23	243
		Peano	39	2.1	3	135
	RTLSpec	TacMiner	93	3.4	13	260
		Peano	25	2	4	58
BigNums	NMake	TacMiner	91	2.9	9	263
		Peano	29	2.3	5	84
	ZMake	TacMiner	56	4.8	19	132
		Peano	23	3.4	10	48
	QMake	TacMiner	95	3.6	21	250
		Peano	36	2.2	4	98
	Overall	TacMiner	918	3.6	34	2430
		Peano	310	2.4	10	842

actions, and, for each pair of subsequences from different proofs, Peano applies anti-unification to find the least general generalization between them. If the two sequences invoke the same actions but on different inputs, anti-unification identifies the shared structure while introducing variables for the differing parts. Similar to our approach, Peano also aims to maximally compress the existing proofs.

Since the implementation of the tactic induction method in Peano does not work on Rocq proofs, we re-implemented their technique to extract tactics from a corpus of Rocq proof scripts. Towards this goal, we represent Rocq proof scripts as a sequence of tactics (i.e., "actions" in Peano terminology) and use the same anti-unification approach to learn custom tactics. However, this syntactic approach does not guarantee that an extracted tactic can be successfully used to refactor the proof from which it was derived. Hence, given a tactic library learned using this baseline, we perform post-processing to discard tactics that cannot be used to re-factor *any* proof in the corpus.

Computational Resources All of our experiments are conducted on a machine with an Apple M2 CPU and 24 GB of physical memory, running on the macOS operating system.

8.2 Evaluation of Learned Tactics

To answer our first research question, we use both TacMiner and Peano to extract tactics from each domain. This evaluation, summarized in Table 3, treats the entire proof corpus of each benchmark as the training data. The figure provides statistics about the number of tactics learned by each technique and quantitative metrics to assess their quality, including average size of learned tactics, the maximum size among all learned tactics, and the total number of times the learned tactics can be applied in the proof corpus. As shown in this table, our proposed method learns substantially more tactics compared to Peano (around 3× more across all benchmarks) and the learned tactics tend to be larger on average (1.5×). Furthermore, across all benchmarks, the tactics learned by Peano can be applied a total of 842 times, whereas those learned by TacMiner can be used 2430 times. We believe these results demonstrate that our proposed TDG abstraction and learning algorithm allow for more effective tactic discovery compared to a baseline that extracts tactics from a syntactic representation of proofs.

> **Results for RQ1:** TacMiner extracts around 3× more tactics compared to Peano. Furthermore, the tactics learned by TacMiner are both larger and more frequently applicable in the corpus.

8.3 Effectiveness of the Learned Tactics for Proof Refactoring

To answer our second research question, we evaluate how useful the learned tactics are in refactoring previously unseen proofs. To perform this experiment, we split the benchmarks into two separate training and test sets. We use the training set for tactic discovery but evaluate the usefulness of learned tactics *only* on the test set. For this experiment, we use 65% of the proofs for training (selected via an automated sampling script), and the remaining 35% as the test set. We further evaluate the effectiveness of our approach for different training vs. test set ratios in Section 8.5.

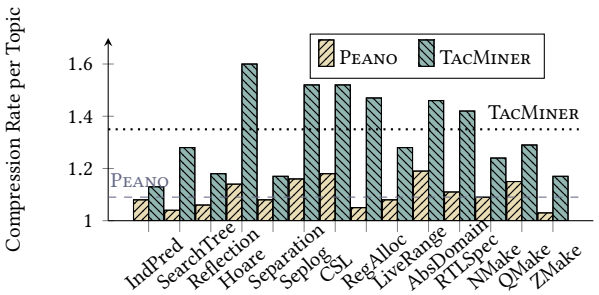


Fig. 18. Average compression power per topic. The dotted lines denote the total across topics for each tool.

The results of this evaluation are presented in Figure 18 where the y-axis shows the compression power (Def 5.2) of the learned tactics on the test set. For each category of benchmarks, we show the compression power achieved by both TacMiner and Peano. As we can see from this figure, TacMiner results in significantly higher compression power compared to Peano in *all* categories. Across all benchmarks, the compression power of TacMiner is 1.35 for TacMiner vs 1.1 for Peano—this means that, using the tactics learned by TacMiner, the size of the proof corpus can be reduced by 26%, whereas the tactics learned by Peano result in a reduction of only 9%. Furthermore, for the Hoare logic proofs, TacMiner achieves the maximum compression of 1.6, indicating that the learned tactics for this domain are particularly effective at simplifying other proofs in the same domain. That is, for this domain, the refactored proofs are approximately 63% of their old size.

