

Generating Correctness Proofs with Neural Networks

ALEX SANCHEZ-STERN, UC San Diego

YOUSEF ALHESSI, UC San Diego

LAWRENCE SAUL, UC San Diego

SORIN LERNER, UC San Diego

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 was previously manual proofs, automatically solving 15.77% of proofs in our test dataset. This corresponds to an over 3X improvement over the prior state of the art machine learning technique for generating proofs in Coq.

Additional Key Words and Phrases: 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 [Filliâtre et al. 1997] or Isabelle HOL [Paulson 1993], to state and prove properties about their programs. The proofs are performed interactively via the use of *proof commands*, which are commands that programmers invoke to make progress on a proof. To complete a proof, a programmer must provide guidance to the proof assistant at each step by picking which proof command to apply. 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 [Leroy 2009], operating systems [Klein et al. 2009], database systems [Malecha et al. 2010], file systems [Chen et al. 2017], distributed systems [Wilcox et al. 2015], and cryptographic primitives [Appel 2015].

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 generated proofs can be checked with a small proof checker. In this setting only the small proof checker needs to be trusted (as opposed to the entire proof assistant), leading to a small trusted computing base. As an example of the high-level of assurance, a study of compilers [Yang et al. 2011] has shown that CompCert [Leroy 2009], 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 [Leroy 2009]

Authors' addresses: Alex Sanchez-Stern, UC San Diego, alexss@eng.ucsd.edu; Yousef Alhessi, UC San Diego, yousef@eng.ucsd.edu; Lawrence Saul, UC San Diego, saul@cs.ucsd.edu; Sorin Lerner, UC San Diego, lerner@cs.ucsd.edu.

2018. 2475-1421/2018/1-ART1 \$15.00

<https://doi.org/>

took 6 person years and 100,000 lines of Coq to write and verify, and seL4 [Klein et al. 2009], which is a verified version of a 10,000 line operating system, took 22 person years to verify. The manual effort is one of the main impediments to the broader adoption of proof assistants.

In this paper, we present Proverbot9001, a novel search-based 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. In contrast to previous work on machine learning for proofs, our approach (1) employs several neural networks, each with a narrow task (2) makes use of a small number of carefully crafted features (3) employs several tree pruning techniques in the 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. On our default configuration, Proverbot9001 solves 15.77% (79/501) of the proofs in our test set. This includes almost half of proofs where the original solution was expressible in our proof command model, as well as 46 proofs where it was not, and one proof whose original solution was 25 proof commands long.

Our main contributions are:

- (1) A technique for creating prediction models which account for the high-level structure of the proof.
- (2) A search procedure that makes use of these models to search for proofs.
- (3) An implementation of the models and the search procedure for the Coq proof assistant.

We demonstrate that Proverbot9001 can solve 15.77% of the proofs in our test subset of CompCert when trained on the rest of CompCert, which is a 3X improvement over the previous state of the art system that attempts to the same task [Yang and Deng 2019].

2 BACKGROUND

2.1 Foundational Verification

Program verification is a well studied problem in the programming languages community. Most work in this field falls into one of two categories: solver-backed automated (or semi-automated) techniques, where a simple witness is checked by a complex procedure; and foundational logic based techniques, where a complex witness is checked by a simple procedure.

While research into solver-backed techniques has produced fully-automated tools in many domains, these approaches are generally incomplete, failing to prove some desirable propositions. When these procedures fail, it is often difficult or impossible for a user to complete the proof, requiring a deep knowledge of the automation. In contrast, foundational verification techniques require a heavy initial proof burden, but scale to any proposition without requiring a change in proof technique. However, the proof burden of foundational techniques can be prohibitive; CompCert, a large and well-known foundationally verified compiler, took 6 person-years of work to verify [Kästner et al. 2018], with other large verification projects sporting similar proof burdens.

2.2 Interactive Theorem Provers

Most foundational (and some solver-backed) verification is done in an *interactive* theorem prover. Interactive theorem provers allow the user to define proof goals alongside data and program definitions, and then prove those goals interactively, by entering commands which

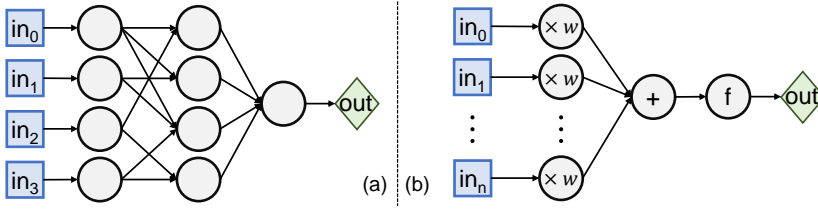


Fig. 1. (a) A feed-forward neural network, where each individual gray circle is a perceptron (b) An individual perceptron, which multiplies all the inputs by weights, sums up the results, and then applies a non-linear function f .

manipulate the proof context. The name and nature of these commands varies by the proof assistant, but in many foundational assistants, these commands are called “tactics”, and coorespond to primitive proof techniques like “induction”, as well as search procedures like “omega” (which searches for proofs over ring-like structures). Proof obligations in such proof assistants take the form of a set of hypotheses (in a Curry-Howard compatible proof theory, bound variables in a context), and a goal (a target type); proof contexts may consist of multiple proof obligations.

2.3 Machine Learning and Neural Networks

Machine learning is an area of computer science dating back to the 1950s. In problems of supervised learning, the goal is to learn a function from labeled examples of input-output pairs. Models for supervised learning parameterize a function from inputs to outputs and have a procedure to update the parameters from a data set of labeled examples. Machine learning has traditionally been applied to problems such as handwriting recognition, natural language processing, and recommendation systems.

Neural Networks are a particular class of learned model where layers of nodes are connected together by a linear combination and a non-linear activation function, to form general function approximators. Neural Networks have a variety of structures, some forming a straightforward “stack” of nodes with some connections removed (convolutional), and others, such as those used for natural language processing, using more complex structures like loops.

We will make use of two different kinds of neural networks: feed-forward networks and recurrent neural networks. Figure 1(a) shows the structure of a feed-forward network, where each gray circle is a perceptron, and Figure 1(b) shows individual structure of a perceptron.

Figure 2 shows the structure of a recurrent neural network (RNN). Inputs are shown in blue, outputs in green and computational nodes in gray. The computational nodes are Gated Recurrent Network nodes, GRU for short, which are a kind of node used in recurrent networks. The network is recurrent because it feeds back into itself, with the state output from the previous iteration feeding into the state input of the next iteration. When we display an RNN receiving data, we visually unfold the RNN, as shown on the right side of Figure 2, even though in practice there is still only one GRU node. The right side of Figure 2 shows an example RNN that processes tokens of a Coq goal, and produces some output values.

