Generating Correctness Proofs with Neural Networks

Anonymous Author(s)

Abstract

Foundational verification allows programmers to build software which has been empirically shown to have high levels of assurance in a variety of important domains. However, the cost of producing foundationally verified software remains prohibitively high for most projects, as it requires significant manual effort by highly trained experts. In this paper we present Proverbot9001, a proof search system using machine learning techniques to produce proofs of software correctness in interactive theorem provers. We demonstrate Proverbot9001 on the proof obligations from a large practical proof project, the CompCert verified C compiler, and show that it can effectively automate what were previously manual proofs, automatically producing proofs for 28% of theorem statements in our test dataset, when combined with solver-based tooling. Without any additional solvers, we exhibit a proof completion rate that is a 4X improvement over prior state-of-the-art machine learning models for generating proofs in Coq.

Keywords Machine-learning, Theorem proving

1 Introduction

A promising approach to software verification is *foundational verification*. In this approach, programmers use an interactive theorem prover, such as Coq [13] or Isabelle/HOL [33], to state and prove properties about their programs. Foundational verification has shown increasing promise over the past two decades; it has been used to prove properties of programs in a variety of settings, including compilers [26], operating systems [22], database systems [29], file systems [8], distributed systems [37], and cryptographic primitives [3].

One of the main benefits of foundational verification is that it provides high levels of assurance. The interactive theorem prover makes sure that proofs of program properties are done in full and complete detail, without any implicit assumptions or forgotten proof obligations. Furthermore, once a proof is completed, foundational proof assistants can generate a representation of the proof in a foundational logic; these proofs can be checked with a small kernel. In this setting only the kernel needs to be trusted (as opposed to the entire proof assistant), leading to a small trusted computing base. As an example of this high-level of assurance, a study of compilers [39] has shown that CompCert [26], a compiler proved correct in the Coq proof assistant, is significantly more robust than its non-verified counterparts.

Unfortunately, the benefits of foundational verification come at a great cost. The process of performing proofs in a proof assistant is extremely laborious. CompCert [26] took 6 person-years and 100,000 lines of Coq to write and verify, and seL4 [22], which is a verified version of a 10,000 line operating system, took 22 person-years to verify. The sort of manual effort is one of the main impediments to the broader adoption of proof assistants.

In this paper, we present Proverbot9001, a novel system that uses machine learning to help alleviate the manual effort required to complete proofs in an interactive theorem prover. Proverbot9001 trains on existing proofs to learn models. Proverbot9001 then incorporates these learned models in a tree search process to complete proofs. The source of Proverbot9001 is publicly available on GitHub [1].

The main contribution of this paper is bringing domain knowledge to the feature engineering, model architecture, and search procedures of machine-learning based systems for interactive theorem proving. In particular, our work distinguishes itself from prior work on machine learning for proofs in three ways:

1. A two part tactic-prediction model, in which prediction of tactic arguments is primary and informs prediction of tactics themselves.
2. An argument prediction architecture which makes use of recurrent neural networks over sequential representations of terms.
3. Several effective tree pruning techniques inside of a prediction-guided proof search.

We tested Proverbot9001 end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 files[2]. When combined with solver-based tooling (which alone can only solve 7% of proofs), Proverbot9001 can automatically produce proofs for 28% of the theorem statements in our test dataset (138/501). In our default configuration without external solvers, Proverbot9001 solves (produces a checkable proof for) 19.36% (97/501) of the proofs in our test set, which is a nearly 4X improvement over the previous state of the art system that attempts the same task [38]. Our model is able to reproduce the tactic name from the solution 32% of the time; and when the tactic name is correct, our model is able to predict the solution argument 89% of the time. We also show that Proverbot9001 can be trained on one project and then effectively predict on another project.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

[1] Link removed for double-blind review

[2] This training/test split comes from splitting the dataset 90/10, and then removing from the test set files that don't contain proofs.

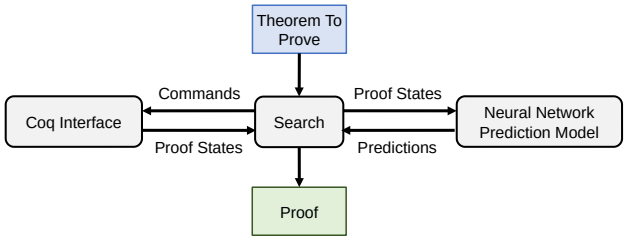

Figure 1. The overall architecture of Proverbot9001, built using CoqSerapi, Python, and PyTroch.

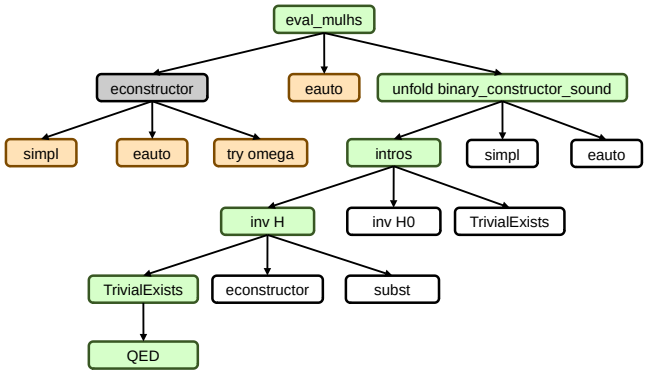

Figure 2. A graph of a Proverbot9001 search. In green are the tactics that formed part of the discovered solution, as well as the lemma name and the QED. In orange are nodes that resulted in a context that is at least as hard as one previously found (see Section 6).

2 Overview

In this section, we'll present Proverbot9001's prediction and search process with an example from CompCert. You can see the top-level structure of Proverbot9001 in Figure 1.

Consider the following theorem from the CompCert compiler:

```
Definition binary_constructor_sound
    (cstr: expr -> expr -> expr)
    (sem: val -> val -> val) : Prop :=
  forall le a x b y,
  eval_expr ge sp e m le a x ->
  eval_expr ge sp e m le b y ->
  exists v, eval_expr ge sp e m le (cstr a b) v
         /\ Val.lessdef (sem x y) v.

Theorem eval_mulhs:
  binary_constructor_sound mulhs Val.mulhs.
Proof.
...
```

This theorem states that the `mulhs` expression constructor is sound with respect to the specification `Val.mulhs`.