Results for RQ2: Using the tactics learned by TacMiner, the size of the proof corpus can be reduced by 26%. In contrast, Peano's tactics can reduce the corpus size by only 9%.

8.4 Using Learned Tactics for Proof Automation

Next, we evaluate whether our learned custom tactics can help a proof automation tool. To perform this investigation, we modify Copra [50], a state-of-the-art proof automation tool based on Large Language Models (LLMs), to leverage our learned tactics. To adapt Copra to use custom tactics, we modify the prompt provided to the LLM for in-context learning [9]. Specifically, for each custom tactic learned by TacMiner, we add the tactic definition as Copra's context as well as one example showing a proof state where that tactic was used (after being refactored by TacMiner) along with the tactic's invocation (i.e., its arguments) that took place in the refactored proof. These two pieces of information should, in principle, allow Copra to leverage custom tactics. For this experiment, we use OpenAI's `gpt-4o-2024-10-06` [27] as the underlying LLM.

Table 4. Comparison of theorem proving success rates.

Method	Theorems proved
Copra with built-in tactics	11/50 (22%)
Copra with Peano tactics	18/50 (36%)
Copra with TacMiner tactics	30/50 (60%)

To evaluate the utility of learned tactics for downstream proof automation, we selected a set of 50 theorems from the same sources listed in Table 2. These theorems were chosen based on two criteria. First, they are theorems for which Copra, with a smaller computational budget than our official evaluation, was able to make partial progress—successfully solving some sub-goals but failing to complete the entire proof. This ensures that the benchmarks are neither trivial nor intractable, providing a meaningful setting for evaluating the added value of tactic learning. Second, we required that the corresponding refactored proof makes use of at least one tactic discovered by our system, ensuring that the learned tactics are relevant and can plausibly aid automation. Together, these criteria yield a benchmark set well-suited for measuring the practical benefits of tactic reuse in proof automation workflows.

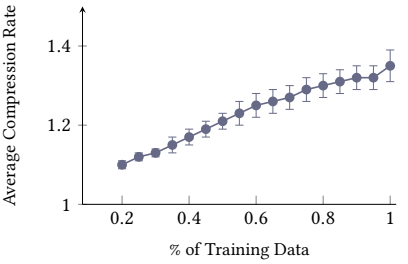


Fig. 19. Data efficiency for TacMiner. The error bar reflects the standard error.

The results of this evaluation are presented in Table 4. The vanilla Copra baseline using only built-in tactics can prove 11 out of the 50 theorems, resulting in a success rate of 22%. We then independently evaluate Copra augmented with learned tactics from Peano and TacMiner on the same 50 theorems. Copra supplied with the tactics learned by Peano can prove 7 additional

theorems, increasing success rate to 36%. Finally, using the custom tactics learned by our method, Copra can prove 19 additional theorems over the baseline, increasing the success rate to 60%. These results demonstrate that (1) custom tactics can be useful for improving proof automation, and (2) tactics learned by our method are more useful for proof automation compared to the Peano baseline.

Results for RQ3: Using the custom tactics learned by TacMiner, we are able to increase the success rate of an LLM-based proof automation tool from 22% to 60%.

8.5 Data efficiency of proposed tactic learning technique

The results of this evaluation are presented in Figure 19. As expected, the larger the number of training benchmarks, the more effective the learning technique. However, even if we only use 25% of the benchmarks for training, TacMiner still achieves a compression power of around 1.13, which is higher than that achieved by Peano with a much larger training set.

Results for RQ4: While TacMiner benefits from a larger training corpus, it can still achieve significant compression on the test set as we decrease the size of the training data.

8.6 Ablation Studies for Tactic Discovery Algorithm

To answer our final research question, we present the results of an ablation study in which we disable some of the key components of our tactic learning algorithm. In particular, we consider the following two ablations of TacMiner:

- **GrammarABL:** This ablation performs a limited form of grammar learning.[4] In particular, it learns graph grammar rules of the form $\eta \to \eta'$ instead of $\eta \to (\eta', \theta)$, meaning that it does not have prior knowledge about the argument dependencies between tactics.
- **PruningABL:** This ablation does not use the pruning procedure (UpperBound) from Section 6.2.