3 OVERVIEW

In this section, we’ll present Proverbot9001’s prediction and search process with an example from CompCert. You can see the toplevel structure of Proverbot9001 in Figure 3.

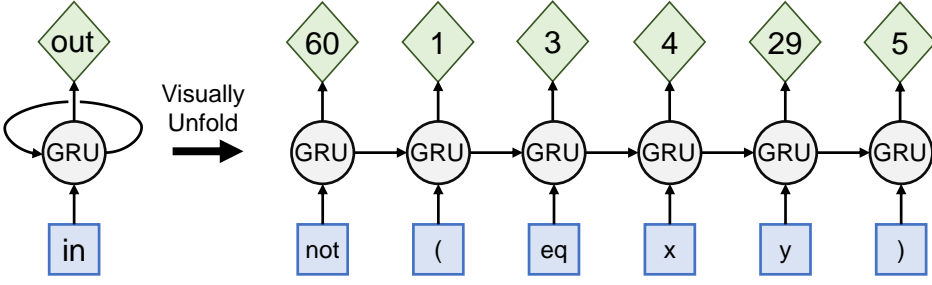


Fig. 2. A recurrent neural network

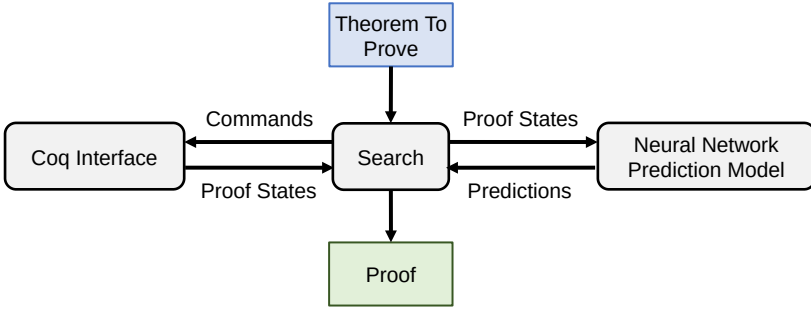


Fig. 3. The overall architecture of Proverbot9001. Each of the modules is written in Python, with the Coq interface as a Python interface, on top of Coq Serapi on top of Coq, and the neural prediction model built on top of PyTorch.

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, or in other words, that it produces an expression that yields a value, in any context, that is at least as defined as applying the semantic definition of mulhs (namely Val.mulhs) to the values produced by the subexpressions. The theorem statement is within a Coq “Section”, which automatically introduces some hypotheses. Therefore, after running the above theorem statement, the proof assistant produces the following proof context.

```
ge : genv
sp : val
e : env
```

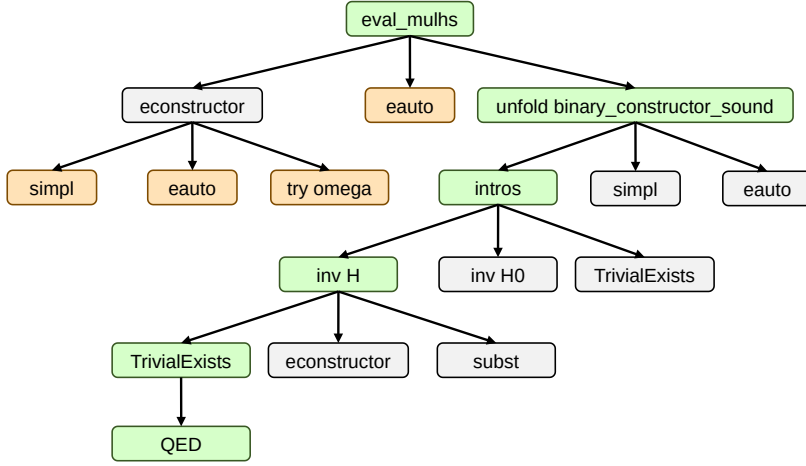


Fig. 4. A graph of a Proverbot9001 search. In green are the tactics that formed part of the discovered solution, as well as the lemma name and the QED. In orange are nodes that resulted in a context that is at least as hard as one previously found (see Section 6.1).

m : Mem.mem

binary_constructor_sound mulhs Val.mulhs

In this context, Proverbot9001 predicts three tactics, `econstructor`, `eauto`, and `unfold`. At this point, Proverbot9001 considers possible arguments to add to each tactic. In this model, arguments can be a hypothesis identifier, a token from the goal, or “none”. For the `econstructor` and `eauto` tactics, Proverbot9001 predicts a “none” argument, making the full proof commands `econstructor.` and `eauto.` For the `unfold` prediction, Proverbot9001 considers tokens a token from the goal as an argument, in particular `binary_constructor_sound`, producing a prediction of `unfold binary_constructor_sound`. In this case, Proverbot9001 predicts each tactic once, with one argument; however, in general it might predict the same tactic multiple times with different arguments (or different tactics with the same argument).

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 4 shows the search tree resulting from this process for this example. The nodes in green are the nodes that produce the final proof. Orange nodes are nodes that generate a prover state that is at least as hard as a previously found one (see Section 6.1), and so those nodes are not expanded further. All the gray 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
$C = \mathcal{T} \times \mathcal{A}$	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
$\mathcal{S} = [\mathcal{O} \times [C]]$	Proof states

Fig. 5. Formalism to model a Proof Assistant

4 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 5 shows the formalism we will use to represent the state of a proof assistant. 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.

5 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:

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

We define a τ -predictor $\mathcal{P}[\tau]$ 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:

$$\mathcal{P}[\tau] = \mathcal{S} \rightarrow \mathcal{D}[\tau]$$

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

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

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

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

$$P_{arg} : \mathcal{T} \rightarrow \mathcal{P}[\mathcal{A}]$$

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 .

5.1 Predicting Tactics (P_{tac})

To predict tactics, Proverbot9001 uses of a set of simple features. These features were manually engineered to reflect important aspects of proof prediction. Each of these features is encoded as a discrete integer. After encoding, we have a vector of discretely encoded features, which we can then run machine learning algorithms on.

The first feature encodes the goal, which is the proposition that is currently being proved. Proverbot9001 captures the goal by encoding the head constructor of the goal. Each head constructor is assigned a unique integer. This feature is clearly important because tactics are often chosen specifically based on what the goal is. For example, if the goal starts with a `forall`, it is likely that the next tactic will be `intros` or `induction`.

The second feature is the name of the previously run tactic. Proverbot9001 encodes each tactic as an integer. This feature is important because there are often patterns of tactics which appear one after another. For example, `intros` is often followed by `auto`, and `inversion` is often followed by `subst`.

