ProofCoop: Collaborative Automated Formal Verification

Zhanna Kaufman
University of Massachusetts
Amherst, USA
zhannakaufma@umass.edu

Emily First
Rutgers University
New Jersey, USA
emily.first@rutgers.edu

Alex Sanchez-Stern
d model
San Francisco, USA
alex@dmodel.ai

Kyle Thompson
University of California San Diego
San Diego, USA
r7thompson@ucsd.edu

Sorin Lerner
University of California San Diego
San Diego, USA
lerner@ucsd.edu

Yuriy Brun
University of Massachusetts
Amherst, USA
brun@cs.umass.edu

Abstract

Formal verification using proof assistants, such as Coq or Lean, is an effective way of ensuring software correctness. Recent research has shown that machine learning can automate proof synthesis, but such approaches are only successful a fraction of the time. Methods for improving such proof synthesis include training more precise models, improving the model-driven synthesis search mechanisms, and effectively combining multiple models into synthesis search. This paper focuses on the latter and develops ProofCoop, for the first time demonstrating *collaborative* cooperation between models. Diva, the state-of-the-art method for combining models, uses a disparate search for each model. On the CoqGym benchmark of 68.5K theorems from 124 open-source Coq projects a Diva combination of five models automatically proves 3,287 (27.5%) of CoqGym's 11.9K test set theorems, while the best individual model proves only 2,604 (21.8%). By contrast, our method, ProofCoop can use the same five models to fully automatically prove 3,937 (33.0%) theorems, meaning that ProofCoop is 19.8% more likely to prove a theorem, on average, than the prior state of the art combination method. ProofCoop allows its models to build on each other's contributions and demonstrates that such collaboration significantly increases synthesis success. ProofCoop enables six different types of collaboration: joint model next-step prediction at each search step; preferential next-step prediction via voting, bidding, and stacking; sharing lemmas proven across models mid-search, and models completing each other's partial proofs. Together with CoqHammer, ProofCoop synthesizes proofs for 36.0% of the theorems. Our research demonstrates that creative uses of learned models can lead to collaborative synthesis that is more effective than prior approaches, suggesting a powerful new research direction in automated formal verification.

ACM Reference Format:
Zhanna Kaufman, Emily First, Alex Sanchez-Stern, Kyle Thompson, Sorin Lerner, and Yuriy Brun. 2026. ProofCoop: Collaborative Automated Formal Verification. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3744916.3787783

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3787783

1 Introduction

Poor software quality cost the US economy $2.41 trillion in 2022 [41] and consequences of faults in operational software range from halting global infrastructure services [63] to loss of life [77]. One promising method for improving software quality is formal verification using proof assistants, such as Coq [76], Lean [11], and HOL4 [70]. Formal verification is highly effective: for example, a compiler evaluation study [91] found bugs in every tested C compiler, including LLVM [43] and GCC [73], but none in the formally verified (in Coq) portions of CompCert [47]. There have been several highly successful industrial verification deployments, including at Airbus France [72], Google [16], and Mozilla [34].

Unfortunately, formal verification adoption is hindered by the significant manual effort and expertise required to write specifications and verification proofs. For example, the proofs verifying CompCert are 8 times longer than the functional code [46]. And even small software changes can require heavy proof editing [61].

Recent research has produced tools that use machine learning (ML) trained on human-written or autoformalized proofs to automatically synthesize proofs for new software [49, 57, 66, 67, 75, 89]. These tools (1) use a neural model to predict the next proof step, and (2) use that model to drive a metaheuristic search through the space of possible proofs, using the interactive theorem prover to prune the search and identify successful proofs. Unfortunately, most state-of-the-art tools can verify only a fraction of the desired properties in Coq projects [17, 66, 68, 79].

There are three ways to improve the efficacy of ML-based proof synthesis tools: (1) by training more precise models [18, 67, 79, 89], (2) by improving the search mechanism that uses a model to navigate the space of possible proofs [68], and (3) by effectively combining the use of multiple models in the search [8, 17]. This paper focuses on the latter approach, for the first time using multiple models *collaboratively* to automatically synthesize verification proofs.

The state of the art in combining models is the Diva tool [17], which uses multiple independent searches to attempt to synthesize a proof in parallel using different, diversely trained models. Gpass builds upon this work by identifying other diversity dimensions, such as training time checkpoints, though uses the same method as Diva for combining models at search time [7]. Notably, these state-of-the-art methods do not allow the models to collaborate, using them in independent searches, with each model trying to prove properties on its own. **In this paper, we demonstrate that a collaborative synthesis approach that allows multiple models**

to work together can significantly increase proof synthesis effectiveness, proving 51.2% more properties than the best individual model, and 19.8% more properties than the state-of-the-art combination method without collaboration.

We develop ProofCoop, a proof-synthesis approach that uses model collaboration. ProofCoop explores six approaches that prior proof synthesis tools have not:

(1) *Union tactic predictions*: use the predictions from every model.
(2) *Certainty bidding*: use the most confident models' predictions.
(3) *Voting*: use predictions that win the vote among the models.
(4) *Stack prediction*: use a fine-tuned large language model (LLM) to choose a subset of the model predictions.
(5) *Subgoal proof sharing*: maintain a shared cache among models of proofs of proven facts.
(6) *Local subgoal proofs*: build shallow exploratory subtrees to allow each model an attempt completing proofs of subgoals fast.

In proof synthesis architectures that employ search trees, a tree node is a *proof state* containing the current proof context (including the goal and current hypotheses) and a tree edge is a *tactic command* that transforms a proof state into a new proof state. While most search trees are entirely built using a single model, ProofCoop's approach allows models to collaboratively try to progress the proof at every tree node, potentially reaching search depths that a single model would be unable to reach. This can greatly boost the proof synthesis capability of component models in a collaborative set, particularly when those models perform poorly on their own. Union tactic predictions, stack prediction, and subgoal utilization approaches can synthesize more proofs than the union of those synthesized by all component models. Meanwhile, voting and bidding synthesize more proofs than any individual model with the same efficiency as any individual search.

To train our component models and evaluate ProofCoop, we use the 68.5K theorems from the CoqGym benchmark of 124 open-source Coq projects, which has a test set of 11,911 theorems [89]. On this test set, prior tools for Coq that rely purely on non-collaborative search, Tok [18], Tac [18], ASTactic [89], Passport [67], and Proverbot9001 [66], prove 1,168 (9.6%), 1,188 (9.8%), 1,324 (10.9%), 1,517 (12.5%), and 2,408 (19.8%) theorems, respectively. A Diva combination (without collaboration) of ProofCoop's component models proves 3,287 (27.5%) of theorems in this test set. ProofCoop, meanwhile, proves 3,937 theorems. This is an improvement of 51.2% over the best individual model (3,937 vs. 2,604), and 19.8% more theorems than the state-of-the-art, Diva (3,937 vs. 3,287). ProofCoop's improvement is greater for less individually effective models: Section 4.4 shows that when combining two poorly performing models, ProofCoop can prove up to 30.4% more theorems than Diva. Given the difficulty of manual proof-writing along with the high cost of bugs, this represents significant savings for engineers, as well as a significant increase in automatically verified software quality.

While the ProofCoop approach is independent of model architecture, we implement ProofCoop with models that are variants of Proverbot9001 [66], an RNN-based model. By not using LLM-based models, our evaluation benefits in multiple ways: (1) there is no threat of test leakage as all models are trained from scratch, (2) it is reproducible on CPUs as Proverbot9001 models and ProofCoop's stack predictor only require GPU use at training time, and (3) there is no reliance on API calls to proprietary models.

The main contributions of our work are:

- A conceptualization of the above six investigatory approaches to collaboration of complimentary models during proof search.
- An implementation of each of these collaboration approaches in our tool ProofCoop.
- An empirical evaluation of ProofCoop on the CoqGym benchmark [89], significantly outperforming the state of the art.

We publicly release all the data, code, and scripts used in the research described in this paper to facilitate replication and extension of this research [40].

2 Motivating Collaborative Search

This section introduces theorem proving in proof assistants and how machine learning can automate proof synthesis. It then describes the Diva approach to combining diverse machine learning models for automating proof synthesis, which our approach builds on, and motivates the need for more collaborative search strategies.

2.1 Theorem Proving

To prove a software property, an engineer starts by stating that property as a *theorem* in a proof assistant language, such as Coq. The proof assistant uses this theorem to create a *proof state* describing all known facts (or the *local context* of assumptions) and all facts to be proven (or the *goals* or *subgoals*). An engineer then successively adds commands called *tactics*, evolving the proof state to QED and in the process building up a *proof script*. The proof assistant turns a completed proof script into a *proof object* [62], which is machine-checkable for validity.

2.2 Machine learning for theorem proving

Machine learning tools interact with proof assistants to help guide them through a heuristic tactic search [28]. After predicting a tactic, they use the proof assistant to validate the current proof script and receive a new proof state from the assistant. They use this new proof state, and potentially the current proof script [18], as an input to predict the next tactic. This interaction is combined with algorithms such as tree traversal [66] to allow for an automated search through the space of possible proofs.