The results of this ablation study are presented in Figures 20 (for Program Logics), 21 (for CompCert), 22 (for CoqArt), and 23 (for BigNums). Here, the x-axis shows the number of total tactics learned, and the y-axis shows the cumulative learning time in seconds. Note that the y-axis uses log scale. As we can see from these plots, both graph grammar learning and pruning via upper bound estimation have a huge impact on the running time of the tactic discovery algorithm. While TacMiner can terminate on the entire corpus in about 13 minutes, both of the ablations fail to terminate within a 30-minute time limit.

Results for RQ5: Both of our algorithmic optimizations (namely, grammar learning and pruning method) significantly reduce the learning algorithm's running time.

[4]We also tried switching off grammar learning entirely, but since it performs *extremely poorly*, we only report the results of a limited form of grammar learning.

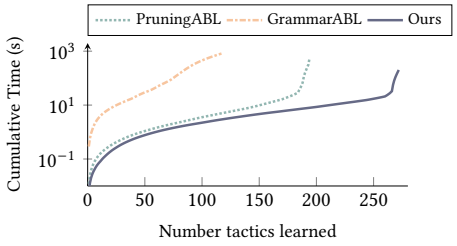


Fig. 20. Learning curve for ProgramLogic.

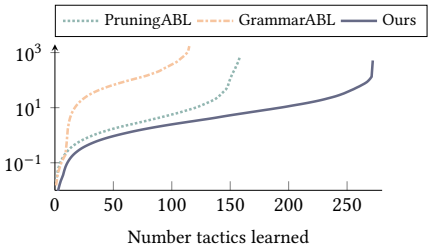


Fig. 21. Learning curve for CompCert.

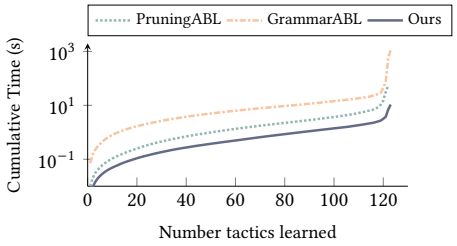


Fig. 22. Learning curve for CoqArt.

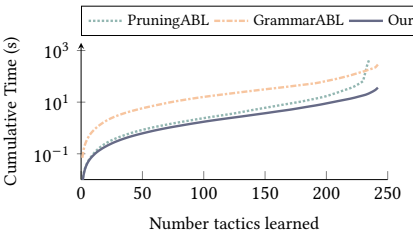


Fig. 23. Learning curve for BigNums.

9 Related Work

In this section, we survey work that is most closely related to our proposed tactic discovery method.

Tactic Learning. Previous work, particularly Peano [41], has focused on learning tactic libraries through anti-unification [40]. This method is syntactic in nature: proofs are modeled as sequences of operations, and new tactics are derived when repeated sub-sequences are identified across multiple proofs. While effective in identifying recurring patterns, this approach is limited by its inability to capture deeper semantic relationships between proof steps or perform meaningful proof refactoring. Additionally, Peano operates within a restricted, custom tactic language.

Proof Automation Through Tactic Prediction. A significant body of recent work has used neural models for *tactic prediction*: given the current proof state, predict a tactic invocation to make progress in the proof [4, 6, 7, 13, 26, 45, 51, 55]. These models can either fully automate proofs or collaborate with human users [48]. Typically, the underlying prediction models are trained on human-written proof steps in the language of interest, and then used to guide a search algorithm interacting with the theorem proving environment. More recent works also explore using LLMs for proof automation [24, 37, 50], finding that broad pre-training also makes current LLMs capable of predicting tactics. Our work complements these methods by alleviating the burden on the tactic predictor: since proofs using higher-level tactics are shorter, they can be generated with fewer calls to the predictor. Systems based on LLMs that are capable of in-context learning, such as Copra [50], make this integration particularly convenient, since they can attempt to use custom tactics without having prior knowledge about them, as long as they are given to the LLM in their context window.