The third feature encodes a hypothesis that is heuristically chosen as being the most relevant to the goal. In particular, Proverbot9001 pretty prints every hypothesis and the goal with notations turned off, and then computes a string similarity measure between each hypothesis and the goal. The hypothesis that is most similar to the goal is considered as being the most relevant to the goal. This hypothesis is encoded in the same way as the goal, using the head constructor. Proverbot9001 also takes the similarity score of this most relevant hypothesis as an additional feature. These features encoding the most relevant hypothesis is important because if there is a hypothesis that is very similar to the goal, there is a high likelihood that this hypothesis will be important in selecting the tactic. For example, if the most relevant hypothesis is an equality, it is likely that `rewrite` is the correct tactic. This heuristic is also computed at testing time.

Once the features are computed, they 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 of three layers 128 nodes wide with a softmax layer at the end, to compute a probability distribution across possible tactic. This architecture is trained on 153402 samples in our training data with a stochastic gradient descent optimizer.

The architecture of this model is shown in Figure 6. 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 standard word embedding module.

5.2 Predicting Tactic Arguments (P_{arg})

Once features have been extracted and a tactic predicted, Proverbot9001 next predicts arguments for the command. Recall that the argument predictor is a function $P_{arg} : \mathcal{P}[\mathcal{A}]$. In contrast to previous work, our argument model is a prediction architecture in it's 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.

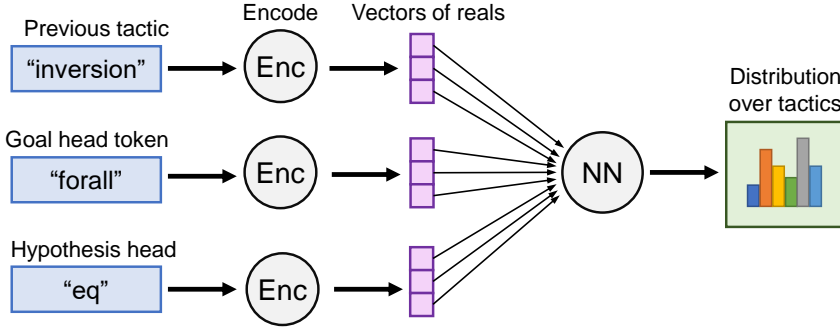


Fig. 6. Proverbot9001’s model for predicting tactic. Takes as input the previous tactic run and the head tokens of the goal and most relevant hypothesis (see Section 5.1). We restrict the previous tactic feature to the 50 most common tactics, and head tokens on goal and hypothesis to the 100 most common head tokens.

Proverbot9001 makes two kinds of predictions for arguments: *goal-token* arguments, and *hypothesis-identifier* 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-identifier arguments are a single identifier referring to a hypothesis in context. For instance, consider the following obligation:

```
...
H : is_path (cons (pair s d) m)
```

```
...
```

```
...
```

In this case, we might predict `inversion H`, where `H` refers to the hypothesis. In the case of tactics like `inversion` and `rewrite`, the argument is often a hypothesis identifier.

The architecture of the scoring functions for these argument types is shown in Figure 7. As before, blue boxes are inputs; purple boxes are encoded values; green diamonds are outputs, in this case scores for each individual possible argument; and gray circles are computational nodes. The GRU nodes are Gated Recurrent Units [Cho et al. 2014]. The NN node is a feed-forward neural network.

For illustration purposes, Figure 7 uses an example to provide sample values. Each token in the goal is an input – in Figure 7 the goal is `not (eq x y)`. The tactic predicted by P_{tac} is also an input – in Figure 7 this tactic is `unfold`. The hypothesis that is heuristically closest to the goal (according to our heuristic from Section 5.2) is also an input, one token at a time being fed to a GRU. In our example, let’s assume this closest hypothesis is `y > (x+1)`. The similarity score of this most relevant hypothesis is an additional input – in Figure 7 this score is 5.2.

There is an additional RNN (the middle row of GRUs in Figure 7) which encodes the goal as a vector of reals. The initial state of this RNN is set to some arbitrary constant, in this case 0.

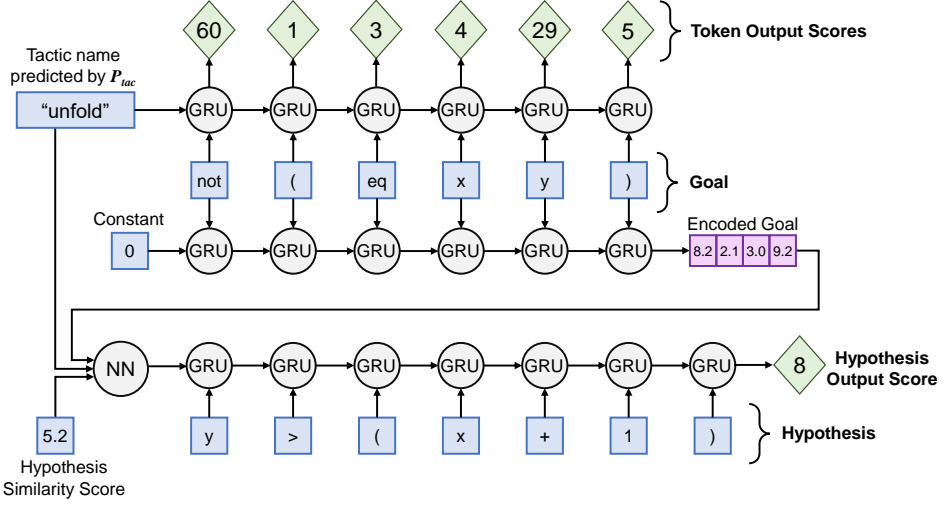


Fig. 7. The models for scoring possible arguments.

The initial state of the hypothesis RNN (the third row of GRUs in Figure 7) is computed using a feed-forward Neural Network (NN). This feed-forward Neural Network takes as input the tactic predicted by P_{tac} , the goal encoded as a vector of reals, and the similarity score of the hypothesis.

The architecture in Figure 7 produces one output score for each token in the goal and one output score for the hypothesis. The highest scoring element will be chosen as the argument to the tactic. In Figure 7, the highest scoring element is the “not” token, resulting in the proof command “unfold not”. If the hypothesis score (in our example this score is 8) would have been the highest score, then the chosen argument would be the identifier of that hypothesis in the Coq context. For example, if the identifier was `IHn` (as is sometimes the case for inductive hypotheses), then the resulting proof command would be “unfold `IHn`”.