Tree-based tactic search in Proverbot9001 [66] uses a search tree in which every node contains a Coq state and every edge is a tactic that changes this state. States include the current goal to be proven and all associated hypotheses and background goals. The root node of the tree contains the initial proof state, including the theorem to be proven as the goal. Predictor models produce tactic options using the context at an edge node of the search tree, ordered by model certainty value. The search procedure involves running each of these tactic options in turn, and adding it on as a child node upon success. This becomes the new edge node. Because the search is depth-first, this new branch is followed until its end — either the depth limit is reached or all model-produced tactics fail. The search then traverses back and uses the next tactic option to start a new branch, up to the maximum width of the search tree. Proverbot9001 uses this approach with RNN-based prediction models.

Models trained with different hyperparameters or training data can result in different sets of valid tactic predictions at the same search step, and therefore very different search trees. This may

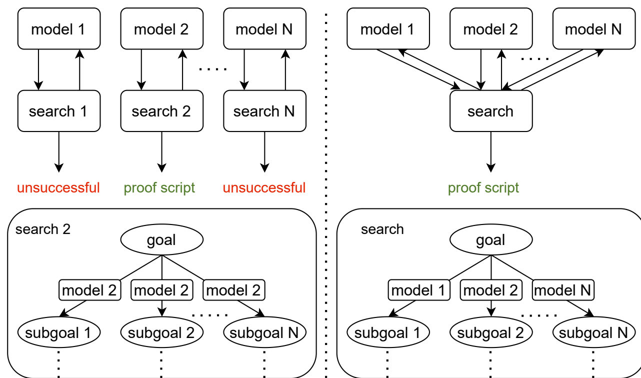

Figure 1: Whole proof level (Diva, left) vs. search step level (right) combination.

mean that two differently trained models are able to synthesize proofs for different subsets of the same set of test theorems.

2.3 The Diva approach

The Diva approach leverages the differences observed among models to combine the proof synthesis capabilities of multiple, *diverse* models [17]. These models can be specialized to synthesize proofs for different types of theorems, increasing overall proving power. Diva performs this combination at the whole proof level, rather than at the level of a single search step (see Figure 1). The models conduct independent searches for a proof. Proof checkers act as oracles to determine whether any individual model succeeded at proving a particular theorem. Accordingly, even if only a single model in an ensemble is able to synthesize a proof in its independent search, the ensemble is considered to have proved the theorem. Diva allows for selecting models that are good at synthesizing proofs of particular theorems. However, when no independent search yield a complete proof, a Diva combination fails to combine models' abilities to predict individual steps or prove particular subgoals, which both are useful capabilities for growing a search tree.

2.4 Illustrating Collaboration

The shortcomings of the Diva approach introduce the need to explore the potential of model collaboration at the search step level as it may allow multiple models to grow a search tree together more effectively than a single model is able to. For instance, model A may fail on a particular search step, while model B may succeed on this step. Meanwhile, model B may fail on a future search step that model A would succeed on. To illustrate this, let us walk through a simple example. Let us suppose that we wish to prove the *power of a power* property for exponents; that is, that raising some natural number x by some exponent n and then raising the result by some exponent m is equivalent to raising x by the product of n and m. This is accomplished in the Coq project `fundamental-arithmetics`. We can formalize this property like so:

```
Lemma power_power: ∀ n m x : ℕ, (x ^ n) ^ m = x ^ (n * m)
```

Let us also assume that we have already proven two supporting lemmas that will make this proof more straightforward.

```
Lemma power_mult: ∀ n x y : ℕ, (x * y) ^ n = x ^ n * y ^ n
```

```
Lemma power_plus: ∀ n m x : ℕ, x ^ (n + m) = x ^ n * x ^ m
```

Finally, let us suppose that we have two models. Model A is very good at induction, and at simplifying goals to make them more tractable, but struggles with using earlier context (such as helper lemmas). Meanwhile, model B is bad at induction and simplification, but is very good at using existing context.

At first, we have only our single goal:

```
1 goal
________________________________________
∀ n m x : ℕ, (x ^ n) ^ m = x ^ (n * m)
```

Model A, effective at induction, begins induction on variable n.

```
1 Proof.
2 induction n.
```

This opens up 2 new goals - the base case and the inductive case. The base case is the focused goal, which model A simplifies via the `simpl.` tactic.

```
2 goals
________________________________________
∀ m : ℕ, (1) ^ m = 1
subgoal 2 is:
∀ m x : ℕ, (x ^ S n) ^ m = x ^ (S n * m)
```

Nothing more can be done here to make the focused goal more tractable, and we proceed with further induction. Model A inducts on the only variable left in this goal, m, which gives us 3 goals to solve. The first goal (`1 ^ 0 = 1`) is trivial, and can therefore be solved with the `trivial.` tactic, while the next goal (`∀ m : ℕ, 1 ^ S m = 1`) can be simplified with automated tactics like `simpl.` and `intuition.` This leaves us with:

```
2 goals
m x : ℕ
H: 1 ^ m = 1
________________________________________
1 ^ m + 0 = 1
subgoal 2 is:
∀ m x : ℕ, (x ^ S n) ^ m = x ^ (S n * m)
```

Now model A gets stuck. Further simplification tactics don't fail but also don't progress the proof. The same is true for continuing to induct on m. We are now at a point where continuing the proof requires the use of context.

Now model B, which is good at using context but poor at induction, is able to correctly intuit that we can replace part of our goal term using the hypothesis, and tries the tactic `rewrite -> H.` This makes our focused goal `1 + 0 = 1`, which is once again trivial. Now we can focus on the final induction step. Model A can fiddle with the goal using tactics like `simpl.` and `intuition.` to make it into a form that can use our existing context:

```
2 goals
n, m, x : ℕ
IHn: ∀ m x : ℕ, (x ^ n) ^ m = x ^ (n * m)
________________________________________
(x * x ^ n) ^ m = x ^ (m + n * m)
```

But only model B is capable of using our previously proven lemma, `power_plus`, to rewrite our foreground goal by replacing `x ^ (m + n * m)` with `x ^ m * x ^ (n * m)`, so the goal becomes `(x * x ^ n) ^ m = x ^ m * x ^ (n * m)`.

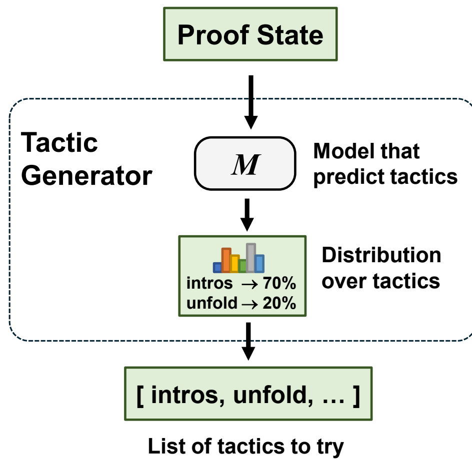

Figure 2: Tactic generator. Given a proof state, it generates a list of tactics to try during proof search.

Now the final term in the goal, `x ^ (n * m)`, matches with the right-hand side of our hypothesis equivalence, which model B is able to use via `rewrite <- IHn` to transform our goal once more:

```
2 goals
n, m, x : ℕ
IHn: ∀ m x : ℕ, (x ^ n) ^ m = x ^ (n * m)
─────────────────────────────────────────
(x * x ^ n) ^ m = x ^ m * (x ^ n) ^ m
```

Finally, this goal is in a form which matches our second previous lemma, `power_mult`, and model B is able to `apply` this lemma to produce:

```
x ^ m * (x ^ n) ^ m = x ^ m * (x ^ n) ^ m
```

All terms on each side of the = are directly equivalent, so the final goal is dispatched and the proof is complete. Neither model A nor model B are able to individually synthesize a complete proof for this lemma. However, when given the opportunity to collaborate at the proof step level, these two models can get to `Qed`.

3 The ProofCoop Approach

While prior work enabled models to collaborate at the theorem level [7, 17], our ProofCoop approach enables proof-search-step-level collaboration. ProofCoop uses six collaboration methods for proof synthesis: union tactic prediction, stack prediction, certainty bidding, voting, subgoal proof sharing, and local subgoal proof exploration. Union tactic prediction, stack prediction, certainly bidding, and voting allow models to share tactic predictions, while subgoal proof sharing and local subgoal proof exploration allow models to share successful subproofs. We next describe the six methods.

3.1 Tactic Prediction Communication

Recall from Section 2.2 that model-based proof-synthesis tools use a learned model that predicts, given the current proof state and the already partially written proof, the set of next proof steps to explore. This is shown by Figure 2 — at each node in the search tree, a *tactic generator*, shown in a dashed box, takes a proof state, and generates a list of tactics to try. Each proof step consists of a tactic that acts to change the proof goal or the hypotheses. Each tactic prediction comes with a confidence score, based on the model's certainty that this is the correct tactic, computed from model-generated probability distributions of the predicted tactics. A proof search can then use a given model repeatedly to predict tactics, trying the tactics in order of decreasing confidence (in other words, trying the top N-predictions). This generates a search tree where each node is a proof state and children nodes are generated by applying tactics predicted by the model. The search tree is then traversed using various search strategies. Because the search tree is usually much larger than can be explored in full, the order in which tactics are tried is crucial to the success of the proof search. In a regular search using a single model M, this tactic generator uses the model M to predict a probability distribution, from which a list of tactics is generated.