Library Learning. Library learning in code-related domains aims to automatically discover reusable components in both programs and formal proofs. In the context of code reuse, researchers have explored various methods for code library learning, including anti-unification techniques [17, 28], program synthesis algorithms [8, 18], and e-graphs [10]. For theorem proving, library learning has primarily focused on extracting reusable lemmas. Kaliszyk and Urban [29] introduced a method for automatically extracting lemmas from the Mizar system to assist in proving additional theorems. This idea was expanded by REFACTOR [59], which applied similar lemma extraction techniques

to the Metamath system. More recently, Xin et al. [56] integrated lemma proposals from large language models (LLMs) into neural theorem proving for the Lean prover. Our work on *tactic* learning provides a complementary addition to these approaches, as tactics represent imperative, untyped proof construction steps rather than specific mathematical facts. While lemma learning focuses on identifying reusable truths, tactic learning captures and generalizes common proof strategies, allowing proofs to be written more concisely and at a higher level of abstraction.

Semantic Code Refactoring Several approaches have been developed for semantic code refactoring across various domains. Revamp [39] focuses on refactoring Abstract Data Types (ADTs) using relational constraints. In other areas, researchers have explored different techniques: using program invariants to detect refactoring candidates [30], employing type constraints to refactor class hierarchies [49, 53], and applying program analysis techniques to refactor Java generics [2]. Additionally, Migrator [54] addresses the refactoring of database programs in response to schema updates. However, these approaches target code refactoring rather than proof refactoring.

Guided Enumerative Synthesis. Our tactic candidate enumeration procedure shares similarities with guided enumerative synthesis techniques from program synthesis literature [8, 11, 18, 23, 33, 42, 46, 58]. While most of these works aim to synthesize complete programs based on input/output examples or other specifications, Stitch [8] also employs a corpus-guided top-down approach to learn library abstractions for programs. However, as noted in our introduction, Stitch primarily focuses on generalizing concrete expressions to lambda abstractions. In contrast, our work emphasizes leveraging the semantics of tactic execution to discover new usable patterns.

Graph-Based Program Abstractions. Tactic Dependence Graphs (TDGs) are inspired by graph-based program abstractions [1, 22, 31, 44, 47], such as control-flow and data-flow graphs, commonly used in program analysis. Broadly, such abstractions represent either the program's control flow (e.g., call graph) or dependencies between data (e.g., points-to graph) and are widely employed for tasks like optimization and security analysis (e.g., malware detection, code clone identification [19, 20, 25, 36, 43, 60]). However, TDGs differ in their purpose and design, focusing on logical proof dependencies between tactics rather than control or data dependencies. Notably, TDGs abstract away irrelevant syntactic differences between proofs (e.g., subgoal naming or tactic order) and concentrate on the semantic relationships between tactic applications.

10 Discussion

The proof refactoring and tactic discovery framework presented in this paper enables multiple applications in interactive theorem proving. In this section, we discuss three possible use cases of our approach in the overall proof engineering workflow.

Improving Proof Automation. By encapsulating recurring proof patterns into higher-level tactics, our approach helps automated tools operate at a more abstract level. Instead of repeatedly generating low-level proof steps, proof automation tools can leverage learned tactics as higher-level building blocks. This strategy can improve scalability, since the search space at a higher level of abstraction is smaller. This approach also facilitates a form of curriculum learning, where newly discovered tactics serve as building blocks for more advanced proofs. As shown in our evaluation in Section 8.4, the tactics learned by TacMiner already significantly improve the success rate of a state-of-the-art proof automation tool. Notably, this improvement is observed even though current automation tools are not explicitly trained to exploit custom tactics. As proof automation methods evolve to take better advantage of custom tactics, we expect these gains to become even more pronounced.

Interactive Tactic Suggestion. Our method can also assist proof engineers by suggesting potential tactics, highlighting repeated patterns in their proofs that might otherwise go unnoticed. In this

interactive mode, we expect users to take inspiration from the suggested tactics while refining them to fit their domain expertise and personal style. For instance, if a learned tactic appears too general—such as requiring an excessive number of arguments—users can specialize it with concrete parameters to make it more practical. Likewise, multiple tactics proposed by TacMiner can be merged into a single tactic that dynamically selects which one to apply based on the structure of the proof goal. Rather than prescribing a fixed way to restructure proofs, the system provides flexible recommendations that users can adapt as needed.

