Diversity-Driven Automated Formal Verification

Emily First
University of Massachusetts Amherst
Amherst, MA, USA
efirst@cs.umass.edu

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

ABSTRACT

Formally verified correctness is one of the most desirable properties of software systems. But despite great progress made via interactive theorem provers, such as Coq, writing proof scripts for verification remains one of the most effort-intensive (and often prohibitively difficult) software development activities. Recent work has created tools that automatically synthesize proofs or proof scripts. For example, CoqHammer can prove 26.6% of theorems completely automatically by reasoning using precomputed facts, while TacTok and ASTactic, which use machine learning to model proof scripts and then perform biased search through the proof-script space, can prove 12.9% and 12.3% of the theorems, respectively. Further, these three tools are highly complementary; together, they can prove 30.4% of the theorems fully automatically. Our key insight is that control over the learning process can produce a diverse set of models, and that, due to the unique nature of proof synthesis (the existence of the theorem prover, an oracle that infallibly judges a proof's correctness), this diversity can significantly improve these tools' proving power. Accordingly, we develop Diva, which uses a diverse set of models with TacTok's and ASTactic's search mechanism to prove 21.7% of the theorems. That is, Diva proves 68% more theorems than TacTok and 77% more than ASTactic. Complementary to CoqHammer, Diva proves 781 theorems (27% added value) that CoqHammer does not, and 364 theorems no existing tool has proved automatically. Together with CoqHammer, Diva proves 33.8% of the theorems, the largest fraction to date. We explore nine dimensions for learning diverse models, and identify which dimensions lead to the most useful diversity. Further, we develop an optimization to speed up Diva's execution by 40×. Our study introduces a completely new idea for using diversity in machine learning to improve the power of state-of-the-art proof-script synthesis techniques, and empirically demonstrates that the improvement is significant on a dataset of 68K theorems from 122 open-source software projects.

CCS CONCEPTS

• **Software and its engineering** → **Software verification**; **Formal software verification**; • **Theory of computation** → **Automated reasoning**.

KEYWORDS

Automated formal verification, language models, Coq, interactive proof assistants, proof synthesis

ACM Reference Format:
Emily First and Yuriy Brun. 2022. Diversity-Driven Automated Formal Verification. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510138

1 INTRODUCTION

Building provably correct systems is critical in high-stakes domains, such as aerospace engineering and software for medical devices. However, most industrial verification tools either aim to simplify the verification process by sacrificing soundness [11] or significantly restrict the programming language in which the system is written [53]. A promising method for building correct software has been to use programming languages that are designed to inherently support program verification, such as interactive theorem provers (ITPs), including Coq [79], Agda [84], and Isabelle/HOL [61]. ITPs have had significant impact on industry. For example, Airbus France uses the Coq-verified CompCert C compiler [50] to ensure safety and improve performance of its aircraft [75]. Chrome and Android both use cryptographic code formally verified in Coq to secure communication [25], while Mozilla has its own verified cryptographic library for Firefox, improving performance [44]. Multiple companies have been successful in using proof assistants to provide formal verification services, including BedRock Systems, who builds formally verified solutions for the healthcare, infrastructure, and financial domains [9], Certora, who formally verifies smart contracts [17], and Galois, Inc., who verifies compiler correctness and hardware design [29]. Meanwhile Amazon successfully applies formal verification to cloud security problems in Amazon Web Services, providing tools for users to detect entire classes of misconfigurations that can potentially expose vulnerable data [6].

With ITPs, the user (a programmer) specifies a theorem about a property of the software and writes a *proof script*, a series of annotated proof tactics, that the interactive theorem prover uses to attempt to construct a proof of the theorem. Still, even with the help of an ITP, the effort required to write proof scripts is often prohibitive. The Coq proof of the C compiler is more than three times that of the compiler code itself and took three person years of work [50]. Meanwhile, it took 11 person years to write the proof script to verify a microkernel [59]. As a general rule, because of the expense of verification, nearly all software companies ship is unverified.