Figure 3(a) illustrates ProofCoop's communication through tactic predictions, where the dashed boxed is like the tactic generator from Figure 2 but modified to make use of multiple models. Say we have k different models, and each one takes as input a proof state, and returns a probability distribution over likely tactics. At each node in the search tree, ProofCoop allows the models to share by asking each of the k models to provide its predicted tactic list and confidence distribution, and then using a *merge* function to combine the k distributions into a single ordered list of tactics to try. We next describe three implementations of *merge* for sharing strategies between models.

3.1.1 Union Tactic Prediction. ProofCoop's first approach to model collaboration works as follows: at each node in the search tree, take the union of the tactics predicted by each model, and then try the tactics from this union. This is implemented by having the *merge* function from Figure 3(a) return the union of the tactics suggested by all of the models. Union Tactic Prediction combines tactics in a round-robin fashion, trying every model's highest confidence tactic before it tries their second highest confidence tactics. This results in one single-ordered list of tactics. When iterating through this list, ProofCoop skips over all tactics proposed by a particular model once the search width limit for that model has been reached. In this way, when ProofCoop uses this approach, tactics from all models are tried at each node in the search tree. Theoretically, this approach subsumes any of the individual models doing their own search: a proof found by any one of the models in its own search will be made up of tactics that the combined approach will also try. Since the approach tries combinations of tactics that no single model would have tried in one single search tree, the combined search tree contains proof attempts that were previously unexplored, thus having the potential to find proofs that no single model could have. While this approach does find previously unexplored parts of the search space, it also leads to a large (specifically wide) search tree.

While aggregating tactic predictions in this way can potentially attempt every possible search combination, the time and space resources required by this approach can quickly multiply beyond feasibility. For instance, a tree with a maximum depth of 10 and a search width of 5 per model (our default), with one model, will contain a maximum of $1.22 * 10^7$ nodes when only 1 model is used, and $9.93 * 10^{13}$ nodes when 5 models are used, an increase of 8 million times. This can greatly increase both search time, as each model must be queried for its predictions at every node, as well as required

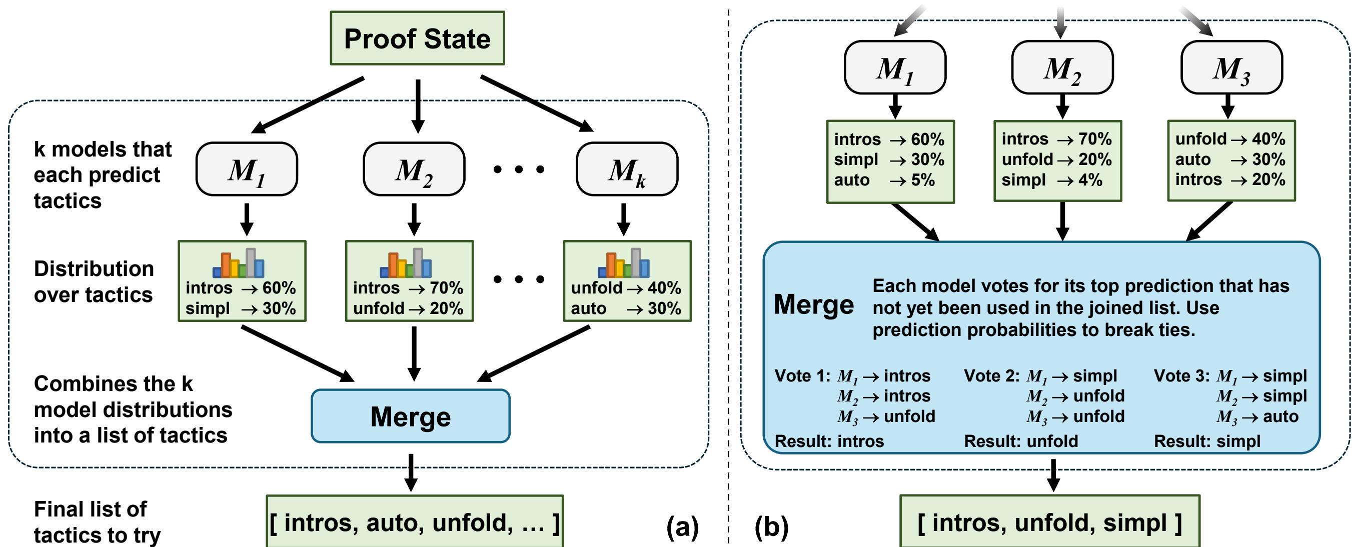

Figure 3: Sharing with tactic predictions as implemented in ProofCoop: (a) A general case of tactic prediction sharing with a *merge* function. (b) A voting example.

space. Depending on available resources, this can make larger model combinations infeasible. To allow this approach to run in more constrained resource setups, we attempt to limit the width of the tree by reducing the number of children created from each node. We first create a list of tactics that have already been run by the search from the current node, and disallow the search to create multiple child nodes with the same predicted tactic. Furthermore, we consolidate predicted tactics that perform similar functions. We do this by replacing sets of tactics with tactics that fully subsume the functionality of every tactic in the set, so as not to prune any valid search paths. We make the following replacements: {`injection`, `discriminate`} →`congruence`, {`omega`} →`lia`, {`trivial`, `reflexivity`, `symmetry`, `contradiction`, `inversion`} →`easy`, {`lazy`, `cbv`} →`vm_compute`, {`tauto`, `intuition`} →`firstorder`, and {`auto`} →`eauto`.

We also notice that models often predict certain automation tactics, either tactics that try to simplify the current goal in an automated manner, or solve it entirely. These automation tactics are particularly useful in the beginning or the end of the proving process. In the beginning, they can simplify the goal in generic ways that prepare it for more complex modifications. In the end, they can perform the final steps of a proof by trying a variety of simple solutions that would potentially involve a trial and error process for the human user.

With the models taking the place of the human user, we use this insight to force more complex goal processing out of the models earlier in the search by offloading their use of automation tactics. We do this by creating a dummy model that always outputs 6 tactics ordered by computational cost; `easy`, `congruence`, `lia`, `eauto`, `firstorder`, and `vm_compute`. The first 5 of these are solution automation tactics, while the last is an automation tactic for simplification. All 6 tactics are capped to a timeout of 1 second per attempt, as a small-scale exploratory study showed that 1 second was both effective enough to cover most tactics (avg. tactic runtime of 0.04 seconds), while still being reasonably efficient. For all other tactics, we use the default Proverbot9001 timeout. Just as every other model in our architecture, we use up to the first 5 successful tactics from this dummy model. However, unlike other models, no additional request/response time is required. These tactics are also tried first, so that our actual models cannot try using them again. This means models are more likely to utilize their specific knowledge of the goal and the proof state, by applying tactics such as the unfolding of specific terms or the application of specific hypotheses. Additionally, if a common automation tactic that is likely to eventually be predicted by our models can make significant progress or even prove the theorem entirely, this is accomplished in the first several branches of a node's subtree. With both of these adaptations, we aim to reduce the width and depth of a search, reducing the overall tree size and therefore the time and space needed for the search, thus improving the feasibility of this approach.

3.1.2 Certainty Bidding. To narrow down the search space, or equivalently prioritize the parts of the search tree to explore, certainty bidding makes use of the confidence score. This approach tries the tactics with the highest (un-normalized) individual prediction scores across all model predictions, regardless of which model predicted them. The intuition is that, at each node in the search tree, the approach uses tactics from whichever model is the most confident in its prediction. This is implemented by having the *merge* function order all the tactics by their confidence score.

3.1.3 Voting. The intuition for iterative voting is that if multiple models predict the same tactic, it is more likely to be a good tactic to try. The simplest way to implement voting is to count the number of models that predicted a tactic, but this does not take the ordering of predictions into account. Instead, ProofCoop uses an approach where models vote to determine the topmost candidate, then vote to determine the second topmost candidate, etc. This is done by

having the *merge* function do multiple rounds of voting. At every round, each model puts forward (as its vote) its predicted tactic with the highest confidence score, skipping over tactics that have already been added to the final result (in prior rounds of voting). In this way, for example, a model will keep voting for its topmost candidate until this tactic is successfully voted upon by enough other models (thus being added to the final result), and then the model will move on to its second topmost candidate. Ties in the voting are broken by the confidence score of the tied tactics.

Figure 3(b) shows how this *merge* function would work on an example with three models. There are three rounds of voting:

- Round 1: M_1 and M_2 vote for `intros`, and thus `intros` wins the first spot in the final list.
- Round 2: M_1 and M_2 cannot vote for `intros` again, since `intros` is already in the final list, so they pick their next highest rated tactic, `simpl` and `unfold` respectively; M_3 votes for `unfold` (as it also did in the first round), and thus `unfold` wins the second spot in the final list.
- Round 3: M_1 votes for `simpl` again; M_2 cannot vote for either `intros` or `unfold` so votes for its third highest tactic, `simpl`; M_3 cannot vote for `unfold`, so it votes for its second highest tactic, `auto`; and thus `simpl` wins the third spot in the list. Assuming that we are only doing a search with a fan-out of 3, the final list is then `[intros, unfold, simpl]`. If we wanted more elements, we could have continued the process.

The voting approach provides a distinct tradeoff compared to the certainty bidding approach: it can prevent a single confident model from dominating the search tree, but it can also lead to trying tactics that many models voted with low confidence.