At the beginning of the proof of `eval_mulhs`, Proverbot9001 predicts three candidate tactics, `econstructor`, `eauto`, and `unfold binary_constructor_sound`. Once these predictions are made, Proverbot9001 tries running all three, which results in three new states of the proof assistant. In each of these three states, Proverbot9001 again makes predictions for what the most likely tactics are to apply next. These repeated predictions create a search tree, which Proverbot9001 explores in a depth first way. The proof command predictions that Proverbot9001 makes are ordered by likelihood, and the search explores more likely branches first.

Figure 2 shows the resulting search tree for `eval_mulhs`. The nodes in green are the nodes that produce the final proof. Orange nodes are predictions that fail to make progress on the proof (see Section 6); these nodes are not expanded further. All the white nodes to the right of the green path are not explored, because the proof in the green path is found first.

$\mathcal{T}$	Tactics
$\mathcal{A}$	Tactic arguments
$\mathcal{C} = \mathcal{T} \times \mathcal{A}$	Proof commands
$\mathcal{I}$	Identifiers
$\mathcal{Q}$	Propositions
$\mathcal{G} = \mathcal{Q}$	Goals
$\mathcal{H} = \mathcal{I} \times \mathcal{Q}$	Hypotheses
$\mathcal{O} = [\mathcal{H}] \times \mathcal{G}$	Obligations
$\mathcal{S} = [\mathcal{O} \times [\mathcal{C}]]$	Proof states

Figure 3. Formalism to model a Proof Assistant

3 Definitions

In the rest of the paper, we will describe the details of how Proverbot9001 works. We start with a set of definitions that will be used throughout. In particular, Figure 3 shows the formalism we will use to represent the state of an in-progress proof. A tactic $\tau \in \mathcal{T}$ is a tactic name. An argument $a \in \mathcal{A}$ is a tactic argument. For simplicity of the formalism, we assume that all tactics take zero or one arguments. We use $\mathcal{I}$ for the set of Coq identifiers, and $\mathcal{Q}$ for the set of Coq propositions. A *proof state* $\sigma \in \mathcal{S}$ is a state of the proof assistant, which consists of a list of obligations along with their proof command history. We use $[X]$ to denote the set of lists of elements from X. An obligation is a pair of: (1) a set of hypotheses (2) a goal to prove. A hypothesis is a proposition named by an identifier, and a goal is a proposition.

4 Predicting a Single Proof Step

We start by explaining how we predict individual steps in the proof. Once we have done this, we will explain how we use these proof command predictions to guide a proof search procedure.

We define $\mathcal{D}[\tau]$ to be a scoring function over τ, where larger scores are preferred over smaller ones:

$$\mathcal{D}[\tau] = \tau \to \mathbb{R}$$

We define a τ-predictor $\mathcal{R}[\tau]$ to be a function that takes a proof state $\sigma \in \mathcal{S}$ (*i.e.* a state of the proof assistant under which we want to make a prediction) and returns a scoring function over τ. In particular, we have:

$$\mathcal{R}[\tau] = \mathcal{S} \to \mathcal{D}[\tau]$$

Our main predictor P will be a predictor of the next step in the proof, *i.e.* a predictor for proof commands:

$$P : \mathcal{R}[\mathcal{T} \times \mathcal{A}]$$

We divide our main predictor into two predictors, one for tactics, and one for arguments:

$$P_{tac} : \mathcal{R}[\mathcal{T}]$$

$$P_{arg} : \mathcal{T} \to \mathcal{R}[\mathcal{A}]$$

Our main predictor P combines P_{tac} and P_{arg} as follows:

$$P(\sigma) = \lambda(\tau, a) \,.\, P_{tac}(\sigma)(\tau) \otimes P_{arg}(\tau)(\sigma)(a)$$

where $\otimes$ is an operator that combines the scores of the tactic and the argument predictors. We now describe the three parts of this prediction architecture in turn: P_{tac}, P_{arg}, and $\otimes$.

4.1 Predicting Tactics (P_{tac})

To predict tactics, Proverbot9001 uses of a set of manually engineered features to reflect important aspects of proof prediction: (1) the head of the goal as an integer (2) the name of the previously run tactic as an integer (3) a hypothesis that is heuristically chosen (based on string similarity to goal) as being the most relevant to the goal (4) the similarity score of this most relevant hypothesis.

These features are embedded into a continuous vector of 128 floats using a standard word embedding, and then fed into a fully connected feed-forward neural network (3 layers, 128 nodes-wide) with a softmax (normalizing) layer at the end, to compute a probability distribution over possible tactic names. This architecture is trained on 153402 samples with a stochastic gradient descent optimizer.

The architecture of this model is shown in Figure 4. Blue boxes represent input; purple boxes represent intermediate encoded values; green boxes represent outputs; and gray circles represent computations. The NN circle is the feed-forward Neural Network mentioned above. The Enc circle is a word embedding module.

4.2 Predicting Tactic Arguments (P_{arg})

Once a tactic is predicted, Proverbot9001 next predicts arguments. Recall that the argument predictor is a function $P_{arg} : \mathcal{R}[\mathcal{A}]$. In contrast to previous work, our argument model is a prediction architecture in its own right.

Proverbot9001 currently predicts zero or one tactic arguments; However, since the most often-used multi-argument Coq tactics can be desugared to sequences of single argument tactics (for example "`unfold a, b`" to "`unfold a. unfold b.`"), this limitation does not significantly restrict our expressivity in practice.

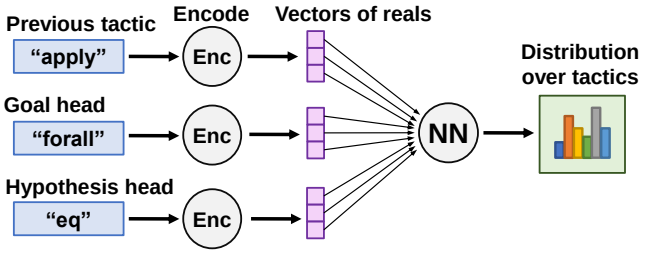

Figure 4. Proverbot9001's model for predicting tactics. Takes as input three features for each data point: the previous tactic run, the head token of the goal, and of the most relevant hypothesis (see Section 4.1). We restrict the previous tactic feature to the 50 most common tactics, and head tokens on goal and hypothesis to the 100 most common head tokens.