5.3 Combining Tactic and Argument Scores (\otimes)

There are several options for the \otimes operator. We first discuss two options that have flaws before describing our approach.

First flawed approach: greedy search. In this approach, we would pick the tactic with the highest score, and for that tactic, we would pick the argument with the highest score. We could implement this approach by making \otimes give a very high weight to its first argument.

The drawback of this approach is that the final choice of tactic does not take into account how good the argument prediction is for that tactic. For example, the tactic name predictor might predict `rewrite`, only to find that there are no good hypotheses to rewrite with.

Second flawed approach: normalized search. In this approach, the scoring functions from P_{tac} and P_{arg} are first normalized to turn the scores into probabilities. Then, from P_{tac} we could pick the n most likely tactics, and for each tactic predict the m most likely arguments. For each proof command, we would multiply the tactic probability and the argument probability to get a combined probability. We could then pick the proof command with the highest combined probabilities. Formally, we could implement this approach by

first normalizing the scoring functions inside of P_{tac} and P_{arg} , and then defining \otimes as multiplication.

The drawback of this approach is that normalizing the scoring functions for arguments loses the relative scoring of arguments across tactics. For example, say that the two highest-scoring tactics are τ_1 and τ_2 and that P_{tac} scores these tactics with the same score. Also, for τ_1 , let's assume that there are three equally scoring arguments, but these three arguments have very *low* scores, meaning that in absolute terms they are bad predictions. And let's assume that for τ_2 , there are three equally scoring arguments, but for τ_2 the scores for the arguments are high. While in this case τ_2 would be a better choice, the normalized approach would not favor τ_2 . Indeed, all proof commands will have the same probability, because for both τ_1 and τ_2 the three arguments are all equally likely, and so the normalized distributions returned by $P_{arg}(\tau_1)$ and $P_{arg}(\tau_2)$ will essentially look the same. In essence, this approach would lose the fact that the argument predictions for τ_1 are much worse in absolute terms than the ones for τ_2 .

Our approach: combined goodness score. The approach we take does *not* normalize the scores from P_{tac} and P_{arg} . Instead, 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. This assumes that the argument scoring functions across different tactics can be compared, which is indeed the case because we train the argument predictor with different possible tactics for each input point (see Figure 10). 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. This approach provides a balanced combination of tactic and argument prediction, taking both into account, even across different tactics.

5.4 Putting it all together

The overall architecture that we have described is shown in Figure 8. The P_{tac} predictor (whose detailed structure is shown in Figure 6) 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 7) 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.

5.5 Training

We now describe how Proverbot9001 performs training. Figure 9 shows the training architecture for the tactic predictor, P_{tac} (recall that the detailed architecture of P_{tac} is shown in Figure 6). The goal of training is to find weights for the neural network that is found inside the gray P_{tac} circle. This is done through a stochastic gradient descent optimizer, with Negative Log Likelihood Loss (NLLLoss) as the criterion. Proverbot9001 processes all the Coq theorems in the training set, and steps through the proof of each of these theorems. Figure 9 shows what happens at each step in the proof. In particular, at each step in the proof, Proverbot9001 computes the three features we are training with, and passes these features to the current tactic model to get a distribution over tactics. This distribution over

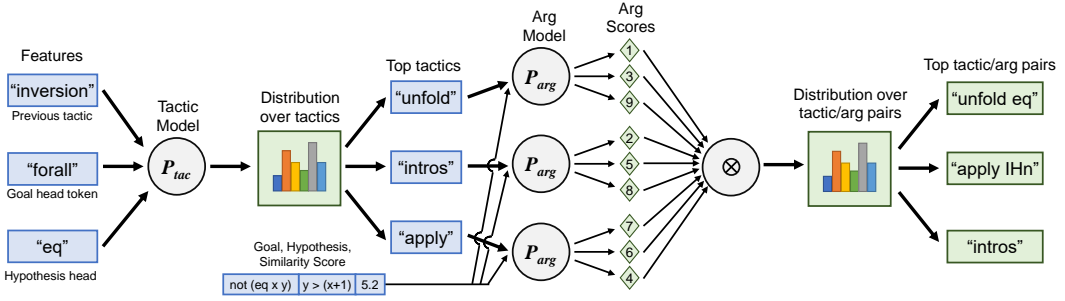


Fig. 8. The overall prediction model, combining the tactic prediction and argument prediction models.

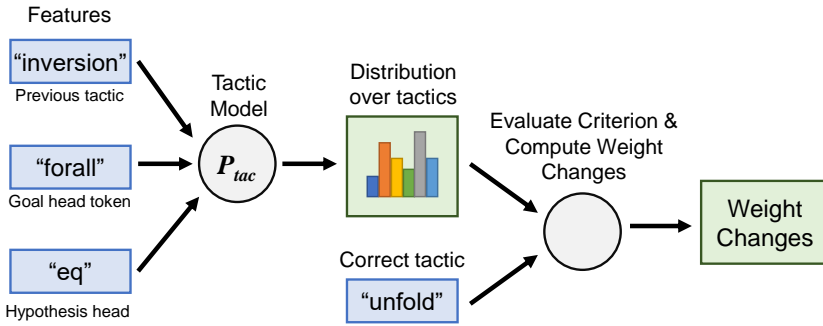


Fig. 9. The architecture for training the tactic models.

tactics, along with the correct tactic name (from the actual proof), are passed to a module that computes changes to the weights based on the NLLLoss criterion. These changes are batched together over several steps of the proof, and then applied to update the tactic model. Running over all the training data to update the weights is called an epoch, and we run our training over about 17 epochs.

Figure 10 shows the training architecture for the argument predictor, P_{arg} (recall that the detailed architecture of P_{arg} is shown in Figure 7). The goal of training is to find weights for the GRU components in P_{arg} . Here again, Proverbot9001 processes all the Coq theorems in the training set, and steps through the proof of each of these theorems. Figure 10 shows what happens at each step in the proof. In particular, at each step in the proof, the current P_{tac} predictor is run to produce the top predictions for tactic. These predicted tactic, along with the correct tactic, are passed to the argument model P_{arg} . To make Figure 10 more readable, we do not show the additional parameters to P_{arg} that were displayed in Figure 8, but these parameters are in fact also passed to P_{arg} during training. 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