3.1.4 Stacking. Both voting and certainty bidding can potentially be negatively impacted by a model that is very certain about the wrong tactic. In the certainty bidding approach, this model can win out among better trained models that predict multiple effective tactics with less certainty. In the voting approach, it can bolster a vote for an incorrect tactic, meaning that this approach is most effective when the majority of models are relatively accurate. A stack predictor approach instead involves training a meta-model that predicts which tactics among all outputs are most likely to be correct at each search tree node. The stack predictor acts as a controller for the component model set, allowing models to be trained for specific tasks (for instance, solving subgoals of certain types) and training the predictor to recognize which model is best suited for the current task. To do this effectively, this predictor must be able to recognize when newly observed goals and proof contexts are semantically similar to those seen in the past. LLMs excel at this task [30], and so we implement ProofCoop's stack predictor using the ModernBERT LLM (22 layers, 149 million parameters) [84]. We use GPUs for training and CPUs for inference. To gather training data, ProofCoop tries all the models on various proof states, and for each proof state records the goal, model, tactic and certainty of the prediction, along with whether the prediction matched the ground truth. ProofCoop uses this data to fine-tune an encoder-based LLM meta-model as a classifier that will return, given this combination of inputs, a certainty that the prediction matches the ground truth. During search, ProofCoop prompts every model for its list of predicted tactics along with their certainties, and inputs each prediction into the stack predictor. The resulting list of certainties outputted by this predictor are sorted in order of positive classification certainty, and the top 10 are returned in order as the collaborative output.

3.2 Communication via Shared Subgoal Proofs

The first four approaches to model collaboration we described worked by having models share tactic predictions. The next two approaches use a different communication mechanism between models: instead of sharing tactic predictions, these approaches share completed proofs of subgoals.

3.2.1 Subgoal Proof Sharing. A subgoal is a proof objective generated at an internal node in the search tree, containing the current goal to be proven and associated hypotheses. Say that model A unsuccessfully tries to prove a theorem, but during its search it generates a subgoal that it *successfully* proves. ProofCoop will store this subgoal proof in a global cache, along with the associated subgoal and its hypotheses. When another model, say B, encounters the same subgoal, it can use the shared proof (from the cache) to complete the subgoal, rather than trying to prove it on its own. This increases the power of model B, which would not have been able to prove this subgoal on its own. Enabling B to prove this subgoal allows B to explore other parts of the search space.

Figure 4(a) illustrates our subgoal sharing approach. Completed proofs of subgoals are collected by instrumenting the proof search to identify when a new subgoal starts and when it is completed, collecting the steps in between to create the proof. At the top level, ProofCoop then runs two sets of full searches, with every search using a single model at a time, in a round-robin order. During the first set of searches, successful subgoal proofs are only collected. During the second set, every search has access to and can reuse all the successful subgoal proofs in the cache from the previous model runs. Furthermore, during the second set of searches, successful subgoal proofs continue to be collected, so later models may have access to a larger cache than earlier models.

3.2.2 Local Subgoal Proofs. Our final approach also involves shared subgoal proofs, in a less robust but more efficient manner that does not require rerunning individual searches with multiple models. In this approach, at each node in the search tree where a new subgoal is opened, ProofCoop asks each model to perform a *local* search of limited scope to try proving the current subgoal. ProofCoop limits the scope by setting a maximum depth for each local search, say for example 3. If any of the models can discharge the subgoal within its local search, then the subgoal proof is considered complete, and the search can move on to the next subgoal. If none of the models can discharge the subgoal within their local search, then a default model is used to predict a set of tactics, with the search going on. This approach allows models to work together in a local way to complete subgoals, but does not require the global coordination of the cache-based approach. Figure 4(b) shows how this approach works. The dashed box in Figure 4(b) represents the tactic generator that replaces the dashed box from Figure 2. Note that M_1 through M_k are the models that perform local searches and model M is the default model that the approach backs onto if no local searches discharge the subgoal.

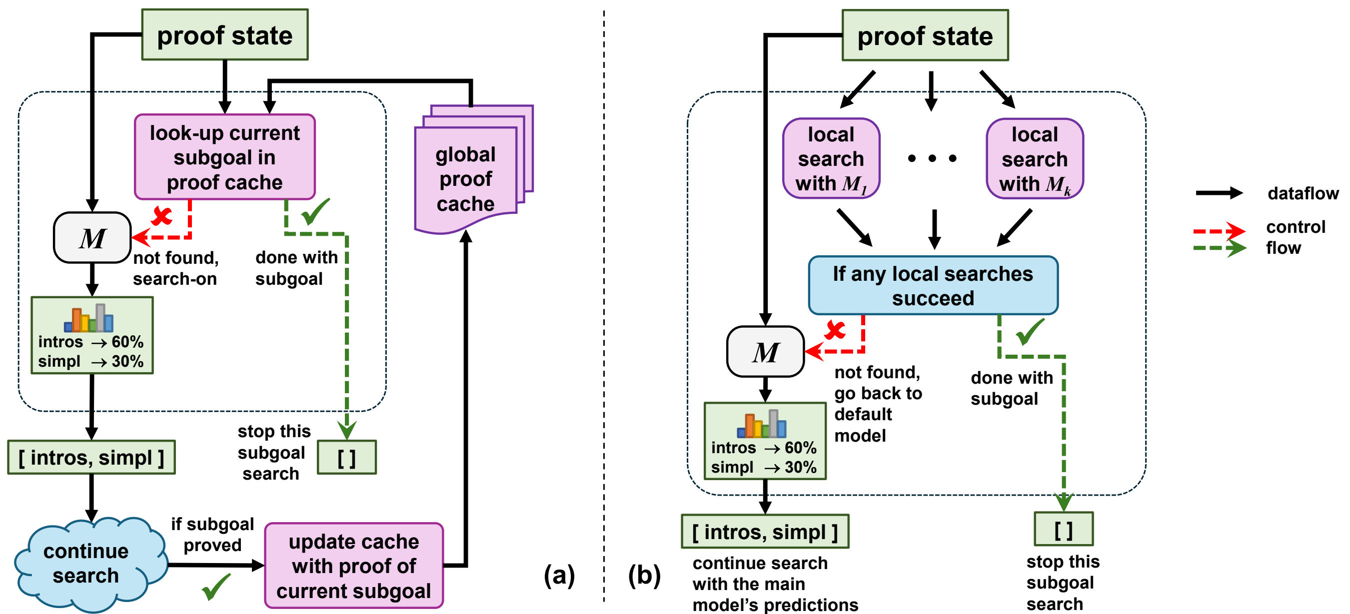

Figure 4: Sharing subgoals: (a) ProofCoop's subgoal sharing approach and (b) ProofCoop's local subgoal proofs approach.

3.2.3 Robustness. Potential issues encountered when running any tree-based search with Coq include stack overflow and timeout errors. These errors can happen when a tactic attempt doesn't effectively terminate due to excessive recursion or looping, or takes up too much memory opening subgoals or attempting paths. When a single model search runs into this error, the entire search fails and no proof script can be synthesized. Unfortunately, it is possible that if such a failing path is explored before others in the above approach, the search will fail even though a different path would have succeeded. This is particularly an issue for approaches where we do not filter our list of model tactics (union tactic prediction, subgoal sharing, and cheap subgoal exploration). To combat this, for these approaches, ProofCoop keeps a cache of 3-tactic sub-paths for each proof that caused failures, and prevents the search from reattempting sub-paths in this cache. ProofCoop also keeps a cache of attempted paths that did not succeed, but also did not cause a crash. As a new search moves through its tree, every path is checked against both of these caches, allowing ProofCoop to continue a proof at exactly the crash point. This approach requires restarting a proof attempt after crash. Unfortunately, many paths that cause crashes are very similar, requiring several restarts. ProofCoop reruns search attempts up to 5 times the number of component models. It is still possible that the number of failing paths is greater than the number of attempts, but this method reduces the chances that a search fails because of an error caused by a single model.

4 Evaluation

We evaluate ProofCoop's ability to synthesize proofs and answer the following research questions:

RQ1 How effective is ProofCoop at proving theorems compared to the state of the art?

RQ2 How effective and efficient are each of ProofCoop's underlying approaches?

RQ3 How does the number of ProofCoop's component models affect its theorem proving capability?

RQ4 What is the relative impact of each of ProofCoop's collaboration strategies?

Our main findings are:

- ProofCoop's union tactic prediction approach proves 19.8% more theorems than a Diva combination of 5 models and provides an upper bound to the capabilities of model collaboration.
- ProofCoop's more efficient subgoal sharing approaches still prove more theorems than a Diva combination.
- ProofCoop's most efficient approaches, voting and certainty bidding, prove more theorems than their component models within the same search time, while its stack predictor approach proves more theorems than a Diva combination within the same search space but requires additional processing overhead.
- ProofCoop is most beneficial in cases where component models themselves are not particularly accurate.

Our data and evaluation scripts are publicly available as part of our replication package [40].