Proverbot9001 makes three kinds of predictions for arguments: *goal-token* arguments, *hypothesis* arguments, *lemma* arguments:

Goal-token arguments are arguments that are a single token in the goal; for instance, if the goal is `not (eq x y)`, we might predict `unfold not`, where `not` refers to the first token in the goal. In the case of tactics like `unfold` and `destruct`, the argument is often (though not always) a token in the goal.

Hypothesis arguments identifiers referring to a hypothesis in context. For instance, if we have a hypothesis `H` in context, with type `is_path (cons (pair s d) m)`, we might predict `inversion H`, where `H` refers to the hypothesis, and `inversion` breaks it down. In the case of tactics like `inversion` and `destruct`, the argument is often a hypothesis identifier.

Finally, *lemma* arguments are identifiers referring to a previously defined proof. These can be basic facts in the standard library, like

```
plus_n_0 : forall n : nat, n = n + 0
```

or a lemma from the current project, such as the `eval_mulhs` described in the overview. In Proverbot9001, lemmas are considered from a subset of the possible lemma arguments available in the global context, in order to make training tractable. Proverbot9001 supports several different modes for determining this subset; by default we consider lemmas defined previously in the current file.

The architecture of the scoring functions for these argument types is shown in Figure 5. One recurrent neural network (RNN) is used to give scores to each hypothesis and lemma by processing the type of the term, and outputting a final score. A different RNN is then used to process the goal, assigning a score to each token in processes.

4.3 Combining Tactic and Argument Scores ($\otimes$)

The $\otimes$ operator attempts to provide a balanced combination of tactic and argument prediction, taking both into account even across different tactics. The operator works as follows.

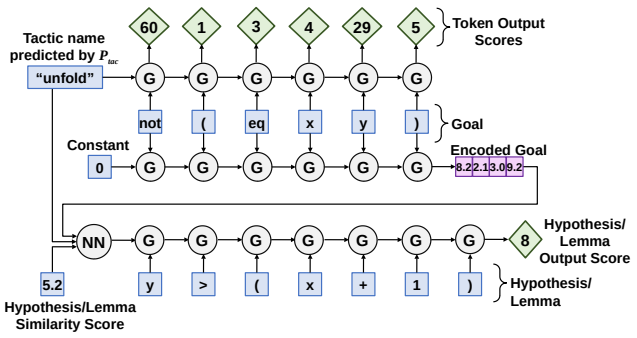

Figure 5. The model for scoring possible arguments.

We pick the n highest-scoring tactics and for each tactic the m highest-scoring arguments. We then score each proof command by multiplying the tactic score and the argument score, without any normalization. Formally, we can implement this approach by defining $\otimes$ to be multiplication, and by not normalizing the probabilities produced by P_{arg} until all possibilities are considered together.

Because we don't normalize the probabilities of tactics, the potential arguments for a tactic are used in determining the eligibility of the tactic itself (as long as that tactic is in the top n). This forms one of the most important contributions of our work: the argument selection is primary, with the tactic prediction mostly serving to help prune it's search space.

4.4 Putting it all together

The overall architecture that we have described is shown in Figure 6. The P_{tac} predictor (whose detailed structure is shown in Figure 4) computes a distribution over tactic using three features as input: the previous tactic, head constructor of goal, and head constructor of the hypothesis deemed most relevant. Then, for each of the top tactic predicted by P_{tac}, the P_{arg} predictor (whose detailed structure is shown in Figure 5) is invoked. In addition to the tactic name, the P_{arg} predictor takes several additional inputs: the goal, the hypotheses in context, and the similarity between each of those hypotheses and the goal. The P_{arg} predictor produces scores for each possible argument (in our case one score for each token in the goal, and one score the single hypothesis). These scores are combined with $\otimes$ to produce an overall scoring of proof commands.

5 Training

5.1 Training Architecture

Figure 7 shows the training architecture for the tactic predictor, P_{tac} (recall that the detailed architecture of P_{tac} is shown in Figure 4). Training is done through a stochastic gradient descent optimizer, with Negative Log Likelihood Loss (NLLLoss) as the criterion.

Figure 8 shows the training architecture for the argument predictor, P_{arg} (recall that the detailed architecture of P_{arg} is shown in Figure 5). Note that it is very important for us to inject the tactics predicted by P_{tac} into the input of the argument model P_{arg}, instead of using just the correct tactic name. This allows the scores produced by the argument model to be comparable *across* different predicated tactic. Once the argument model P_{arg} computes a score for each possible argument, we combine these predictions using $\otimes$ to get a distribution of scores over tactic/argument pairs. Finally, this distribution, along with the correct tactic/argument pair is passed to a module that computes changes to the weights based on the NLLLoss criterion. In our main CompCert benchmark the 153402 tactic samples from the training set are processed for 20 epochs.

5.2 Learning From Higher-order Proof Commands

Proof assistants generally have higher-order proof commands, which are tactics that take other proof commands as arguments; in Coq, these are called *tacticals*. While higher-order proof commands are extremely important for human proof engineers, they are harder to predict automatically because of their generality. While some previous work [38] attempts to learn directly on data which uses these higher-order proof commands, we instead takes the approach of desugaring higher-order proof commands into first-order ones as much as possible; this makes the data more learnable, without restricting the set of expressible proofs.

6 Prediction-Guided Search

Now that we have explained how we predict a single step in the proof, we describe how Proverbot9001 uses these predictions in a proof search.

In general, proof search works by transitioning the proof assistant into different states by applying proof commands, and backtracking when a given part of the search space has either been exhausted, or deemed unviable. Exhaustive proof search in proof assistants is untenable because the number of possible proof commands to apply is large. Instead, we use the predictor described above to guide the search. Aside from using these predictions, the algorithm is a straightforward depth-limited search, with three subtleties.

First we stop the search when we find a proof goal that is at least as hard (by a syntactic definition) as a goal earlier in the history. While in general it is hard to formally define what makes one proof state harder than another, there are some obvious cases which we can detect. A proof state with a superset of the original obligations will be harder to prove, and a proof state with the same goal, but fewer assumptions, will be harder to prove.