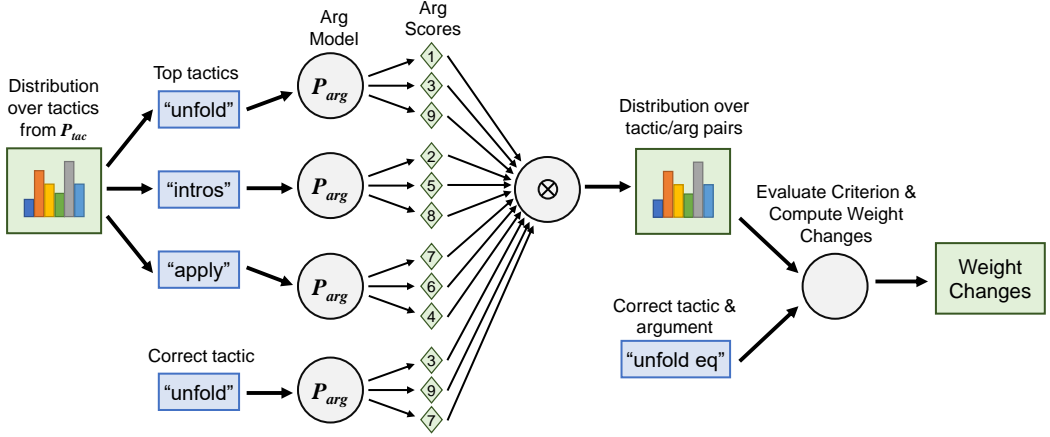


Fig. 10. The architecture for training the argument models. Note that we inject predicted tactics into the input of the argument model, instead of just using the correct tactic, so that argument scores will be comparable.

weights based on the NLLoss criterion. Here again, the changes to the weights are batched, and the 153402 tactic samples from the training set are processed for 17 epochs.

5.6 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*. One of the most common examples is the `;` infix operator which runs the proof command on the right on every sub-goal produced by the tactic on the left. Another example is the `repeat` tactical, which repeats a provided tactic until it fails.

While higher-order proof commands are extremely important for human proof engineers, they are harder to predict automatically because of their generality. Indeed, effectively predicting such higher-order tactics would require learning deeper contextual features.

While some previous work [Yang and Deng 2019] attempts to learn directly on data which uses these higher-order proof commands, we instead takes the approach of desugaring higher-order proof commands as much as possible. This makes the data more learnable, without restricting the set of expressible proofs.

For example, instead of trying to learn and predict `;` directly, Proverbot9001 has a system which attempts to desugar `;` into linear sequences of proof commands. This is not always possible (without using explicit subgoal switching commands), due to propagation of existential variables across proof branches. Proverbot9001 desugars the cases that can be sequenced, and the remaining commands containing `;` are filtered out of the training set.

In addition to the `;` tactical, there are other tacticals in common use in Coq. Some can be desugared into simpler forms. For example:

- “now <tac>” becomes “<tac>;easy”.
- “rewrite <term> by <tac>” becomes “rewrite <term> ; [| <tac>]”
- “assert <term> by <tac>” becomes “assert <term> ; [| <tac>]”

```

proof-search( $g : \mathcal{G}, f_0 : \mathbb{N}$ )
  worklist :  $[\mathbb{N} \times [C]] = \{(f_0, [])\}$ 
  while size(worklist) > 0 :
    ( $f, h$ ) :  $\mathbb{N} \times [C] = \text{pop}(\text{worklist})$ 
     $\Sigma : [S] = \text{run-commands}(g, h)$ 
    if (done( $\Sigma$ )) :
      return Some( $h$ )
    elif (crashed( $\Sigma$ ) or any( $\Sigma[0] > \sigma$  for  $\sigma$  in  $\Sigma[1:]$ )) :
      continue
    else :
      oo : Option ( $O \times [C]$ ) = finished-obligations( $\Sigma$ )
      match oo :
        ||Some( $o, l$ ) →
          worklist = prune-nodes-in-obligation(worklist,  $o$ )
           $f' = f - 1 + \text{size}(l)$ 
        ||None →
           $f' = f - 1$ 
      if ( $f' > 0$ ) :
        for  $p$  in  $P(\Sigma[0])$  :
          worklist = push(worklist, ( $f', h + [p]$ ))
  return None

```

Fig. 11. The core search algorithm of Proverbot9001. Takes a goal (theorem statement) and amount of fuel, and attempts to return a proof command solution S that completes the proof. For simplicity, details of running to particular proof states, checking for repeated contexts, managing subgoal state, and pruning the search tree have been elided here. For details, see Section 6.

In other cases, like `try <tac>` or `solve <tac>`, the tactical changes the behavior of the proof command in a way that cannot be desugared; for these we simply treat the prefixed tactic as a separate, learned tactic. For example, we would treat `try eauto` as a new tactic.

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.

We can define proof search in an interactive proof assistant as the problem of finding a list of proof commands, which when applied to a given theorem will produce the terminal state, in which no more obligations are left to process. 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. Proof search in proof assistants is very hard because the number of possible proof commands to apply is large, and so the fan-out of the search tree is very large. To address this problem, we will use the predictions from Section 5 to guide the search.

The core search algorithm of Proverbot9001 is outlined in 11. The search algorithm takes a goal (theorem statement) and an integer representing the amount of fuel left. Fuel is meant to capture the amount of depth still available to explore. Fuel generally decreases by each time a tactic is applied.

We describe below several of the key functions used in the algorithm:

- `run-commands(g, h)` takes a goal g and a list of proof commands h and runs the proof commands starting from the given goal, with no hypotheses. It returns a list of proof states that Coq went through.
- `done(Σ)` takes a list Σ of proof states, and returns true if the proof is complete (meaning that the last element of Σ has no more obligations, and corresponds to “QED” in Coq).
- `crashed(Σ)` takes a list Σ of proof states, and returns true if the proof stopped because of a failed proof command application.
- $\Sigma[0]$ refers to the 0th element of the list of proof states Σ .
- $\Sigma[1:]$ refers to the list of proof states containing all proof states from Σ , except for the first one.
- $<$ is an operator on states that is used for pruning loops in the proof search and is described in Section 6.1.
- `finished-obligations(Σ)` takes a list Σ of proof states and returns whether an obligation (“subgoal” in Coq terminology) just discharged in the proof, and if so returns the obligation o and the list of proof commands l used to discharge the obligation.
- `prune-nodes-in-obligation(worklist, o)` takes a worklist and an obligation that has just been proved, and returns a new worklist without any open proof states related to that obligation. This pruning is described in more detail in Section 6.2.
- `size(l)` returns the size of list l . We use `size` to perform a kind of fuel refunding that is described in more detail in Section 6.3.
- The prediction function P was described in Section 5.

There are three remaining components that require additional explanation:

- Pruning the search tree based on reproducing earlier states (or states strictly “worse”)
- Pruning the search tree based on proof irrelevance of subgoals
- Refunding fuel based on subgoal solution

We discuss each of these in the next three subsections.