4.1 Benchmarks

We train and evaluate on the CoqGym benchmark [89] of 124 open-source Coq projects with 68,501 theorems. This benchmark contains a compiler, proofs and implementations for mathematical and automation concepts, programs, and automation libraries. These projects range in size from over 100K lines of code (CompCert [47]) to just over 1K (zsearch-trees). Some of these projects do not contain any proofs but are included for completeness in line with prior work [66]. Also in line with prior work, we use 97 projects (57,719 theorems) as training data for our component models, and the other

26 projects (11,911 theorems) for evaluation. Our 5 main component models are trained on the largest 5 training projects (29,685 theorems). To accommodate resource constraints, for some of the research questions, as detailed below, we use a uniformly random representative subset of 500 theorems, which we refer to as 500-test.

4.2 Experimental Setup

We implemented our collaboration strategies on top of Proverbot9001's search tree architecture [66]. We first performed a limited exploration of diversity metrics influenced by the work done in Diva [17]. To find model combinations that exhibit diversity, we trained and evaluated models with different training settings on the CompCert [47] project from the benchmark set. We explored different training data sets, learning rates (0.7, 0.35, 0.07, 0.007, 0.0007), different optimizers (Adam, RMSProp, SGD), and used the Proverbot9001 architecture to modify its RNN model structure by turning off certain input weights (goal heads, hypothesis heads, hypothesis scores, previous tactics).

We found that using different training sets was a particularly effective diversity metric, and that this metric had the added benefit of providing us with some models that were as effective as state of the art tools and some models that performed poorly. This allows us to explore how the benefits of ProofCoop in cases where its component models are fine-tuned to high accuracy, and cases where a model's accuracy is average or low.

Consequently, we trained Proverbot9001 separately on each of the 5 largest CoqGym projects (CompCert, GeoCoq, VST, Coq, distributed-reference-counting). These projects contain 120,576, 74,085, 96,640, 72,387, and 15,950 training data points, respectively. We trained on a cluster including NVIDIA Tesla A100s, NVIDIA Tesla V100s, NVIDIA A40s and NVIDIA Ada Lovelace L40S GPU nodes. We primarily used default Proverbot9001 parameters, including tactic timeout, maximum search tree width, and maximum search tree depth. We only varied learning rate (0.05 for all models aside from CompCert, which trained best on the default of 0.7), the context filter (where we filtered for goal arguments, hypothesis arguments, and relevant lemma arguments), and tactic timeout in the case of dummy tactics (as discussed in Section 3.1.1). The Diva combination of the proofs synthesized by these models proved 3,287 theorems, or 27.6% of the test set.

RQ1 (Section 4.3) evaluates union tactic prediction, ProofCoop's most comprehensive traversal of the search space (recall Section 3.1) on the entire CoqGym test set, and compared it to the Diva combination of the above 5 models. RQ2 (Section 4.4) compares these results to ProofCoop's alternative approaches in terms of the number of theorems proven in our full and 500-test sets, and the average number of search steps per theorem proven in the 500-test set. RQ3 and RQ4 also use the 500-test set (Sections 4.5 and 4.6). RQ3 compares Diva combinations of 2, 3, and 4 models to ProofCoop's combinations of those same models for the union tactic approach. RQ4 performs ablation studies to test algorithmic and Diva combinations of ProofCoop's approaches.

4.3 RQ1: Comparison to State of the Art

While ProofCoop's approach can be implemented using any search-tree-based proof synthesis architecture, we evaluate using and against component models trained on Proverbot9001 (recall Section 4.2). We find that ProofCoop's union tactic prediction approach synthesizes successful proofs for 33.0% of theorems in the CoqGym benchmark, compared to 27.5% proven by the Diva combination of the 5 component models. The best individual model can prove 2,604 (21.8%) of theorems. In combination with CoqHammer, ProofCoop's union tactic prediction approach synthesizes successful proofs for 36.0% of theorems.

While, in theory, ProofCoop should subsume all Diva combination results, due to the feasibility design decisions discussed in Section 3, ProofCoop does not in fact prove all of the theorems in that combination. Instead, Diva combinations and ProofCoop can be complimentary. This is especially important since simply adding models to Diva is not always an effective way to increase the number of theorems proven. For instance, we more than doubled the number of models in our Diva combination to 11 by including 6 of the most successful models from our set discussed in Section 4.7 (models trained on the entirety of the CoqGym training set with goal head weights, hypothesis head weights, and previous tactic weights nullified, as well as models trained on the entirety of the CoqGym training set with maximum term lengths of 15, 30, and 45). On average, these models proved 2,411 theorems. The result of the combination of 11 models was a total of 3,586 theorems proven, or 30.1% of the test set. However, a Diva combination of the original 5 models along with ProofCoop proved 4,091, or 34.3% of the test set. This indicates that training fewer models and using them collaboratively is more effective than training more models.

approach	theorems proven	average steps
best individual model	2,604	231.4
Diva combination	3,287	306.0
Diva & automated tactics	3,327	268.0
union tactic prediction	**3,937**	709.0
subgoal proof sharing	3,328	503.5
local subgoal proofs	3,451	265.2
certainty bidding	2,627	201.6
voting	2,633	184.0
stack predictor	2,670	229.5

Figure 5: The number of theorems proven in the full CoqGym test set, along with average number of search steps per synthesized proof in the 500-test set.

4.4 RQ2: Comparisons of individual strategies

Figure 5 shows the number of theorems in the CoqGym test set proven by the most successful individual model, a Diva combination, and ProofCoop's union tactic prediction, subgoal proof sharing, local subgoal proofs, and certainty bidding approaches. It further shows the average number of search steps per proven theorem, computed over the 500-test set.

We use average number of search steps as a proxy for efficiency. Runtime is dominated by theorem prover calls, which happen once per search step [66]. Cluster load, node assignments, and parallel search executions make wall-clock runtime an unreliable measure. Search steps is the typical efficiency measure in this domain [68].

# of models	individual model	Diva combo	ProofCoop	cheap exploration	subgoal sharing	stack predictor	voting	bidding
2	74.2	98.8	119.2	103.0	99.8	85.7	87.2	90.1
3	74.2	110.2	127.9	115.7	110.5	90.7	94.9	95.0
4	74.2	116.8	140.8	121.2	116.6	95.8	95.2	95.4
5	74.2	121.0	150.0	126.0	121.0	97.0	96.0	96.0

Figure 6: The number of theorems proven from a 500 theorem test set comparing the following: For $X \in \{2, 3, 4, 5\}$, The average of theorems proven by each individual model component model, and then for every combination of X models in the set of 5 component models, the average number of theorems proven by: all Diva combinations, ProofCoop, the cheap exploration approach, the subgoal sharing approach, the stack predictor approach, the voting approach, and the bidding approach.

The results show that union tactic prediction, subgoal proof sharing, and local subgoal proofs prove more theorems than Diva. While the union tactic prediction strategy synthesizes the most proofs, it is also the least efficient, requiring on average 2.3 times more search steps than Diva. This inefficiency also limits scalability. All of the search steps for this strategy are taken within the same search tree (unlike the subgoal proof sharing and local subgoal proofs strategies, which create separate search trees for each component model). Thus, it is not possible to easily parallelize or serialize any parts of this search to fit time or computational constraints, limiting the number of component models. Meanwhile, other strategies are far more efficient, with search step requirements similar to or below an individual model. The local subgoal proofs approach uses on average 13.3% fewer search steps than Diva while also proving 5.0% more theorems. Subgoal proof sharing is 1.2% more effective than Diva. For subgoal sharing to be effective, it must be the case that some model can prove a subgoal that another model gets stuck on, but not the preceding subgoal. This sets a strict requirement for the way in which models must complement one another for this collaborative approach to be successful. It may also be that two searches per model, one to gather proven subgoals and one with access to them, is not enough, and using more search rounds to allow for more depth is an interesting direction for future work. Alternatively, subgoal sharing may be more useful in combination with other approaches (our study in Section 4.6 shows that it can help improve the effectiveness of union tactic prediction).

The other collaboration strategies did not prove more theorems than Diva. However, voting, bidding, and stack prediction can still be effective. All three of these strategies outperformed the best-performing individual model, which synthesized proofs for 2,604 theorems. Stack prediction synthesized 2,670 (22.4%) theorems, an increase of 2.5%. Naïve voting and bidding performed more modestly, successfully synthesizing proofs for 2,633 and 2,627 theorems, respectively. However, both strategies used fewer search steps than the most successful individual model search, with reductions of 20.5% and 12.9%, respectively. Furthermore, unlike the other approaches, both voting and certainty bidding can be negatively affected by the worst models in the combined set, as bad models can be very certain about poor tactic predictions. Reducing the set of component models to the three best performing ones results in an increase in proven theorems for both of these approaches, with voting and bidding proving 2,679 and 2,676 theorems, respectively, an increase of 2.8%–2.9% over the best performing individual model.

Finally, we compare ProofCoop to a Diva combination that includes a model that only uses ProofCoop's automated tactics. Altogether, this combination proves 3,327 of theorems in the test set, making ProofCoop 18.3% more effective.

4.5 RQ3: Comparisons of model counts

Figure 6 presents the number of theorems (from the small test set) Diva and ProofCoop combinations of 2, 3, 4, and 5 models prove.