To formalize this intuition, we define a relation $\geq$ between states such that $\sigma_1 \geq \sigma_2$ is meant to capture "Proof state σ_1 is at least as hard as proof state σ_2". We say that $\sigma_1 \geq \sigma_2$ if and only if for all obligations O_2 in σ_2 there exists an obligation O_1 in σ_1 such that $O_1 \geq_o O_2$. For obligations O_1 and O_2, we

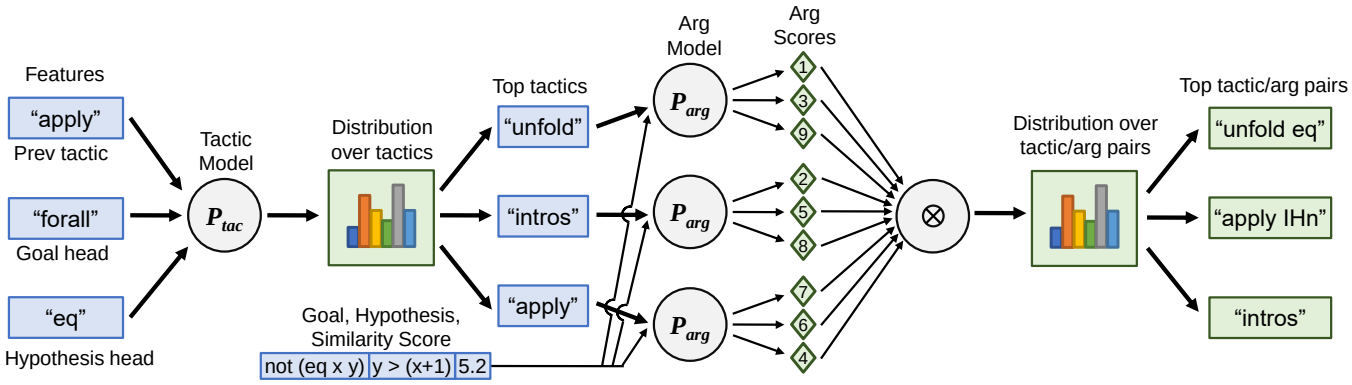

Figure 6. The overall prediction model, combining the tactic prediction and argument prediction models.

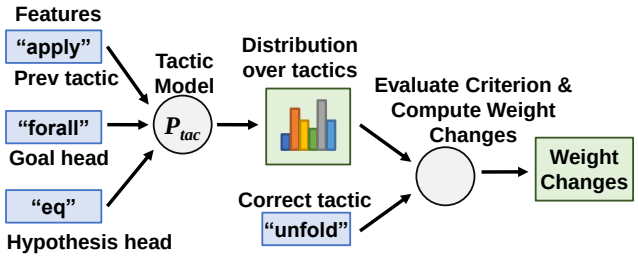

Figure 7. The architecture for training the tactic models.

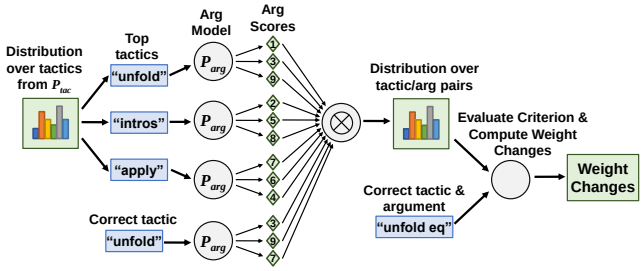

Figure 8. The architecture for training the argument models. Note that we inject predicted tactics into the input of the argument model, instead of just using the correct tactic, so that argument scores will be comparable.

say that $O_1 \geq_o O_2$ if and only if each hypothesis in O_1 is also a hypothesis in O_2, and the goals of O_1 and O_2 are the same.

Since $\geq$ is reflexive, this notion allows us to generalize all the cases above to a single pruning criteria: "proof command prediction produces a proof state which is $\geq$ than a proof state in the history".

Second when backtracking, we do not attempt to find a different proof for an already proven sub-obligation. While in general this can lead to missed proofs because of existential variables (typed holes filled based on context), this has not been an issue for the kinds of proofs we have worked with so far.

Third we had to adapt our notion of search "depth" to the structure of Coq proofs (in which a tactic can produce multiple sub-obligations). A naïve tree search through the Coq proof space will fail to exploit some of the structure of sub-proofs in Coq.

Consider for example the following two proofs:

1. `intros. simpl. eauto.`
2. `induction n. eauto. simpl.`

At first glance, it seems that both of these proofs have a depth of three. This means that a straightforward tree search (which is blind to the structure of subproofs) would not find either of these proofs if the depth limit were set to two.

However, there is a subtlety in the second proof above which is important (and yet not visible syntactically). Indeed, the `induction n` proof command actually produces two obligations ("sub-goals" in the Coq terminology). These correspond to the base case and the inductive case for the induction on `n`. Then `eauto` discharges the first obligation (the base case), and `simpl` discharges the second obligation (the inductive case). So in reality, the second proof above really only has a depth of two, not three.

Taking this sub-proof structure into account is important because it allows Proverbot9001 to discover more proofs for a fixed depth. In the example above, if the depth were set to two, and we used a naïve search, we would not find either of the proofs. However, at the same depth of two, a search which takes the sub-proof structure into account would be able to find the second proof (since this second proof would essentially be considered to have a depth of two, not three).

7 Evaluation

This section shows that Proverbot9001 is able to successfully solve many proofs. We also experimentally show that Proverbot9001 improves significantly on the state-of-the-art presented in previous work.

First, in Section 7.2, we compare experimentally to previous work, by running both Proverbot9001 and the CoqGym [38] project on CompCert, in several configurations

outlined in the CoqGym paper. Next, in Section 7.3, we experiment with using the weights learned from one project to produce proofs in another. Then, in Section 7.4, we show the "hardness" of proofs that Proverbot9001 is generally able complete, using the length of the original solution as proxy for proof difficulty. Finally, in Section 7.5, we measure the predictor subsystem, without proof search. Additional evaluation can be found in the appendix.