6.1 Reproduced states

During search, sometimes running a proof command prediction reproduces an earlier state in that predictions history.

The simplest case of this is when a proof command crashes with an error message, such as when we run the ring-solver `omega` on a goal not involving ring equations. In this case, the proof state does not change, and we can close off that branch of the search, since we cannot run any proof commands from the error state. Other proof commands however, may fail to change the proof state without producing an error message; for instance if we run the simplifying tactic `simpl` on an goal which it cannot simplify any further. In this case, we can determine that the resulting proof state is the same as the initial one, and close off that branch of the search as well.

Even when a proof command changes the proof state, it can end up reproducing a proof state from earlier in the history. For instance, one might introduce a hypothesis and then immediately revert it, or rewrite by an equation and then rewrite by its inverse. Even though each proof command individually changes the proof state, together they result in a repeated state. For this reason we check whether the proof state is in any part of the history to determine whether to prune its descendants.

Finally, a proof command might produce a new proof state which is harder to prove than the initial one. In this case, we can prune the tree rooted at that proof command application. While in general it is hard to formally define what makes a proof state harder than another,

there are some obvious cases which we can detect. For example, a proof state with a superset of the original obligations will be harder to prove. Also, a proof state with the same goal, but fewer assumptions, will be harder to prove.

To formalize this intuition, we define a relation $>$ between states such that $\sigma_1 > \sigma_2$ is meant to capture “Proof state σ_1 is definitely harder than proof state σ_2 ”. We say that $\sigma_1 > \sigma_2$ if and only if for all obligations O_2 in σ_2 there exists an obligation O_1 in σ_1 such that $O_1 >_o O_2$. For obligations O_1 and O_2 , we say that $O_1 >_o O_2$ 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.

We generalize the pruning criteria above to “proof command prediction produces a proof state which is $>$ than a proof state in the history”.

6.2 Proof irrelevance of obligations

During search, it is often the case that a proof command, while not solving all the obligations of the proof state, will solve some of those obligations. In a simple exhaustive backtracking search, if we fail to find a proof after completing an obligation, we might backtrack to the solved obligation, and attempt to find another way to solve it.

However, in Proverbot9001, we assume that obligations are *proof irrelevant*, which means that any proof of an obligation is as good as any other. For this reason, once we’ve found the solution to an obligation, we prune all other search nodes that are meant to prove the same obligation..

This may not always be the right thing to do, since existential variables can propagate across different obligations, and thus there are some proofs of obligations which **are** better than others, in that they make other obligations easier to solve. However, since these cases are rare, and this pruning allows us to search much deeper for a proof, the trade-off is beneficial.

6.3 Refunding fuel

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).

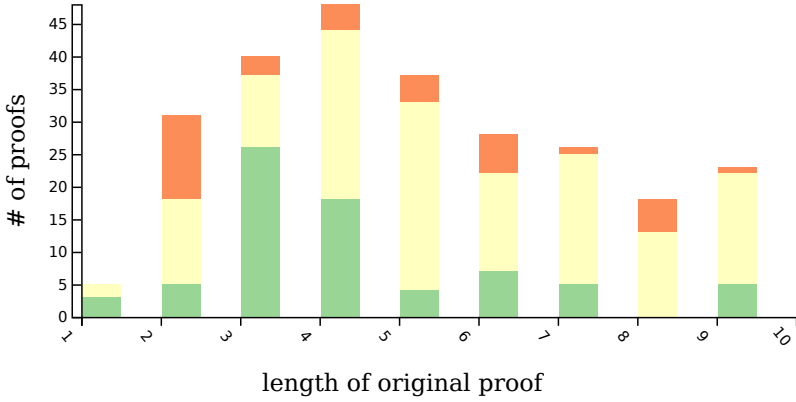


Fig. 12. A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes, for proofs with length 10 or less. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted.

There are many ways of implementing a search that takes the sub-proof structure into account. We achieve this in Proverbot9001 this by decrementing a “fuel” variable each time a proof command is applied and then “refunding” fuel upon completion of an obligation within a proof; the amount of fuel refunded is the number of tactic steps that it took to solve the obligation. In the example above with induction n , after `eauto` discharges the base case of the induction, Proverbot9001 would refund 1 fuel, because that base case obligation took 1 step to discharge.

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 7.1, we’ll describe Proverbot9001’s ability to produce correct proofs of lemmas in the CompCert verified C compiler. Then, we’ll compare experimentally to previous work, by running the CoqGym [Yang and Deng 2019] project on CompCert, in several configurations outlined in their paper. Finally, we’ll describe experiments run to test the effectiveness of various subsystems of Proverbot9001.

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 Tesla v100 16GB 4096-bit, and 61 gigabytes of memory. All experiments were run on the CompCert verified C compiler, on the tag 8.9.1, using Coq 8.9.1, and a compatible version of Coq Serapi. Experiment running uses GNU Parallel [Tange 2011].

7.1 Proof Production

We tested Proverbot9001 end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 files. On our default configuration, Proverbot9001 solves 15.77% (79/501) of the proofs in our test set.

7.1.1 Original Proof Length vs Completion Rate. In Figure 12 and Figure 13, we plot a histogram of the original proof lengths (in proof commands) vs the number of proofs of

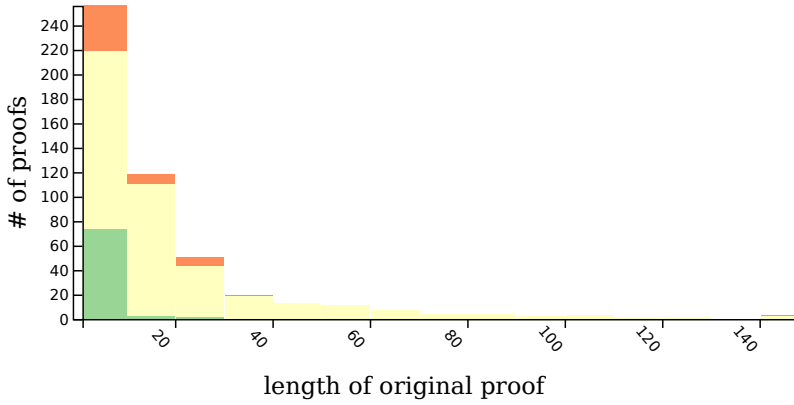


Fig. 13. A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted. Note that most proofs are between 0 and 10 proof commands long, with a long tail of much longer proofs.

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 12 shows proofs of length 10 or below, and Figure 13 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 28% (72/256), compared to our overall solve rate of 15.77% (79/501). *Third*, we are in some cases able to handle proofs whose original length is longer than 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.