However, some formal verification can be fully automated by synthesizing either the underlying proofs or the guiding proof scripts. A series of tools called *hammers* (e.g., CoqHammer [21]) use a set of precomputed mathematical facts to attempt to "hammer" out

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510138

a proof. Evaluated on the CoqGym benchmark [90], CoqHammer can automatically prove 26.6% of theorems found in open-source Coq projects. But hammers are restricted by their precomputed facts and cannot reason about proof approaches such as induction, greatly limiting their power. To overcome these limitations, researchers have used machine learning to model existing proof scripts, and then, given a new theorem, applied that model to guide metaheuristic search [35] to attempt to synthesize a new proof script [28, 69, 90]. While these tools tend to prove fewer theorems, e.g., ASTactic proves 12.3% [90] and TacTok proves 12.9% [28], they are capable of applying higher-order proof approaches learnt from existing proofs, including induction, and so are complementary to hammers. Together with CoqHammer, they prove 30.4% of the theorems. The central goal of this paper is to improve on this fraction, particularly focusing on the tools that model existing proofs.

We make two key observations that enable us to improve the proving power of proof-script-synthesis techniques. First, the formal verification domain is a unique application of machine learning because it has a correctness oracle. In most machine learning applications, it is not known when the model is correct. This is why models are typically evaluated for precision or accuracy. In the formal verification domain, however, the interactive theorem prover can use a synthesized proof script to determine whether it truly proves the underlying theorem. If the prover can get to **Qed**, then the synthesized proof script must be correct. Thus, proof-script-synthesis systems always have a precision of 100%: they never return a failed script, instead continuing the search or timing out. While recall may be low, precision is always perfect. Second, variations in the models can alter the search-based synthesis of a proof script enough that two models can potentially produce different scripts for the same theorem. This, in turn, can, hypothetically, lead to models that prove complementary sets of theorems. And because of our first observation, they can be combined without sacrificing their power. The combined system can synthesize successful proof scripts for all theorems each one of the models can prove individually; if one model fails to synthesize a successful script, the theorem prover unequivocally tells us so, and we instead use the other model's successful script. Thus, if one can learn models that differ in a way to produce different scripts, potentially, this set of models may be able to prove far more theorems than a single model. The central question this paper answers is whether model diversity can be created to improve the proving power of proof-script-synthesis techniques, and whether such an approach improves on the state-of-the-art automated formal verification techniques. We find that the answer to both questions is "yes." As we will demonstrate on a benchmark of 68,501 theorems from 122 open-source software projects in Coq, we are able to create a set of 62 models by varying learning parameters and learning data that, together, prove 68% more theorems than TacTok and 77% more than ASTactic, despite using the same search method. Combining our approach, Diva, with CoqHammer [21], we can prove 33.8% of all the theorems, the highest such result to date. Diva proves 364 theorems that none of the prior tools have been able to prove. The difficulty of manually writing proof scripts for formal verification is so great, that even small improvements in proving power can be significant, and the savings in human effort that our approach represents are quite substantial.

Our insights enable for a completely new way to combine machine learning models. Of course, the idea of combining models is not new. Ensemble learning allows weighting the results of multiple models to improve the precision or recall of a single model [68]. And stacking uses a classifier to decide which model to apply to each input [24]. While both these methods can improve precision and recall in practice, they can also, hypothetically, reduce them, and often cannot properly amplify the correct results of a small minority of models. By contrast, in our domain, our method for combining models can never produce a wrong result or ignore the correct result produced by even a single model. This represents a killer app for ensemble learning and stacking. We are the first to combine the idea of ensemble learning with an oracle to produce optimal stacking.

This paper explores nine dimensions for learning diverse models, and identifies which dimensions lead to the most useful diversity. Altering the types of information (the proof script, state, and term) the model learns from resulted in the greatest diversity, while varying the depth of the proof script and the learning rate provided the second most diversity. As running a large number of models can be inefficient, we develop a model interrupts optimization that speeds up Diva's execution by 40×.

The main contributions of our work are:

- A novel approach for combining varied machine learning models to formally verify software properties.
- A systematic exploration of which learning dimensions provide usable model diversity.
- An implementation of our approach, Diva, that proves 68% more theorems than TacTok and 77% more than ASTactic, the prior work most closely related to ours. Diva is open-source and is available at https://github.com/LASER-UMASS/Diva/.
- An optimization for improving Diva's performance.
- A platform for evaluating models and rerunning experiments, and all data and source code used in our experiments for replications [27].

The rest of this paper is structured as follows. Section 2 explains verification in Coq. Section 3 presents Diva, and Section 4 evaluates our use of diversity to increase the proving power of automated formal verification tools. Section 5 places our research in the context of related work, and Section 6 summarizes our contributions.