Experiments were run on two machines. Machine A is an Intel i7 machine with 4 cores, a NVIDIA Quadro P4000 8BG 256-bit, and 20 gigabytes of memory. Machine B is Intel Xeon E5-2686 v4 machine with 8 cores, a Nvidia Tesla v100 16GB 4096-bit, and 61 gigabytes of memory. Experiment running uses GNU Parallel [36].

During the development of Proverbot9001, we explored many alternatives, including n-gram/bag-of-words representations of terms, a variety of features, and several core models including k-nearest neighbors, support vector machines, and several neural architectures. While we include here some experiments that explore high-level design decisions (such as training and testing on the same projects vs cross project, working with and without solver-based tooling, modifying the search depth and width, and running with and without pre-processing), we also note that in the development of a large system tackling a hard problem, it becomes intractable to evaluate against every possible permutation of every design decision. In this setting, we are still confident in having demonstrated a system that works for the specific problem of generating correctness proof with performance that outperforms the state-of-the-art techniques by many folds.

7.1 Summary of Results

Proverbot9001, run using CoqHammer [10] and the default configuration, is able to produce proofs for 28% of the theorem statements in CompCert. This represents a 2.4X improvement over the previous state-of-the-art. Without any external tooling, Proverbot9001 can produce proofs for 19.36%, an almost 4X improvement over previous state-of-the-art prediction-based proofs. Our core prediction model is able to reproduce the tactic name from the solution 32% of the time; and when the tactic name is correct, our model is able to predict the solution argument 89% of the time. We also show that Proverbot9001 can be trained on one project and then effectively predict on another project.

7.2 Experimental Comparison to Previous Work

We tested Proverbot9001 end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 different files. On our default configuration, Proverbot9001 solves 19.36% (97/501) of the proofs in our test set.

In addition to running Proverbot9001 on CompCert, we ran the CoqGym [38] tool, which represents the state of the art in this area, on the same dataset in several configurations.

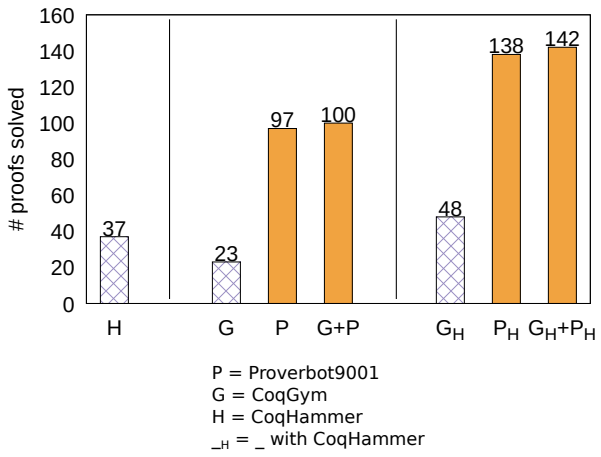

Figure 9. A comparison of Proverbot9001 and CoqGym's abilities to complete proofs. H stands for CoqHammer by itself, as a single invocation; G stands for CoqGym by itself; P stands for Proverbot9001 by itself; G+P stands for the union of proofs done by G or P; G_H stands for CoqGym with CoqHammer; P_H stands for Proverbot9001 with CoqHammer; G_H+P_H stands for the union of proofs done by G_H or P_H.

To account for differences in training dataset, we ran CoqGym with their original training schema, and also our training schema, and reported the best of the two numbers. CoqGym is intended to be combined with a solver based proof-procedure, CoqHammer [10], which is run after every proof command invocation. While our system was not originally designed this way, we compare both systems using CoqHammer, as well as both systems without. We also compared our system to using CoqHammer on the initial goal directly, which simultaneously invokes Z3 [12], CVC4 [6], Vampire [25], and E Prover [34], in addition to attempting to solve the goal using a crush-like tactic [9].

Figure 9 shows the proofs solved by various configurations. The configurations are described in the caption. For all configurations, we ran Proverbot9001 with a search depth of 6 and a search width of 3 (see Section 9.5). Note that in Figure 9 the bars for H, G, and G_H are prior work. The bars P, G+P and G_H+P_H are the ones made possible by our work.

When CoqHammer is not used, Proverbot9001 can complete nearly 4 times the number of proofs that are completed by CoqGym. In fact, even when CoqGym is augmented with CoqHammer Proverbot9001 by itself (without CoqHammer) still completes 39 more proofs, which is a 67% improvement (and corresponds to about 8% of the test set). When enabling CoqHammer in both CoqGym and Proverbot9001, we see that CoqGym solves 48 proofs whereas Proverbot9001 solves 138 proofs, which is a 2.88X improvement over the state of art.

Finally, CoqGym and Proverbot9001 approaches are complementary; both can complete proofs which the other cannot. Therefore, one can combine both tools to produce

more solutions than either alone. Combining CoqGym and Proverbot9001, without CoqHammer, allows us to complete 100/501 proofs, a proof success rate of 20%. Combining Proverbot9001 and CoqGym, each with CoqHammer, allows us to solve 142/501 proofs, a success rate of 28%. It's important to realize that, whereas the prior state of the art was CoqGym with CoqHammer, at 48 proofs, by combining CoqGym and Proverbot9001 (both with CoqHammer), we can reach a grand total of 142 proofs, which is a 2.96X improvement over the prior state of art.

7.3 Cross-Project Predictions

To test Proverbot9001's ability to make use of training across projects, we used the weights learned from CompCert, and and ran Proverbot9001 in it's default configuration on three other Coq projects from the Coq Contrib collection, `concat`, `float`, and `zfc`.

`concat` is a library of constructive category theory proofs, which showcases Coq proofs of mathematical concepts instead of program correctness. The `concat` library is made of 514 proofs across 105 files; Proverbot9001 was able to successfully produce a proof for 91 (17.7%) of the extracted theorem statements, without the use of CoqHammer.

`float` is a formalization of floating point numbers, made of 742 proofs across 38 files; Proverbot9001 was able to successfully produce a proof for 100 (13.48%) proofs.

`zfc` is a formalization of set theory made of 241 proofs across 78 files; 41 (17.01%) were successfully completed.

The comparable number for CompCert was 19.36%.

These results demonstrate not only that Proverbot9001 can operate on proof projects in a variety of domains, but more importantly that it can effectively transfer training from one project to another. This would allow programmers to use Proverbot9001 even in the initial development of a project, if it had been previously trained on other projects.