7.1.2 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 solved using these types of proof commands *proofs that are in Proverbot9001's prediction domain*. There are 59 such proofs in our test dataset (11.78% of the proofs in the test dataset). Of these 59, Proverbot9001 was able to solve 33, which means Proverbot9001 can solve 55.93% of the proofs that are in its prediction domain.

What is also 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 46 proofs (out of a total of 79 proofs). Sometimes this

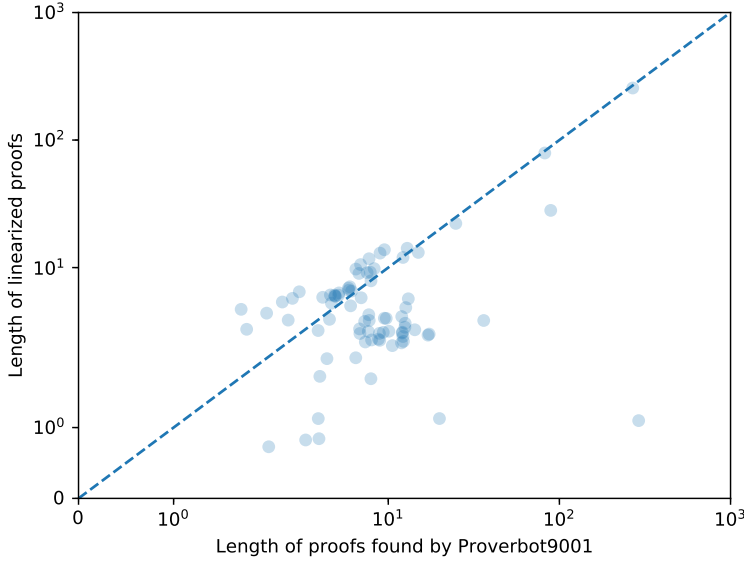


Fig. 14. A comparison of the lengths of our found solution proofs and the lengths of their original solution proofs.

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, 27 of Proverbot9001’s 79 found solutions are shorter than the original. It’s useful to note that while previous work had a more expressive proof command model, that could in theory solve every proof, in practice it was unable to solve as many proofs as Proverbot9001 could in our more restricted model.

7.1.3 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 made (and tried) 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 first fixed the depth at 4 and varied width. With a search width of 3, and a depth of 4, we’re only able to solve 7.58% (38/501) of proofs, as opposed to a width of 5 and depth of 4, where we can solve 8.98% of proofs. When we increase our search width to 7, and keep a depth of 4, we only solve one additional proof, bringing total solved to 9.18%. Our default width is 5.

To explore variations in depth, we also the width at 5, and varied depth. With a depth of 2, we were able to solve 2.40% (12/501) of the proofs in our test set. By increasing the depth to 4, we were able to solve 8.98% (45/501) of the proofs in our test set. At a depth of 6 (our default), that amount goes up to 15.77% (79/501).

7.1.4 Original Proof Lengths vs Solution Lengths. In Figure 14, 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 (27 out of 79 proofs); dots *below* the diagonal dashed line are cases where Proverbot9001's proof is longer than the original proof (48 out of 79 proofs); dots *on* the diagonal dashed line are cases where Proverbot9001's proof is the same length as the original proof (4 out of 79 proofs);

While it is unsurprising that for many proofs our solution is longer, for 27 proofs our solution being 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.

7.2 Experimental Comparison to Previous Work

In addition to running Proverbot9001 on CompCert, we ran the CoqGym [Yang and Deng 2019] tool, which represents the state of the art in this area, on the same dataset in several configurations.

There were several differences in the experimental setup of CoqGym that we needed to account for in our experiments. First, CoqGym trained on a larger dataset of multiple proof projects. It was unclear to us whether that would hurt or help their performance on a single proof project, because it gained them additional data, but also resulted in less understanding of project specific proof techniques. To account for this, we ran CoqGym with their original training schema, and also our training schema, and reported the best of the two numbers.

Second, CoqGym is intended to be combined with a solver based proof-procedure, CoqHammer [Czajka and Kaliszyk 2018], which is run after every proof command invocation. While our system was not intended to be used in this way, we compare both systems using CoqHammer, as well as both systems without.

It is important to realize that when we add CoqHammer to a system, be it CoqGym or Proverbot9001, CoqHammer is run after every single proof command invocation, not just once.

Figure 15 shows the proofs solved by various configurations. The configurations are described in the caption. Note that in Figure 15 the bars for H, C, and C_H are prior work. The bars P, C+P and C_H+P_H are the ones made possible by our work.

When CoqHammer is not used, Proverbot9001 can complete more than 3 times the number of proofs that are completed by CoqGym. In fact, even when CoqGym is augmented with CoqHammer (which simultaneously invokes Z3 [de Moura and Björner 2008], CVC4 [Barrett et al. 2011], Vampire [Kovács and Voronkov 2013], and E Prover [Schulz 2013], after every predicted proof command) Proverbot9001 by itself (without CoqHammer) still completes 21 more proofs than CoqGym with CoqHammer, which is a 36% improvement (and corresponds to about 4% of the test set). When enabling CoqHammer in both CoqGym and Proverbot9001, we see that CoqGym solves 58 proofs whereas Proverbot9001 solves 109 proofs, which is a 88% 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 88/501 proofs, a proof success rate of 17.5%. Combining Proverbot9001 and CoqGym, each with CoqHammer, allows us to solve 131/501 proofs, a success rate of 26%. It's important to realize that, whereas the prior state of the art was CoqGym with CoqHammer, at 58 proofs, by combining CoqGym and Proverbot9001 (both with

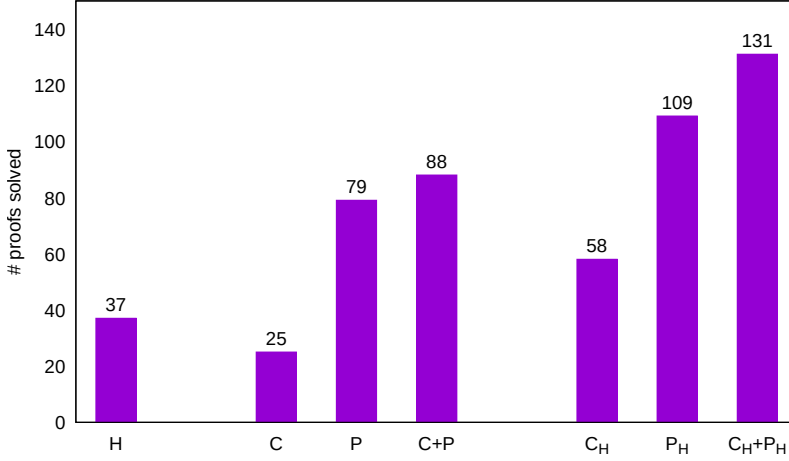


Fig. 15. A comparison of Proverb9001 and CoqGym’s abilities to complete proofs. H stands for CoqHammer by itself, as a single invocation; C stands for CoqGym by itself; P stands for Proverb9001 by itself; C+P stands for the union of proofs done by C or P; C_H stands for CoqGym with CoqHammer; P_H stands for Proverb9001 with CoqHammer; C_H+P_H stands for the union of proofs done by C_H or P_H.