For certain approaches, as the number of component models increases, ProofCoop's improvements over Diva decrease. This happens for the same reason that the benefit of Diva combination itself decreases — reaching the ceiling of the ability of such models to predict usable tactics. Interestingly, the increase in theorem's proven by ProofCoop over the Diva combination was most effective for models that performed more poorly individually. The models trained on GeoCoq and distributed-reference-counting proved 53 and 42 theorems out of 500, respectively, and in Diva combination proved only 69, or 13.8% of the theorems. However, when combined through ProofCoop, they proved 90 theorems, an improvement of over 30%. Additionally, the collaboration of these models was as successful as the top performing individual models, only less successful than the most accurate model, trained on the VST project, which proved 97 theorems. Meanwhile, the improvement seen when component models performed well already was far smaller. Models trained on the VST and Coq projects proved 97 and 89 theorems out of 500, respectively, and in Diva combination proved 111. When used as components of ProofCoop, the tool proves 126 theorems, an improvement of only 13%.

For a more resource-equivalent comparison against depth-first search (DFS) for each component model, it would be ideal to include average values for each component model given search tree widths of 5, 10, 15, 20, and 25. This would be the equivalent of the ProofCoop search widths. However, as discussed in Section 3.2.3, larger widths of 20 and 25 result in an infeasible runtime (>1 week) for DFS searches with our available resources. We did find that, for instance, searches given a width of 10 resulted in 96.6 theorems proven across searches by each of our 5 component models, which is less than the average Diva combination of 2 models. This indicates that the Diva combination is likely an effective benchmark against which to compare in spite of potential resource inequality.

4.6 RQ4: Ablation Study

We seek to better understand the relative impact of ProofCoop's approaches. Measuring this requires looking at our approaches in combination. However, not all combinations in which strategies run together are possible because some approaches are orthogonal to each

algorithmic combination	theorems proven
subgoal sharing & voting	123
subgoal sharing & bidding	122
subgoal sharing & stack prediction	122
subgoal sharing & union tactic prediction	187
local subgoal proofs & voting	128
local subgoal proofs & bidding	130
local subgoal proofs & stack prediction	129
local subgoal proofs & union tactic prediction	141

Figure 7: Algorithmic combinations of ProofCoop's approaches and # of theorems proven on 500-test set.

Diva combination	theorems proven
all 6 approaches	4,091
all without voting	4,091
all without bidding	4,090
all without stack prediction	4,088
all without subgoal sharing	4,079
all without local subgoal proofs	4,066
all without union tactic prediction	3,556

Figure 8: Number of theorems from the full test set ProofCoop's approaches prove in a Diva combination when we remove each of them individually.

other; voting, bidding, and stack prediction cannot all be performed together because they are different methods of selecting the best single tactic at a particular search step. Subgoal sharing and local subgoal proofs cannot be combined because they are different methods of pooling subgoal solutions. As a result, we measure the relative impact in two ways: one where we combine pairs of approaches within the tool algorithmically such that they have access to each other's functionality, and one where we analyze the effect of removing each of ProofCoop's approaches from their Diva combination.

For algorithmic combinations of approaches, we combine subgoal sharing and local subgoal proofs separately with each of voting, bidding, stack prediction, and union tactic prediction (see Figure 7). We use the 500-test set of theorems and measure the number of theorems proved by each combination. For Diva combinations of approaches, we combine all 6 ProofCoop approaches on the full set of 11,911 theorems, and show the effect of removing every approach individually from this combination (see Figure 8).

For both of these experiments, our results show that the most impactful approach is union tactic prediction. Local subgoal proofs and subgoal sharing contribute a modest impact, while the preferential prediction methods contribute no visible impact. This is not surprising, as the union tactic prediction and subgoal methods are able to traverse much larger search trees that likely subsume the paths taken by preferential prediction methods. The benefits for less effective approaches exist instead in their efficiency, as discussed in Section 4.4. Indeed, algorithmic combinations provide evidence for the limitations that computation places on the effectiveness of union tactic prediction. For instance, theoretically, union tactic prediction should subsume all search paths that subgoal sharing would contribute. However, integrating subgoal sharing with union tactic prediction likely allows ProofCoop to reduce the size of the search tree union tactic prediction explores, increasing its efficiency and allowing the algorithmic combination to prove 24.7% more theorems with the same computational resources. Meanwhile, local subgoal exploration may increase the size of the search tree that union tactic prediction explores by creating sub-trees at every newly opened subgoal, thus increasing the potential for Coq proof assistant errors, and decreasing the number of theorems proven by 6.0%.

4.7 Discussion: Exploration of Diversity Metrics

This section examines different diversity metrics for Diva model combinations. From our results in Sections 4.5 and 4.4, we see that model combinations that are more effective in Diva combination also result in a more effective ProofCoop combination. While we chose to combine models trained on the five largest CoqGym projects, we also explored other approaches, including those that Diva [17] showed to be most effective. The Diva authors make the claim that learning rate and number of previous tactics input into the ML model were the most effective methods of creating diversity.

However, methods for creating diversity may be architecture dependent. For instance, we found through a small-scale experiment that we were not able to successfully train models on the full CoqGym training set when we used learning rates that differed by a factor of 10 from 0.05, while small learning rate changes did not result in diversity. Ultimately, this prevented us from creating diversity through learning rate. Furthermore, the Proverbot9001 architecture limits the number of previous tactics input into a model to 1, and changing this value would involve changing the architecture itself. Therefore, we investigated alternative approaches to diversity.

We focused on training data, length of the input term (standardized number of tokens input into the tactic predictor), and model architecture settings in the form of nullifying certain features (goal head weights, hypothesis head weights, and previous tactic weights). We also investigated whether a combination of diversity metrics may result in a more successful search, and trained several models that modified each of these variables at once.

In Figure 9, we see that the largest increase over the best individual model comes from varying term length and weight nullification, with an increase of 13.3% and 13.4%, respectively. While our collection of trained models is not exhaustive, this exploration shows the potential benefits of varying model settings to diversify model capabilities and potentially maximize combined model utility. Though we found that varying term length and weight nullification resulted in the effective model diversity, we also found varying training data to be effective, and motivate further experimentation for a more systemic exploration of model diversity metrics.

4.8 Threats to Validity

While we explored several approaches to searching for complimentary models to aggregate, our search was not exhaustive, and a more rigorous exploration of potential factors in complimentary models should be performed as future work. This may involve not only searching for models that synthesize the highest number of proofs in aggregate, but also models that solve high numbers of aggregate

metric	mean	best	Diva combination
term length	2,427.3	2,542	2,879
null weights	2,431.0	2,461	2,792
train data	1,746.3	2,494	2,756
combination	1,264.0	2,233	2,287

Figure 9: The diversity metric, the number of theorems proven by the best individual model, and the number of theorems proven by a Diva combination.

subgoals, as the main collaboration between models happens at the subgoal level. Factors that influence the ability of two models to compliment each other may include differences in hyperparameters, including different optimizers, learning rates, and training epochs, as well as training data (training some models to be better suited to proofs contained in particular projects) and potentially even different model architectures.

Our list of collaboration strategies is not exhaustive. The goal of this work is to use insights from prior work on ML-driven tree-based proof search to develop collaboration methods, and understand the trade-offs between efficacy and efficiency. There exist both other methods of ML-driven proof search and other methods of tree-based model collaboration. For instance, other algorithmic strategies exist to heuristically determine the best model or prediction to use at a given search step. One such method (multiplicative weights update) maintains weights for every component predictor that are iteratively updated based on performance [2]. Our results motivate the need for continued research of collaboration strategies.

Transformer-based models, such as LLMs, can prove more theorems than RNN-based models [79]. Our approach is just as applicable to LLM-based tactic prediction models, and investigating LLM collaboration is an important future work direction. However, testing with LLMs, especially on an established test set like CoqGym, provides the potential for test leakage. Using RNN-based models allows us to directly compare to existing state-of-the-art tool, such as Diva and Proverbot9001, without potential test leakage compromising evaluation integrity.

We found that increasing search width beyond 15 for a standard DFS search increased required search time enough to limit feasibility even on a test set as small as 500 theorems. Thus, this prevented us from performing a full comparison against searches that are given comparable resources (in terms of search width).

5 Related Work

Our work is inspired by SMT solvers, which, in industrial settings, are deployed in a portfolio in a winner take all strategy [64, 87]. The portfolios can include different solvers, e.g., Z3 [10] and CVC4 [4], and differently randomly seeded solvers. The solvers may partition an SMT instance to avoid redundant work, with each partition solved by its solver [50]. The solvers may *share solved clauses* [26, 50, 60], which inspired our subgoal proof sharing.

Recent work has explored using machine learning to synthesize proofs, where a predictive model guides a search through the space of candidate proofs. Model architectures for this task include RNNs [32], LSTMs [89], GNNs [57, 65], and transformer-based LLMs [27, 36, 37, 59]. While these models typically predict the next tactic (with arguments), Proverbot9001 [66] uses two separate models, one for tactic prediction and one for argument prediction.

Reinforcement learning (RL) has also been effective for proof synthesis; this requires creating an action space, either by providing a limited static tactic list [88], using an ML model to generate the action space as a ranked tactic list [3, 68], or using an LLM's token space as an action space [12]. Expert iteration and self-play can be useful for training LLM-based proof synthesis models [13, 48], learning from machine-generated proofs. Augmenting training data for component ProofCoop models with proofs generated by other models is a promising future work direction. However, more training data is not always beneficial, as filtering strategies are required to exclude examples during RL training [7, 48]. Our work supports the need for exploring training data filtering as a model trained on all CoqGym training projects does not subsume models trained on individual projects, and when the individual projects' models collaborate, they perform even better. RL models can also be used to learn a fitness function that predicts promising proof search paths [68]. Future work should explore the incorporation of such a function into model collaboration. This may increase efficiency by pruning unpromising search tree nodes and enable stage-based collaboration, as different collaboration methods may be optimal as the proof search progresses.