7.4 Original Proof Length vs Completion Rate

In Figure 10 and Figure 11, we plot a histogram of the original proof lengths (in proof commands) vs the number of proofs of that length. We break down the proofs by (from bottom to top) number we solve, number we cannot solve but still have unexplored nodes, and number run out of unexplored nodes before finding a solution. Note that for the second class (middle bar), it's possible that increasing the search depth would allow us to complete the proof. Figure 10 shows proofs of length 10 or below, and Figure 11 shows all proofs, binned in sets of 10.

There are several observations that can be made. *First*, most original proofs in our test set are less than 20 steps long, with a heavy tail of longer proofs. *Second*, we do better on shorter proofs. Indeed, 51% (256/501) of the original proofs in our test set are ten proof commands or shorter, and of those proofs, we can solve 35% (89/256), compared to our overall solve rate of 19.36% (97/501). *Third*, we are in some cases able to handle proofs whose original length is longer then

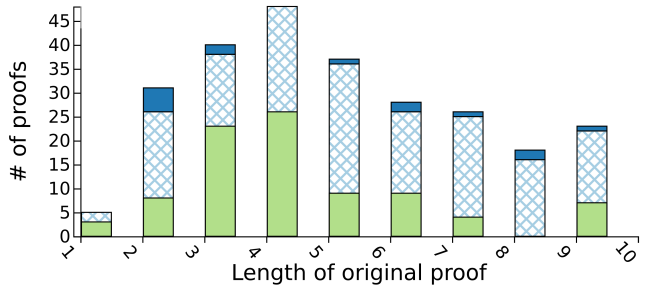

Figure 10. A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes, for proofs with length 10 or less. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted without finding a solution.

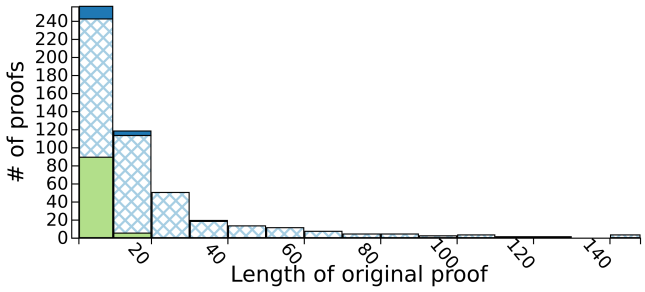

Figure 11. A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted without finding a solution. Note that most proofs are between 0 and 10 proof commands long, with a long tail of much longer proofs.

10. Indeed, 7 of the proofs we solve (out of 79 solved) had an original length longer than 10. In fact, the longest proof we solve is originally 25 proof commands long; linearized it's 256 proof commands long. Our solution proof is 267 (linear) proof commands long, comparable to the original proof, with frequent case splits. The depth limit for individual obligations in our search was 6 in all of these runs.

7.5 Individual Prediction Accuracy

We want to measure the effectiveness of the predictor subsystem that predicts proof command pairs (the P function defined in Section 4). To do this, we broke the test dataset down into individual (linearized) proof commands, and ran to just before each proof command to get it's prediction context. Then we fed that context into our predictor, and compared the result to the proof command in the original solution. Of all the proof commands in our test dataset, we are able to predict 28.66% (3784/13203) accurately. This includes the

correct tactic and the correct argument. If we only test on the proof commands which are in Proverbot9001's prediction domain, we are able to predict 39.25% (3210/8178) accurately.

During search, our proof command predictor returns the top N tactics for various values of N, and all of these proof commands are tried. Therefore, we also measured how often the proof command in the original proof is in the top 3 predictions, and the top 5 predictions. For all proof commands in the data set, the tactic in the original proof is in our top 3 predictions 38.93% of the time, and in our top 5 predictions 42.66% of the time. If we restrict to proof commands in Proverbot9001's prediction domain, those numbers are 52.17% and 60.39%.

8 Related Work

8.1 Program Synthesis

Program Synthesis is the automatic generation of programs from a high-level specification [17]. This specification can come in many forms, the most common being a logical formula over inputs and outputs, or a set of input-output examples. Programs generated can be in a variety of paradigms and languages, often domain-specific. Our tool, Proverbot9001, is a program synthesis tool that focuses on synthesis of proof command programs.

Several program synthesis works have used types extensively to guide search. Some work synthesizes programs purely from their types [18], while other work uses both a type and a set of examples to synthesize programs [14, 31]. In Proverbot9001, the programs being synthesized use a term type as their specification, however, the proof command program itself isn't typed using that type, rather it must generate a term of that type (through search).

Further work in [27] attempts to learn from a set of patches on GitHub, general rules for inferring patches to software. This work does not use traditional machine learning techniques, but nevertheless learns from data, albeit in a restricted way.

8.2 Machine Learning for Code

Machine learning for modeling code is a well explored area [2], as an alternative to more structured methods of modeling code. Several models have been proposed for learning code, such as AST-like trees [30], long-term language models [11], and probabilistic grammars [7]. Proverbot9001 does not attempt to be so general, using a model of programs that is specific to its domain, allowing us to capture the unique dependencies of proof command languages. While the model is simple, it is able to model real proofs better than more general models in similar domains (see Section 7.2). Machine learning has been used for various tasks such as code and patch generation [2, 7, 11], program classification [30], and learning loop invariants [15].

8.3 Machine Learning for Proofs

While machine learning has previously been explored for various aspects of proof writing, we believe there are still significant opportunities for improving on the state-of-the-art, getting closer and closer to making foundational verification broadly applicable.

More concretely, work on machine learning for proofs includes: using machine learning to speed up automated solvers [4], developing data sets [5, 21, 38], doing premise selection [1, 28], pattern recognition [24], clustering proof data [23], learning from synthetic data [20], interactively suggesting tactics [19, 23].