2 THEOREM PROVING IN COQ

Coq is a dependently-typed language with a small kernel, which provides a high assurance that Coq-verified programs are truly correct. However, program verification in Coq is not automatic. To prove a theorem in Coq, a programmer must write a proof script (in Ltac), which, when executed, helps automatically generate a proof (in Gallina) of the theorem. Alternatively, metaheuristic search techniques [35] can automatically search for a proof script, thus alleviating the burden for the programmer [28, 69, 90]. However, metaheuristic search is only as good as the predictive model that is used to bias the search. In this section, we will discuss how a programmer interactively writes proof scripts in Coq (Section 2.1), how metaheuristic search can be used to automatically generate a proof script (Section 2.2), and design considerations for building a predictive model to generate proof scripts (Section 2.3).

2.1 Interactively writing proof scripts

When a theorem is proven in Coq, this means that in Gallina (Coq's internal language), a *proof term* of the desired type has been constructed. The type of this term is the theorem itself. A programmer could write the Gallina proof term themselves, but this can be a long, unforgiving process [65]. To simplify this task, Coq has a meta-programming language called Ltac in which programmers can write *proof scripts*, which when completed and run, generate the Gallina proof term automatically.

Programmers use an *interactive proof assistant* (e.g., CoqIDE or Proof General) to write proof scripts, which consist of a sequence of *proof tactics*. The proof assistant executes a proof script, even a partial one, and provides immediate human-readable feedback after each tactic's execution. This feedback is Coq's internal *proof state*, which includes the *goals* to prove, the *local context* of assumptions, and the *environment* of proven-so-far set of facts. The programmer can even ask to see the intermediate Gallina proof term by writing and executing the **`Show Proof`** command in their proof script. When starting to prove a theorem, Coq's proof state is a single goal, which is the theorem itself (the corresponding proof term is `?Goal`). The aim is to manipulate the proof state through the use of tactics until the goal is proven and thus removed from the proof state. Since the search space of goal manipulation is too large, a programmer helps manage the exploration by using the current proof state to select a sequence of proof tactics to try.

The interactive proof assistant checks that a partially-written proof script is valid and updates the current proof state, allowing the programmer to incrementally develop a proof script. The programmer can choose a tactic, examine the output from the proof assistant, and then choose the next tactic. If the programmer chooses an invalid tactic, the proof assistant displays an error. If the programmer chooses tactics that are valid, but do not make progress, they can use the proof assistant to backtrack to an earlier proof state and try a different approach. The programmer continues selecting tactics until the proof assistant prints *no more subgoals*, and then uses **`Qed`** to complete the proof script.

2.2 Proof script synthesis via metaheuristic search

In interactive proof script generation, the burden is on the programmer to choose the sequence of tactics. To remove this burden from the programmer, metaheuristic search techniques can sometimes automatically generate a proof script.

The space of possible proof scripts is infinite and quite complex. Because of this size and complexity, automatically searching blindly through this space for a proof script that might prove a theorem is unlikely to succeed. Metaheuristic search [35] can help guide the search to improve the chances of success, and, in fact, is often successful [28, 69, 90]. Such search starts with an empty proof script, and predicts a first most likely proof step. This prediction can be made based, for example, on the theorem being proven and examples of past, successful proof scripts. The search executes the partial proof script and determines, using some heuristic-based fitness function, whether adequate progress has been made. If it has not, the proof search can try another likely proof step. If it has, the search can iteratively augment the partial proof script, adding subsequent predicted proof steps, making progress toward proving

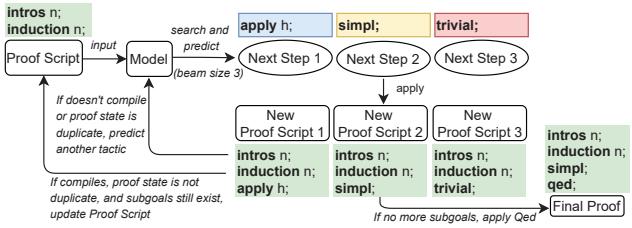


Figure 1: The process of synthesizing a proof script using metaheuristic search, biased by a black box predictive model. Given an incomplete proof script, a model can predict the next proof step. Here, using a beam search of width 3, the model predicts 3 likely next steps. If using the step satisfies certain criteria (here, the proof script must compile and the resulting proof state must not have been previously seen within this proof script) the process iterates until, either, the proof state has no subgoals and the proof script can be completed using `Qed`, or the search reaches a timeout.

the theorem. Making reasonably accurate predictions is, of course, a critical part of successful metaheuristic search, and Section 2.3 will describe possible ways to do that.