Proof Refactoring. A third use case of our tool is automatically or interactively refactoring existing proofs, particularly in large projects that require long-term maintenance. As definitions change or new lemmas are introduced, proof engineers often need to define new tactics and restructure existing proofs to incorporate them. However, manually refactoring proofs in this way is labor-intensive. Our method simplifies this process by identifying where a given tactic—whether user-defined or adapted from a tactic discovered by TacMiner —can replace existing sequences of proof steps.

11 Conclusion

We introduced a new approach to tactic discovery using Tactic Dependency Graphs (TDGs), which abstract away syntactic variations while capturing the logical dependencies between tactic applications. TDGs facilitate both the learning of custom tactics and the refactoring of existing proofs into more concise, modular forms. We implemented this method in a tool called TacMiner and evaluated it on several domains, including various program logics, arbitrary-precision arithmetic, and compiler transformations. We also compared our approach against an anti-unification-based tactic discovery method from prior work (Peano) and demonstrated the advantages of our approach in terms of the number and quality of the learned tactics: TacMiner learns around 3× as many tactics compared to Peano and achieves an compression rate of 1.35 on the test set, reducing the size of the corpus by 26%. We also showed that the tactics learned by TacMiner are useful for improving proof automation: When Copra, a state-of-the-art proof automation tool, is supplied with the custom tactics learned by TacMiner, its success rate yields a relative increase of 172%. Overall, our work shows that tactic discovery provides a promising avenue for both proof refactoring and automation.

Data-Availability Statement

Our artifact, along with the benchmark suite used for evaluation, is available on Zenodo [57].

Acknowledgments

We thank our anonymous reviewers for their helpful feedback and support. This work was conducted in a research group supported by the National Science Foundation under Grant No. 1762299, Grant No. 1918889, Grant No. 1908304, Grant No. 1901376, Grant No. 2120696, Grant No. 2210831, and Grant No. 2319471.

References

[1] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization* (Urbana-Champaign, Illinois). Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/800028.808479

[2] John Altidor and Yannis Smaragdakis. 2014. Refactoring Java generics by inferring wildcards, in practice. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 271–290. https://doi.org/10.1145/2660193.2660203

[3] Emilio Jesús Gallego Arias. [n. d.]. Coq SerAPI Library. https://github.com/ejgallego/coq-serapi

[4] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 454–463. https://proceedings.mlr.press/v97/bansal19a.html

[5] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg.

[6] Lasse Blaauwbroek and Herman Urban, Josefand Geuvers. 2020. The Tactician. In *Intelligent Computer Mathematics*, Christoph Benzmüller and Bruce Miller (Eds.). Springer International Publishing, Cham, 271–277.

[7] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic Learning and Proving for the Coq Proof Assistant. In *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 73)*, Elvira Albert and Laura Kovacs (Eds.). EasyChair, 138–150. https://doi.org/10.29007/wg1q

[8] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* 7, POPL, Article 41 (jan 2023), 32 pages. https://doi.org/10.1145/3571234

[9] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, Vol. 33. 1877–1901.

[10] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proc. ACM Program. Lang.* 7, POPL, Article 14 (jan 2023), 29 pages. https://doi.org/10.1145/3571207

[11] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

[12] Coq Community. 2024. Coq-Art: Practical Foundations for Programming with Dependent Types. https://github.com/coq-community/coq-art. Accessed: 2024-11-04.

[13] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[14] Ana de Almeida Borges, Annalí Casanueva Artís, Jean-Rémy Falleri, Emilio Jesús Gallego Arias, Érik Martin-Dorel, Karl Palmskog, Alexander Serebrenik, and Théo Zimmermann. 2023. Lessons for Interactive Theorem Proving Researchers from a Survey of Coq Users. In *14th International Conference on Interactive Theorem Proving (ITP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:18. https://doi.org/10.4230/LIPIcs.ITP.2023.12

[15] David Delahaye. 2000. A tactic language for the system Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning* (Reunion Island, France) *(LPAR'00)*. Springer-Verlag, Berlin, Heidelberg, 85–95.

[16] Reinhard Diestel. 2024. *Graph theory*. Springer (print edition); Reinhard Diestel (eBooks).

[17] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/7aa685b3b1dc1d6780bf36f7340078c9-Paper.pdf

[18] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[19] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. 576–587.