Finally, CoqGym attempts to model proofs with a fully general proof command and term model expressing arbitrary AST's. We experimentally compare Proverbot9001's ability to complete proofs to that of CoqGym in detail in Section 7.2 There are also several important conceptual differences. *First*, the argument model in CoqGym is not as expressive as the one in Proverbot9001. CoqGym's argument model can predict a hypothesis name, a number between 1 and 4 (which many tactics in Coq interpret as referring to binders, for example `induction 2` performs induction on the second quantified variable), or a random (not predicted using machine learning) quantified variable in the goal. In contrast, the argument model in Proverbot9001 can predict any token in the goal, which subsumes the numbers and the quantified variables that CoqGym can predict. Most importantly because Proverbot9001's model can predict symbols in the goal, which allows effective unfolding, for example "`unfold eq`". *Second*, in contrast to CoqGym, Proverbot9001 uses several hand-tuned features for predicting proof commands. One key example is the previous tactic, which CoqGym does not even encode as part of the context. *Third*, CoqGym's treatment of higher-order proof commands like "`;`" is not as effective as Proverbot9001's. While neither system can predict "`;`", Proverbot9001 learns from "`;`" by linearizing them, whereas CoqGym does not.

There is also a recent line of work on doing end-to-end proofs in Isabelle/HOL and HOL4 [5, 16, 32]. This work is hard to experimentally compare to ours, since they use different benchmark sets, proof styles, and proof languages. Their most recent work [32] uses graph representations of terms, which is a technique that we have not yet used, and could adapt if proven successful.

Finally, there is also another approach to proof generation, which is to generate the term directly using language translation models [35], instead of using tactics; however this technique has only been applied to small proofs due to it's direct generation of low-level proof term syntax.

References

[1] Alexander A. Alemi, François Chollet, Geoffrey Irving, Christian Szegedy, and Josef Urban. 2016. DeepMath - Deep Sequence Models for Premise Selection. *CoRR* abs/1606.04442 (2016). arXiv:1606.04442 http://arxiv.org/abs/1606.04442

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *CoRR* abs/1709.06182 (2017). arXiv:1709.06182 http://arxiv.org/abs/1709.06182

[3] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 7 (April 2015), 31 pages. https://doi.org/10.1145/2701415

[4] Mislav Balunoviundefined, Pavol Bielik, and Martin Vechev. 2018. Learning to Solve SMT Formulas. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 10338–10349.

[5] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher-Order Theorem Proving (extended version). *CoRR* abs/1904.03241 (2019). arXiv:1904.03241 http://arxiv.org/abs/1904.03241

[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 171–177. http://dl.acm.org/citation.cfm?id=2032305.2032319

[7] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[8] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 270–286. https://doi.org/10.1145/3132747.3132776

[9] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

[10] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[11] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *CoRR* abs/1608.02715 (2016). arXiv:1608.02715 http://arxiv.org/abs/1608.02715

[12] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[13] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. 1997. *The Coq Proof Assistant - Reference Manual Version 6.1.* Technical Report.

[14] Jonathan Frankle, Peter-Michael Osera, David Walker, and S Zdancewic. 2016. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices* 51 (01 2016), 802–815. https://doi.org/10.1145/2914770.2837629

[15] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. *SIGPLAN Not.* 51, 1 (Jan. 2016), 499–512. https://doi.org/10.1145/2914770.2837664

[16] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. 2017. TacticToe: Learning to Reason with HOL4 Tactics. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Thomas Eiter and David Sands (Eds.), Vol. 46. EasyChair, 125–143. https://doi.org/10.29007/ntlb

[17] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *PPDP '10 Hagenberg, Austria* (ppdp '10 hagenberg, austria ed.). https://www.microsoft.com/en-us/research/publication/dimensions-program-synthesis/

[18] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion using Types and Weights. *PLDI 2013* (2013), 12. 27–38. http://infoscience.epfl.ch/record/188990

[19] Jónathan Heras and Ekaterina Komendantskaya. 2014. ACL2(ml): Machine-Learning for ACL2. In *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014*. 61–75. https://doi.org/10.4204/EPTCS.152.5

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

[21] Cezary Kaliszyk, François Chollet, and Christian Szegedy. 2017. HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving. *CoRR* abs/1703.00426 (2017). arXiv:1703.00426 http://arxiv.org/abs/1703.00426

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[23] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. 2012. Machine Learning in Proof General: Interfacing Interfaces. *Electronic Proceedings in Theoretical Computer Science* 118 (12 2012). https://doi.org/10.4204/EPTCS.118.2

[24] Ekaterina Komendantskaya and Kacper Lichota. 2012. Neural Networks for Proof-Pattern Recognition, Vol. 7553. 427–434. https://doi.org/10.1007/978-3-642-33266-1_53

[25] Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire, Vol. 8044. 1–35. https://doi.org/10.1007/978-3-642-39799-8_1

[26] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. http://xavierleroy.org/publi/compcert-CACM.pdf

[27] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. 727–739. https://doi.org/10.1145/3106237.3106253

[28] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep Network Guided Proof Search. *CoRR* abs/1701.06972 (2017). arXiv:1701.06972 http://arxiv.org/abs/1701.06972

[29] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/1706299.1706329

[30] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR* abs/1409.5718 (2014). arXiv:1409.5718 http://arxiv.org/abs/1409.5718

[31] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630. https://doi.org/10.1145/2813885.2738007

[32] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2019. Graph Representations for Higher-Order Logic and Theorem Proving. *CoRR* abs/1905.10006 (2019). arXiv:1905.10006 http://arxiv.org/abs/1905.10006

[33] Lawrence C. Paulson. 1993. Natural Deduction as Higher-Order Resolution. *CoRR* cs.LO/9301104 (1993). http://arxiv.org/abs/cs.LO/9301104

[34] Stephan Schulz. 2013. System Description: E 1.8. In *Proc. of the 19th LPAR, Stellenbosch (LNCS)*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.), Vol. 8312. Springer.

[35] Taro Sekiyama, Akifumi Imanishi, and Kohei Suenaga. 2017. Towards Proof Synthesis Guided by Neural Machine Translation for Intuitionistic Propositional Logic. *CoRR* abs/1706.06462 (2017). arXiv:1706.06462 http://arxiv.org/abs/1706.06462

[36] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

[37] 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 *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

[38] Kaiyu Yang and Jia Deng. 2019. Learning to Prove Theorems via Interacting with Proof Assistants. *CoRR* abs/1905.09381 (2019). arXiv:1905.09381 http://arxiv.org/abs/1905.09381

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *PLDI* (2011).

9 Appendix: Additional Evaluation

We now explore more detailed measurements about proof production.