Figure 1 shows how beam search can use a predictive model to bias a metaheuristic search for a proof script. In this example, the model predicts 3 likely next proof script steps (the beam width is 3). The search then uses a heuristic-based fitness function to determine criteria for applying the candidate proof steps. Here, the criteria are that the partial proof script compiles and results in a proof state that has not been previously seen within this search. If successful, the search appends the proof step to the script and iterates, growing the script. Once the proof state has no more subgoals, the proof script can be completed by using **`Qed`**. The search fails if it times out.

2.3 Proof script modeling

Prior proof script synthesis tools, such as ASTactic and TacTok, use the predictions from learnt proof script models to bias the metaheuristic search for a proof script. Such a model is learnt from a set of existing, successful proof scripts to predict the next proof step (tactic and arguments) of an incomplete proof script. Recall from Section 2.1 that there are three relevant aspects of proof scripts we may want to encode to serve as input to such a model: the proof state, the proof script, and the Gallina proof term. ASTactic only encodes the proof state, while TacTok encodes both the proof state and the proof script. There has yet to be a proof script synthesis tool that encodes the Gallina proof term. Next, Sections 2.3.1, 2.3.2, and 2.3.3 describe how to encode the proof state, proof script, and Gallina proof term, respectively.

2.3.1 Encoding the proof state. The proof state consists of the goals to be proven, local context, and the environment. While the programmer sees them in a human-readable format, each term of the proof state has an underlying abstract syntax tree (AST) representation. ASTactic and TacTok serialize these ASTs and encode them using a neural model, specifically a TreeLSTM [78]. Prior work has

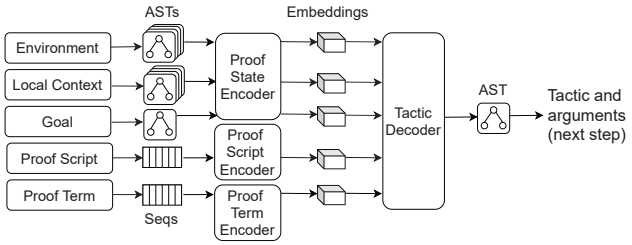


Figure 2: Model of proof script, which Diva uses internally to drive search.

empirically argued that neural models are much more effective than other architectures [10, 56, 76].

2.3.2 Encoding proof script features. The proof script is comprised of a sequence of tokens in Ltac. For the model to encode these tokens, each proof script needs to be preprocessed to remove high-frequency low-signal tokens, such as punctuation. Then, encoding such a sequence is traditionally done using a language model [10, 56, 76]. Language models are widely used in natural language processing tasks [7, 74]. The primary function of a language model is to predict the next token in a sequence of tokens. While prior work has used n-grams to model Coq [37], TacTok found neural language models work better to encode the sequence of tokens because it can generate a representative vector (embedding) for the sequence, that can then be combined with other types of inputs. Among the most extensively used neural language models are transformers [23] and RNNs [62]. TacTok uses an RNN (specifically a Bidirectional LSTM [62]) because transformers require massive amounts of data to train [23], which is typically not available in the formal verification domain.

2.3.3 Encoding the proof term. Prior tools have not encoded proof terms, but, conceptually, the Gallina sequence is similar to the proof script Ltac sequence, and we encode it in a similar way using a Bidirectional LSTM [62]. This allows all three, the proof state, proof script, and proof term, to be encoded with a single model.

3 DIVA: DIVERSITY-DRIVEN SYNTHESIS

Machine learning models can be sensitive to noise in the training data [60, 82] and to parameters applied during the learning process [31]. This sensitivity can cause great variability in the accuracy of models. Of course, this can hurt the generalizability of machine learning results, but we posit that in the right domain, this sensitivity, and the diversity of models it can produce, can provide a significant benefit.

In the formal verification domain, tools such as ASTactic [90], Proverbot9001 [69], and TacTok [28] use a learnt model of a proof script to guide metaheuristic search toward synthesizing a proof script for a theorem. Variations in the models can alter the search, resulting in potentially different attempted synthesized scripts. The key uniqueness of this domain is that an interactive theorem prover can act as an oracle for each proof script. If the proof script leads the theorem prover to generate a proof terminating in `Qed`, then the proof script is, by definition, correct. This allows a synthesis tool to try applying many different models to bias the search in different ways, and then pick out just the successful synthesis attempts, discarding the failed ones.