CoqHammer), we can reach a grand total of 131 proofs, which is a 2.2X improvement over the prior state of art.

7.3 Subsystems

Finally, we measured the various subsystems of Proverb9001 to determine how much each one contributes to it’s success.

7.3.1 Individual Prediction Accuracy. We want to measure the effectiveness of the predictor subsystem that predicts proof commands pairs (the P function defined in Section 5). 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 26.7% (3703/13867) accurately. This includes the correct tactic and the correct argument. If we only test on the proof commands which are in Proverb9001’s prediction domain, we are able to predict 38.83% (3703/9537) 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 43.03% of the time. If we restrict to proof commands in Proverb9001’s prediction domain, those numbers are 56.6% and 62.57%.

7.3.2 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 26.7% of the time, we predict the tactic correctly but the argument wrong 30.33% of the time. Put another way, when we successfully predict the tactic, we can predict the argument successfully with 88% accuracy. If we only test on proof commands within Proverbot9001's prediction domain, where we correctly predict the entire proof command 38.83% of the time, we predict the name correctly 40.99% of the time; that's a 94% argument accuracy when we get the tactic right. It's important to note, however, that many common tactics don't take any arguments, and thus we can easily predict them fully.

7.3.3 Data Transformation. Crucial to Proverbot9001's performance is it's 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 8.18% (41/501 proofs). Recall that with data transformation enabled as usual, this accuracy is 15.77%. This shows that the end-to-end performance of Proverbot9001 benefits greatly from the transformation of input data, although it still outperforms prior work (CogGym) without it.

When we measure the individual prediction accuracy of our model, trained without data transformation, on only proof commands within Proverbot9001's domain, we see that it's performance does not significantly decrease (36.96% instead of 38.83%). Nevertheless, the kinds of mistakes that our model makes when trained in this way are significantly worse for the end-to-end performance of the model.

8 RELATED WORK

8.1 Program Synthesis

Program Synthesis is the automatic generation of programs from a high-level specification [Gulwani 2010]. 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 [Gvero et al. 2013], while other work uses both a type and a set of examples to synthesize programs [Frankle et al. 2016; Osera and Zdancewic 2015]. 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 [Long et al. 2017] 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 [Allamanis et al. 2017], as an alternative to more structured methods of modeling code. Several models have been proposed for learning code, such as AST-like trees [Mou et al. 2014], long-term language models [Dam et al. 2016], and probabilistic grammars [Bielik et al. 2016]. Proverbot9001 does not attempt to be so general, using a model of programs that is specific to the domain, and allows 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 [Allamanis et al. 2017; Bielik et al. 2016; Dam et al. 2016], program classification [Mou et al. 2014], and learning loop invariants [Garg et al. 2016].

8.3 Theorem Proving Sub-Tasks using Machine Learning

Using machine learning for various theorem proving tasks has been explored with a variety of models [Kaliszyk et al. 2017; Komendantskaya et al. 2012]. The most basic is using machine learning to prune a database of theorems/premises, for use by an solver-based theorem prover [Alemi et al. 2016; Loos et al. 2017].

Work has also been done in attempting to classify proofs as correct or incorrect, recognizing the anti-unification of sets of proof trees [Komendantskaya and Lichota 2012], suggesting common tactics to the user during interaction, and generating auxiliary lemmas [Heras and Komendantskaya 2014].

In contrast to this work, the goal of Proverbot9001 is to synthesize full proofs of propositions in the context of a larger project.

8.4 Automatically Generating Tactic-based Proofs with Machine Learning

The most closely related work to Proverbot9001 is that which attempts to automatically generate tactic-based proofs, given a proposition, using machine learning.

ML4PG [Komendantskaya et al. 2012], Gamepad [Huang et al. 2018], and CoqGym [Yang and Deng 2019], all introduce benchmark suites and frameworks for exploring machine learning in Coq. ML4PG, while it introduces machinery which should allow it to generate proofs, focuses instead on clustering proofs, and does not attempt to generate proofs. Gamepad considers the problem of predicting proofs, but does not combine their proof command predictor with a high-level search, and trains mainly on synthetic data of simple proofs. Where they do run on non-synthetic proofs, they only attempt to match proof commands in the original proof, and improve only modestly over a constant baseline predictor.

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 CogGym, Proverbot9001 uses several hand-tuned features for predicting proof commands. One key example is the previous tactic, which CogGym 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.

Similar work has been done in Isabelle HOL [Gauthier et al. 2017] and HOL light [Bansal et al. 2019]. These works would be hard to compare experimentally to Proverbot9001, since they necessitate different benchmark sets and proof styles; however, their proof completion performance and techniques are roughly equivalent to those of CoqGym. Basic proof term generation has also been modeled using language translation models [Sekiyama et al. 2017]; however it has only been applied to small proofs due to its direct generation of proof term syntax.

ACKNOWLEDGMENTS

We would like to thank Joseph Redmon for his invaluable help building the first Proverbot9001 version, Proverbot9000.

REFERENCES

- 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>
- 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>
- 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>
- 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>
- 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>
- 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>
- 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>
- Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 <http://arxiv.org/abs/1406.1078>
- Łukasz Czapka 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>
- 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>
- Leonardo de Moura and Nikolaaj 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.

- 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, Judicaël 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.
- 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>
- 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>
- 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/ntl>
- 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/>
- 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>
- 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>
- 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>
- Cezary Kaliszyk, François Chollat, 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>
- 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>
- 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>
- 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
- 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
- Daniel Kästner, Joerg Barroh, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler.
- Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. <http://xavierleroy.org/publi/comp-cert-CACM.pdf>
- 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>
- 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>
- 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>
- 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>

- 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>
- Lawrence C. Paulson. 1993. Natural Deduction as Higher-Order Resolution. *CoRR* cs.LO/9301104 (1993). <http://arxiv.org/abs/cs.LO/9301104>
- 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.
- 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>
- 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>
- 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>
- 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>
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *PLDI* (2011).