Prior work explores models that consider different features of the proving environment. LeanDojo [90], Magnushammer [52], and Graph2Tac [65] retrieve already proven premises from the environment to serve as input to their tactic prediction models. TacTok [18] uses the proof script, Diva [17] additionally uses the proof term, and Passport [67] includes rich identifier information. Baldur [19] takes error messages as input to their whole proof prediction model. Gpass considers training checkpoints to use fewer models in a Diva combination [7]. All of these features serve as diversity dimensions or metrics, but for this work, we only considered a limited set of these to make our experiments tractable.

Neural models can perform proving tasks apart from next tactic prediction. Thor [36] fine-tunes an LLM to learn when to apply Sledgehammer [58] or when to predict a tactic. DraftSketchProve [38] prompts an LLM to translate informal proofs into a formal proof sketch with holes, filling them in with calls to Sledgehammer, thereby *autoformalizing* the natural language proof. Follow-up work extends this framework [83, 93, 94]. LEGO [83] decomposes theorems into helper lemmas by prompting an LLM. Future work should explore the collaboration of these models.

Search procedures are typically a depth-first, breadth-first, or best-first search [66, 89, 90]. Some work uses more complex search, such as Monte-Carlo tree search [42] or A* search [21]. However, even in an approach like Diva that considers multiple models, prior work in proof synthesis has always conducted the search with each model separately. Model combinations have been used to solve geometry problems via AlphaGeometry2 [9], by recording proven facts in a shared database. Cobblestone [39] employs a divide-and-conquer approach to decompose and recursively repair an incorrect whole-proof attempt in collaboration with oracles.

To provide feedback to models during search, we follow Proverbot9001 in using Coq [76], used widely in formal code verification work [24, 25, 33, 35, 47, 53, 62, 69, 86]. However, Coq is one of a number of existing proof assistant tools [29, 45, 56, 70, 74, 80–82].

Our approach is compatible with any theorem prover and any synthesis tool as long as it uses search trees that receive tactic success feedback at every node. Formally specifying system properties is complementary to proving them. Prior work has formalized natural language specifications [15, 23, 54, 92]. Once specified, some properties, such as fairness [6, 20], can be verified probabilistically [1, 22, 31, 51, 78, 85].

The increase in AI-driven code generation risks reducing software quality [5]. Automated program repair [44] can actually break functionality and reduce overall quality [55, 71]. Poor automated code suggestions can mislead developers, again reducing quality [14]. AI-powered formal specification refinement and proof generation, such as by ProofCoop, can, in theory, improve this quality and reduce human effort [5]. Future research should integrate formal verification with automated program repair [5].

6 Contributions

We have presented ProofCoop, an approach for enabling proof synthesis models to collaborate in six different ways. Our empirical evaluation on a large benchmark showed that our preferential next-step prediction approaches outperform the best individual models, when controlling for running time, and that our subgoal proof sharing and union tactic prediction approaches outperform Diva, the state-of-the-art combination approach, which doesn't use collaboration. ProofCoop enables models with lower individual performance to be as effective as a high performance model. Our work demonstrates the potential benefits of collaboration between models developed for automated proof synthesis and supports continuing exploration of robust and effective collaboration methods.

Acknowledgments

This work is supported by the National Science Foundation under grants no. NSF CCF-1955457, CCF-2210243, and CCF-222089, and by the Defense Advanced Research Projects Agencies (DARPA) under Contract no. HR0011-24-2-0307.

References

[1] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya Nori. 2017. FairSquare: Probabilistic Verification for Program Fairness. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. Vancouver, BC, Canada, 80:1–80:30. doi:10.1145/3133904

[2] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The Multiplicative Weights Update Method: A Meta-Algorithm and Applications. *Theory of Computing* 8, 6 (2012), 121–164. doi:10.4086/toc.2012.v008a006

[3] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *International Conference on Machine Learning (ICML)*, Vol. 97. PMLR, Long Beach, CA, USA, 454–463. http://proceedings.mlr.press/v97/bansal19a.html

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. cvc4. In *International Conference on Computer Aided Verification (CAV)*. Snowbird, UT, USA, 171–177.

[5] Yuriy Brun, Saikat Chakraborty, Claire Le Goues, Corina Păsăreanu, and Adish Singla. 2026. Automatically Engineering Trusted Software: A Research Roadmap. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2026). doi:10.1145/3779132

[6] Yuriy Brun and Alexandra Meliou. 2018. Software Fairness. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results Track* (6–9). Lake Buena Vista, FL, USA, 754–759. doi:10.1145/3236024.3264838

[7] Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. 2025. Seed-prover: Deep and broad reasoning for automated theorem proving. *CoRR* abs/2507.23726 (2025). https://arxiv.org/abs/2507.23726

[8] Yizhou Chen, Zeyu Sun, Guoqing Wang, and Dan Hao. 2025. Gpass: A Goal-Adaptive Neural Theorem Prover Based on Coq for Automated Formal Verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada, 29–41. doi:10.1109/ICSE55347.2025.00116

[9] Yuri Chervonyi, Trieu H. Trinh, Miroslav Olšák, Xiaomeng Yang, Hoang Nguyen, Marcelo Menegali, Junehyuk Jung, Vikas Verma, Quoc V. Le, and Thang Luong. 2025. Gold-medalist Performance in Solving Olympiad Geometry with AlphaGeometry2. *CoRR* abs/2502.03544 (2025). https://arxiv.org/abs/2502.03544

[10] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340. doi:10.1007/978-3-540-78800-3_24

[11] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction — CADE 28*. 625–635.

[12] DeepSeek-AI, Daya Guo, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR* abs/2501.12948 (2025). https://arxiv.org/abs/2501.12948

[13] Kefan Dong and Tengyu Ma. 2025. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. In *International Conference on Machine Learning (ICML)*. https://openreview.net/forum?id=zWArMedNuW

[14] Hadeel Eladawy, Claire Le Goues, and Yuriy Brun. 2024. Automated Program Repair, What Is It Good For? Not Absolutely Nothing!. In *International Conference on Software Engineering (ICSE)* (14–20). Lisbon, Portugal, 1017–1029. doi:10.1145/3597503.3639095

[15] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proceedings of the ACM Software Engineering (PACMSE)* 1, FSE (July 2024), 84:1–84:24. doi:10.1145/3660791

[16] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic — With Proofs, Without Compromises. In *IEEE Symposium on Security and Privacy (S&P)*. 1202–1219. doi:10.1109/SP.2019.00005

[17] Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *International Conference on Software Engineering (ICSE)* (22–27). Pittsburgh, PA, USA, 749–761. doi:10.1145/3510003.3510138

[18] Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue* 4 (November 2020), 231:1–231:31. doi:10.1145/3428299

[19] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* (6–8). San Francisco, CA, USA, 1229–1241. doi:10.1145/3611643.3616243

[20] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. 2017. Fairness Testing: Testing Software for Discrimination. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (6–8). Paderborn, Germany, 498–510. doi:10.1145/3106237.3106277

[21] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. 2017. TacticToe: Learning to reason with HOL4 tactics. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, Vol. 46. 125–143.

[22] Stephen Giguere, Blossom Metevier, Yuriy Brun, Bruno Castro da Silva, Philip S. Thomas, and Scott Niekum. 2022. Fairness Guarantees under Demographic Shift. In *International Conference on Learning Representations (ICLR)* (25–29). https://openreview.net/forum?id=wbPObLm6ueA

[23] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*. Saarbrücken, Genmany, 213–224. doi:10.1145/2931037.2931061

[24] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

[25] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine Verified Network Controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, USA, 483–494. doi:10.1145/2491956.2462178

[26] Youssef Hamadi, Said Jabbour, and Jabbour Sais. 2012. Control-based clause sharing in parallel SAT solving. In *Autonomous Search*. 245–267.

[27] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. 2021. Proof Artifact Co-training for Theorem Proving with Language Models. *CoRR* abs/2102.06203 (2021). https://arxiv.org/abs/2102.06203

[28] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 342–357. doi:10.1109/FOSE.2007.29

[29] John Harrison. 1996. HOL Light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Palo Alto, CA, USA, 265–269. doi:10.1007/BFb0031814

[30] Yoshihiko Hayashi. 2025. Evaluating LLMs' Capability to Identify Lexical Semantic Equivalence: Probing with the Word-in-Context Task. In *International Conference on Computational Linguistics (COLING)*. ACL, Abu Dhabi, UAE, 6985–6998. https://aclanthology.org/2025.coling-main.466/

[31] Austin Hoag, James E. Kostas, Bruno Castro da Silva, Philip S. Thomas, and Yuriy Brun. 2023. Seldonian Toolkit: Building Software with Safe and Fair Machine Learning. In *International Conference on Software Engineering (ICSE) Demo Track* (14–20). Melbourne, Australia, 107–111. doi:10.1109/ICSE-Companion58688.2023.00035

[32] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2018. GamePad: A Learning Environment for Theorem Proving. *CoRR* (2018). https://arxiv.org/abs/1806.00608

[33] Atalay İleri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2018. Proving Confidentiality in a File System Using DiskSec. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, USA, 323–338. https://www.usenix.org/conference/osdi18/presentation/ileri

[34] Kevin Jacobs and Benjamin Beurdouche. 2020. Performance Improvements via Formally-Verified Cryptography in Firefox. https://blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/.