(a) Proof state encoder (b) Proof script and term encoders

Figure 3: The neural models used to encode the proof state AST, proof script sequence, and the proof term sequence.

This is not the typical case in applications of machine learning. Ensemble learning [68] and stacking [24] attempt to combine the results of multiple machine learning models to improve precision or recall. However, without an oracle, ensembles and stacks are unlikely to always pick the correct result, especially when relatively few of the diverse models produce it. By contrast, in our domain, with the theorem prover acting as an oracle, even a single model producing the correct proof script can establish an answer.

To demonstrate this insight, we develop Diva, a proof-script-synthesis tool that uses the diversity in machine learning to significantly improve its proving power. Diva is open-source and is available at https://github.com/LASER-UMASS/Diva/.

Diva's key contributions are the generation of a diverse set of models capable of proving complementary sets of theorems, a mechanism for combining the benefits of the models, and an optimization to make running a large number of searches using independent models feasible.

To automate proof script synthesis, Diva uses a learnt model of a proof script to guide metaheuristic beam search. During this search, Diva samples a fixed number (beam width) of the most likely tactics, predicted by the model, across all search tree nodes at the same level, and then uses these tactics to search for a complete proof script. Diva backtracks when the Coq compiler fails to check the attempted proof script step or detects a duplicate proof state. Diva uses the same beam search configuration (width of 20, search depth limit of 5, and a timeout of 10 minutes) as ASTactic and TacTok.

To intentionally produce a diverse set of models that prove complementary sets of theorems, control over the learning process is key. When training a model of proof scripts, Diva varies the learning parameters and which features of the training data to encode. Next, Section 3.1 describes what a Diva model looks like; Sections 3.2 and 3.3 detail how Diva generates a diverse set of models by controlling learning parameters and the encoded features of the training data, respectively; and Section 3.4 explains our Diva efficiency optimization.

3.1 Diva's learnt model

Figure 2 illustrates Diva's proof script model, learnt from a set of existing proof scripts. Diva uses the predictions from this model to drive the search for a complete proof script.

Figure 3 details the encoders used in the Diva model to encode relevant aspects of proof scripts. Figure 3(a) presents the proof state encoder, which Diva uses to encode the goal, local context, and environment, in AST form. To encode a tree, it uses a TreeLSTM network [20], which generates embeddings for each proof state term. Figure 3(b) details the proof script encoder, which Diva uses to encode the proof script sequence. We encode the parsed sequence

of previous tokens using a Bidirectional LSTM, which generates an embedding for the sequence. A Bidirectional LSTM improves on the LSTM by capturing more contextual information by processing the input sequence in two ways, forward and backward [62], allowing the output layer to simultaneously see both directions of information. Diva encodes the Gallina proof term (the first synthesis tool to do this) using the same encoder in Figure 3(b). Similar to the proof script sequence encoding, we choose to encode the sequence of proof term tokens using a Bidirectional LSTM, generating an embedding. Diva jointly learns embeddings for the sequences and ASTs.

Diva's tactic decoder is modified from the tactic decoder first used in ASTactic and, later, TacTok. This tactic decoder is conditioned on the sequence of embeddings. In Diva, however, the embeddings are a concatenation of a subset of the embeddings generated from the proof script, proof term, and proof state encoders. This allows for modeling of more relevant proof script aspects and the choice of which subset to combine allows us to create variability in the models (see Section 3.3). The tactic decoder then generates a tactic by sequentially growing an AST [91]. It chooses a production rule from the context free grammar of the tactic space at a non-terminal node in the AST, while it synthesizes arguments based on semantic constraints at a terminal node. A GRU [19, 20] controls this process of growing the tree, as it updates its hidden state using the input embeddings of the partially generated AST.

Diva trains the model on a set of existing proof scripts. Each proof script in this set is broken down into training instances, which are the inputs to the model. A training instance is comprised of the proof state before the tactic execution, the proof script up to the tactic execution, the Gallina proof term before the tactic execution, and the next step of the proof script. The Diva model jointly learns embeddings for the proof state ASTs, the proof script, and proof term sequence, and then uses these embeddings to predict the next proof script step in the form of an AST. The model sends the predicted AST along with ground-truth next tactic AST to the trainer, where the trainer compares these tactic ASTs and back-propagates the loss.