[20] Yu Feng, Osbert Bastani, Ruben Martins, Isil Dillig, and Saswat Anand. 2017. Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-synthesis-semantic-malware-signatures-using-maximum-satisfiability/

[21] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices* 53, 4 (2018), 420–435.

[22] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[23] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

[24] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

[25] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/1368088.1368132

[26] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1xwKoR9Y7

[27] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o System Card. *arXiv preprint arXiv:2410.21276* (2024).

[28] Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning Programmatic Idioms for Scalable Semantic Parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 5426–5435. https://doi.org/10.18653/v1/D19-1545

[29] Cezary Kaliszyk and Josef Urban. 2015. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation* 69 (2015), 109–128.

[30] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin. 2001. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* 736–743. https://doi.org/10.1109/ICSM.2001.972794

[31] Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[32] Adarsh Kumarappan, Mo Tiwari, Peiyang Song, Robert Joseph George, Chaowei Xiao, and Anima Anandkumar. 2024. LeanAgent: Lifelong Learning for Formal Theorem Proving. arXiv:2410.06209 [cs.LG] https://arxiv.org/abs/2410.06209

[33] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

[34] Xavier Leroy. 2021. Companion Coq development for Xavier Leroy's 2021 lectures on program logics. https://github.com/xavierleroy/cdf-program-logics. GitHub repository.

[35] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[36] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06)*. Association for Computing Machinery, New York, NY, USA, 872–881. https://doi.org/10.1145/1150402.1150522

[37] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. https://doi.org/10.1145/3691620.3695521

[38] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (2015), 619–630.

[39] Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL, Article 28 (jan 2024), 32 pages. https://doi.org/10.1145/3632870

[40] Gordon D Plotkin. 1970. A note on inductive generalization. *Machine intelligence* 5, 1 (1970), 153–163.

[41] Gabriel Poesia and Noah D. Goodman. 2023. Peano: learning formal mathematical reasoning. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 381, 2251 (2023), 20220044. https://doi.org/10.1098/rsta.2022.0044 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2022.0044

[42] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

[43] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic code search via equational reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[44] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.

[45] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. *Generating correctness proofs with neural networks*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3394450.3397466

[46] Ameesh Shah, Eric Zhan, Jennifer J. Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning differentiable programs with admissible neural heuristics. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 415, 13 pages.

[47] Marc Shapiro and Susan Horwitz. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–14.

[48] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. 2024. Towards large language models as copilots for theorem proving in lean. *arXiv preprint arXiv:2404.12534* (2024).

[49] Friedrich Steimann. 2018. Constraint-Based Refactoring. *ACM Trans. Program. Lang. Syst.* 40, 1, Article 2 (Jan. 2018), 40 pages. https://doi.org/10.1145/3156016

[50] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *First Conference on Language Modeling*. https://openreview.net/forum?id=V7HRrxXUhN

[51] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. Ottawa, ON, Canada. https://arxiv.org/abs/2412.14063 To appear.

[52] Laurent Théry, Benjamin Grégoire, Arnaud Spiwack, Evgeny Makarov, and Pierre Letouzey. 2026. Bignums. https://github.com/coq-community/bignums. GitHub repository.

[53] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.* 33, 3, Article 9 (may 2011), 47 pages. https://doi.org/10.1145/1961204.1961205

[54] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3314221.3314588

[55] Daniel Whalen. 2016. Holophrasm: a neural Automated Theorem Prover for higher-order logic. arXiv:1608.02644 [cs.AI] https://arxiv.org/abs/1608.02644

[56] Huajian Xin, Haiming Wang, Chuanyang Zheng, Lin Li, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, et al. 2023. Lego-prover: Neural theorem proving with growing libraries. *arXiv preprint arXiv:2310.00656* (2023).

[57] Yutong Xin. 2025. *TacMiner: Automated Discovery of Tactic Libraries for Interactive Theorem Proving*. https://doi.org/10.5281/zenodo.15761151

[58] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Punta Cana, Dominican Republic, 1691–1704. https://doi.org/10.18653/v1/2021.findings-emnlp.146

[59] Jin Peng Zhou, Yuhuai Wu, Qiyang Li, and Roger Grosse. 2024. REFACTOR: Learning to Extract Theorems from Proofs. *arXiv preprint arXiv:2402.17032* (2024).

[60] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2021. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 931–942. https://doi.org/10.1145/3324884.3416541