9.1 Argument Accuracy

Our argument prediction model is crucial to the success of our system, and forms one of the main contributions of our work. To measure it's efficacy at improving search is hard, because it's impossible to separate it's success in progressing a proof from the success of the tactic predictor. However, we can measure how it contributes to individual prediction accuracy.

On our test dataset, where we can predict the full proof command in the original proof correctly 28.66% of the time, we predict the tactic correctly but the argument wrong 32.24% of the time. Put another way, when we successfully predict the tactic, we can predict the argument successfully with 89% accuracy. If we only test on proof commands within Proverbot9001's prediction domain, where we correctly predict the entire proof command 39.25% of the time, we predict the name correctly 41.01% of the time; that is, our argument accuracy is 96% when we get the tactic right. It's important to note, however, that many common tactics don't take any arguments, and thus predicting their arguments is trivial.

9.2 Completion Rate in Proverbot9001's Prediction Domain

Proverbot9001 has a restricted model of proof commands: it only captures proof commands with a single argument that is a hypothesis identifier or a token in the goal. As result, it makes sense to consider Proverbot9001 within the context of proofs that were originally solved with these types of proof commands. We will call proofs that were originally solved using these types of proof commands *proofs that are in Proverbot9001's prediction domain*. There are 79 such proofs in our test dataset (15.77% of the proofs in the test dataset), and Proverbot9001 was able to solve 48 of them.

What is interesting is that Proverbot9001 is able to solve proofs that are *not* in its prediction domain: these are proofs that were originally performed with proof commands that are *not* in Proverbot9001's domain, but Proverbot9001 found another proof of the theorem that *is* in its domain. This happened for 49 proofs (out of a total of 97 solved proofs). Sometimes this is because Proverbot9001 is able to find a simpler proof command which fills the exact role of a more complex one in the original proof; for instance, `destruct (find_symbol ge id)` in an original proof is replaced by `destruct find_symbol` in Proverbot9001's solution. Other times it is because Proverbot9001 finds a proof which takes an entirely different path than the original. In fact, 31 of Proverbot9001's 97 found solutions are shorter than the original. It's useful to note that while previous work had a more expressive proof command model, in practice it was unable to solve as many proofs as Proverbot9001 could in our more restricted model.

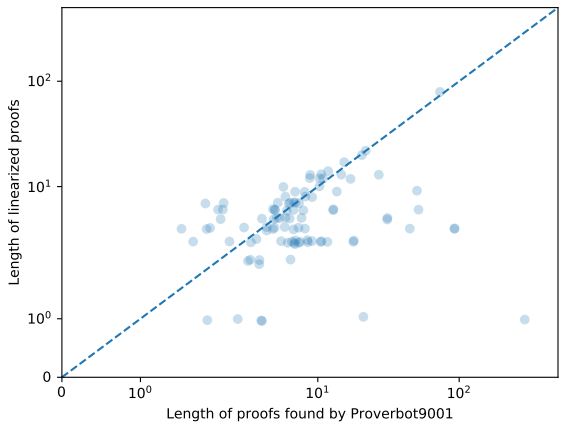

Figure 12. A comparison of the lengths of our found solution proofs and the lengths of their original solution proofs.

Together, these numbers indicate that the restricted tactic model used by Proverbot9001 does not inhibit it's ability to solve proofs in practice, even when the original proof solution used tactics outside of that model.

9.3 Original Proof Lengths vs Solution Lengths

In Figure 12, we compare, for proofs which Proverbot9001 was able to solve, the original (linearized) proof length and our solution proof length. Dots *above* the diagonal dashed line are cases where Proverbot9001's proof is shorter than the original proof (31 out of 97 proofs); dots *below* the diagonal dashed line are cases where Proverbot9001's proof is longer than the original proof (53 out of 97 proofs); dots *on* the diagonal dashed line are cases where Proverbot9001's proof is the same length as the original proof (13 out of 97 proofs);

While it is unsurprising that for many proofs our solution is longer, the fact that for 31 proofs our solution was shorter is unexpected. Since our proof command model forces us into more primitive tactics than those used in the original solutions, one would think that it should take us at least as many proof commands to solve the same propositions. However, since Proverbot9001 searches a large space for a solution proof, it can often find correct sequences of proof commands that are not apparent to human proof engineers.

9.4 Data Transformation

Crucial to Proverbot9001's performance is its ability to learn from data which is not initially in its proof command model, but can be transformed into data which is. This includes desugaring tacticals like `now`, splitting up multi-argument tactics like `unfold a, b` into single argument ones, and rearranging proofs with semicolons into linear series of proof

commands. To evaluate how much this data transformation contributes to the overall performance of Proverbot9001, we disabled it, and instead filtered the proof commands in the dataset which did not fit into our proof command model.

With data transformation disabled, and the default search width (5) and depth (6), the proof completion accuracy of Proverbot9001 is 15.57% (78/501 proofs). Recall that with data transformation enabled as usual, this accuracy is 19.36%. This shows that the end-to-end performance of Proverbot9001 benefits greatly from the transformation of input data, although it still outperforms prior work (CoqGym) without it.

When we measure the individual prediction accuracy of our model, trained without data transformation, we see that its performance significantly decreases (16.32% instead of 26.77%), demonstrating that the extra data produced by preprocessing is crucial to training a good tactic predictor.

9.5 Search Widths and Depths

Our search procedure has two main parameters, a *search width*, and a *search depth*. The *search width* is how many predictions are explored at each context. The *search depth* is the longest path from the root a single proof obligation state can have.

To explore the space of possible depths and widths, we varied the depth and width, on our default configuration without external tooling. With a search width of 1 (no search, just running the first prediction), and a depth of 6, we can solve 5.59% (28/501) of proofs in our test dataset. With a search width of 2, and a depth of 6, we're able to solve 16.17% (81/501) of proofs, as opposed to a width of 3 and depth of 6, where we can solve 19.36% of proofs.

To explore variations in depth, we set the width at 3, and varied depth. With a depth of 2, we were able to solve 5.19% (26/501) of the proofs in our test set. By increasing the depth to 4, we were able to solve 13.97% (70/501) of the proofs in our test set. At a depth of 6 (our default), that amount goes up to 19.36% (97/501).