[35] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees Through Formal Shim Verification. In *USENIX Security Symposium (USENIX Security)*. Bellevue, WA, USA, 113–128. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang

[36] Albert Jiang, Konrad Czechowski, Mateja Jamnik, Piotr Milos, Szymon Tworkowski, Wenda Li, and Yuhuai Tony Wu. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *Neural Information Processing Systems (NeurIPS)*. New Orleans, LA, USA.

[37] Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. In *Conference on Artificial Intelligence and Theorem Proving (AITP)*. Aussois, France, 17.1–17.3.

[38] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=SMa9EAovKMC

[39] Saketh Ram Kasibatla, Arpan Agrawal, Yuriy Brun, Sorin Lerner, Talia Ringer, and Emily First. 2026. Cobblestone: A Divide-and-Conquer Approach for Automating Formal Verification. In *Proceedings of the 48th International Conference on Software Engineering (ICSE)* (15–17). Rio de Janeiro, Brazil. doi:10.1145/3744916.3773178

[40] Zhanna Kaufman, Emily First, Alex Sanchez-Stern, Kyle Thompson, Sorin Lerner, and Yuriy Brun. 2026. ProofCoop Replication Package. https://osf.io/qeyk9.

[41] Herb Krasner. 2022. The Cost of Poor Software Quality in the US: A 2022 Report. https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.

[42] Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 35. 26337–26349.

[43] Chris Lattner and Nikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*. Palo Alto, CA, USA, 75–86. doi:10.1109/CGO.2004.1281665

[44] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. doi:10.1145/3318162

[45] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Dakar, Senegal. doi:10.1007/978-3-642-17511-4_20

[46] Xavier Leroy. 2006. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Charleston, SC, USA, 42–54. doi:10.1145/1111037.1111042

[47] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM (CACM)* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[48] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. 2025. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *CoRR* abs/2508.03613 (2025). https://arxiv.org/abs/2508.03613

[49] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Sacramento, CA, USA. doi:10.1145/3691620.3695521

[50] Matteo Marescotti, Antti EJ Hyvärinen, and Natasha Sharygina. 2016. Clause sharing and partitioning for cloud-based SMT solving. In *Automated Technology for Verification and Analysis (ATVA)*. Chiba, Japan, 428–443.

[51] Blossom Metevier, Stephen Giguere, Sarah Brockman, Ari Kobren, Yuriy Brun, Emma Brunskill, and Philip S. Thomas. 2019. Offline Contextual Bandits with High Probability Fairness Guarantees. In *Annual Conference on Neural Information Processing Systems (NeurIPS), Advances in Neural Information Processing Systems 32* (9–14). Vancouver, BC, Canada, 14893–14904. http://papers.neurips.cc/paper/9630-offline-contextual-bandits-with-high-probability-fairness-guarantees

[52] Maciej Mikuła, Szymon Antoniak, Szymon Tworkowski, Albert Qiaochu Jiang, Jin Peng Zhou, Christian Szegedy, Łukasz Kuciński, Piotr Miłoś, and Yuhuai Wu. 2023. Magnushammer: A Transformer-based Approach to Premise Selection. https://arxiv.org/abs/2303.04488

[53] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China. doi:10.1145/2345156.2254111

[54] Manish Motwani and Yuriy Brun. 2019. Automatically Generating Precise Oracles from Structured Natural Language Specifications. In *International Conference on Software Engineering (ICSE)* (29–31). Montreal, QC, Canada, 188–199. doi:10.1109/ICSE.2019.00035

[55] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2022. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering (TSE)* 48, 2 (February 2022), 637–661. doi:10.1109/TSE.2020.2998785

[56] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[57] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *Conference on Artificial Intelligence (AAAI)*. AAAI Press, New York, NY, USA, 2967–2974.

[58] Larry Paulson and Tobias Nipkow. 2023. The Sledgehammer: Let Automatic Theorem Provers write your Isabelle scripts! https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html.

[59] Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *CoRR* (2020). https://arxiv.org/abs/2009.03393

[60] Joseph E Reeves, Marijn JH Heule, and Randal E Bryant. 2023. Preprocessing of propagation redundant clauses. *Journal of Automated Reasoning* 67, 3 (2023), 31.

[61] Talia Ringer. 2021. *Proof Repair*. Ph. D. Dissertation. University of Washington.

[62] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. *Foundations and Trends® in Programming Languages* 5, 2–3 (2019), 102–281. doi:10.1561/2500000045

[63] Nick Robins-Early. 2024. CrowdStrike global outage to cost US Fortune 500 companies $5.4bn. *The Guardian* (July 2024). https://www.theguardian.com/technology/article/2024/jul/24/crowdstrike-outage-companies-cost

[64] Neha Rungta. 2022. A billion SMT queries a day. In *International Conference on Computer Aided Verification (CAV)*. Haifa, Israel, 3–18.

[65] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem Proving. *CoRR* abs/2401.02949 (2024). http://arxiv.org/abs/2401.02949

[66] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. 1–10.

[67] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 45, 2, Article 12 (June 2023), 12:1-12:30 pages. doi:10.1145/3593374

[68] Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Dylan Zhang, Talia Ringer, and Yuriy Brun. 2025. QEDCartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)* (28–30). Ottawa, ON, Canada, 307–320. doi:10.1109/ICSE55347.2025.00033

[69] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proceedings of the ACM on Programming Languages (PACMPL)* 2, POPL (Dec. 2017), 28:1–28:30. doi:10.1145/3158116

[70] Konrad Slind and Michael Norrish. 2008. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Montreal, QC, Canada, 28–32. doi:10.1007/978-3-540-71067-7_6

[71] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (2–4). Bergamo, Italy, 532–543. doi:10.1145/2786805.2786825

[72] Jean Souyris. 2014. Industrial Use of CompCert on a Safety-Critical Software Product. http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf.

[73] Richard M. Stallman. 2012. *Using the GNU Compiler Collection*. Free Software Foundation, Boston, MA, USA.

[74] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub,

Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, USA, 256–270. doi:10.1145/2914770.2837655

[75] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *Conference on Language Modeling (COLM)*. Philadelphia, PA, USA.

[76] The Coq Development Team. 2017. Coq, v.8.7. https://coq.inria.fr.

[77] Ken Thomas. 2010. 89 deaths tied to Toyota acceleration, U.S. says. *NBC News* (May 2010). https://www.nbcnews.com/id/wbna37345463

[78] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. 2019. Preventing Undesirable Behavior of Intelligent Machines. *Science* 366, 6468 (22 November 2019), 999–1004. doi:10.1126/science.aag3311

[79] Robert 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)* (28–30). Ottawa, ON, Canada, 347–359. doi:10.1109/ICSE55347.2025.00161

[80] Andrzej Trybulec and Howard A Blair. 1985. Computer Assisted Reasoning with MIZAR. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, Vol. 85. Los Angeles, CA, USA, 26–28. https://www.ijcai.org/Proceedings/85-1/Papers/006.pdf

[81] Niki Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* Ph. D. Dissertation. University of California, San Diego.

[82] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2020. *Programming Language Foundations in Agda.* http://plfa.inf.ed.ac.uk/20.07/

[83] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, et al. 2024. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *International Conference on Learning Representations (ICLR)*.

[84] Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024. Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference. http://arxiv.org/abs/2412.13663

[85] Aline Weber, Blossom Metevier, Yuriy Brun, Philip S. Thomas, and Bruno Castro da Silva. 2025. Beyond Prediction: Managing the Repercussions of Machine Learning Applications. In *39th Annual Conference on Neural Information Processing Systems (NeurIPS), Advances in Neural Information Processing Systems 38* (2–7). San Diego, CA, USA and Mexico City, Mexico.

[86] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, 357–368.

[87] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. 2009. A concurrent portfolio approach to SMT solving. In *International Conference on Computer Aided Verification (CAV)*. Grenoble, France, 715–720.

[88] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://openreview.net/forum?id=edmYVRkYZv

[89] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*. Long Beach, CA, USA. http://proceedings.mlr.press/v97/yang19a/yang19a.pdf

[90] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*.

[91] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA, 283–294. doi:10.1145/1993498.1993532

[92] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 25–37. doi:10.1145/3368089.3409716

[93] Xueliang Zhao, Wenda Li, and Lingpeng Kong. 2023. Decomposing the enigma: Subgoal-based demonstration learning for formal theorem proving. *CoRR* abs/2305.16366 (2023). https://arxiv.org/abs/2305.16366

[94] Chuanyang Zheng, Haiming Wang, Enze Xie, Zhengying Liu, Jiankai Sun, Huajian Xin, Jianhao Shen, Zhenguo Li, and Yu Li. 2023. Lyra: Orchestrating dual correction in automated theorem proving. *CoRR* abs/2309.15806 (2023). http://arxiv.org/abs/2309.15806