Unlike prior tools, Diva jointly trains a language model over the tokens in the proof term. Section 3.2 details further modifications in this training process for creating Diva's diverse models.

3.2 Diversity via varying learning parameters

One way in which we create a diverse set of models is by varying the learning parameters, which affects the model's size and the learning algorithm itself. For this, we start with the Tac model from TacTok, and explore varying six dimensions: sequence tactic depth, sequence token depth, the learning rate, the embedding size, the number of layers, and the order of the training data.

Tactic and token sequence depth. The sequence depth denotes the size of the input the learning algorithm considers. When training, the model can consider the entire proof script written so far, or part of it, such as only the most recent tactic and its arguments, or several most recent tactics with arguments, or only several most recent tokens. The proof script encoder considers only that portion of the proof script (and, symmetrically, the decoder will consider the same depth when decoding the next proof step). Diva varies the sequence depth along both tactics and tokens, from a depth of 0, which does not consider the proof script at all (it considers only proof state, making the model equivalent to ASTactic's model), to the entire proof script. Diva considers sequence depth sizes (excluding the start token) of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30.

Learning rate. During training, the algorithm updates the model's weights in every iteration. The learning rate is a hyperparameter that determines how much the weights can be changed in each iteration. A larger learning rate is less likely to result in the training getting stuck in a local optimum, but may also take longer to converge or fail to explore a region long enough to find an optimal solution. Accordingly, the models produced by varying the rate can be quite different. Diva considers learning rates of 3×10^k for $k \in \{-2, -3, -4, -5, -6, -7, -8\}$.

Model size (embedding and layers). The model's size is defined by two hyperparameters, the number of model layers and embedding size, which is the size the vector space in which a proof aspect is embedded. Diva varies the proof script encoder size by trying 1, 2, 3, 4, and 5 layers and embedding sizes of 64, 128, 256, and 512.

Training data order. The order of the training data can affect the model [31]. We vary the order in which Diva sees the training instances by creating ten random orders.

3.3 Diversity via varying training data

The second way in which we create a diverse set of models is by varying aspects of the training data available to the learning algorithm. There are three types of data in the training proof scripts: the proof state, the proof script tactics and tokens, and the proof term that the proof assistant generates when it executes the proof script (recall Section 2.3). When training a model, we either include each of these three types of data or we exclude them. For the proof script, we include either the tactics or the tokens, since they encode fundamentally the same information. This leads us to a total of 11 models. For the models that do not include proof state, when we encode the training instance that represents the very start of proof-script synthesis, we include the theorem being proven (otherwise the model would not know what it is trying to prove). Similarly, at test time, when synthesizing the first proof script step, we include the theorem being proven.

3.4 Efficiently combining model executions

Executing a large set of models in sequence is slow since Diva has to wait for a model to finish its proof script synthesis attempt before it can try the next one. We develop model interrupts to improve Diva's efficiency. In model interrupts, given a set of models, Diva assigns an arbitrary order of model application. In order, each model will be given a specified amount of time to try to synthesize a proof script. Once the time runs out, the next model attempts to synthesize a proof script from scratch. Figure 4 illustrates this concept. The first model attempts synthesis from scratch for X seconds, at which point, if a complete proof script is not generated, the partial proof script is stored and the second model attempts to synthesize a proof script from scratch for X seconds. And so on. Once each model is given an opportunity to try for X seconds and a complete proof script is not found, the models will be given more time to synthesize

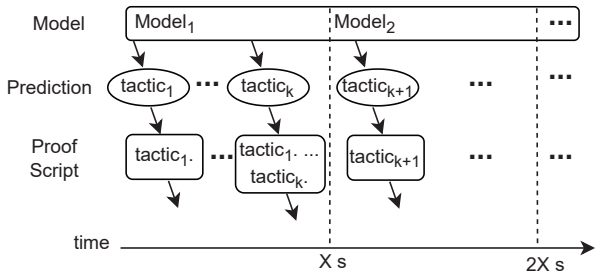


Figure 4: Model interrupts allows Diva to let models take turns synthesizing proof scripts from scratch.

a proof script starting from the stored partial proof script associated with the model.

4 EVALUATION

We evaluate Diva to measure how much diversity of models of proof scripts can increase the effectiveness of proof script generation. We follow the methodologies of prior evaluations of proof-script synthesis tools [28, 90], in terms of the dataset (Section 4.1.1) and metrics (Section 4.1.2) used; we compare to two state-of-the-art proof-script-synthesis tools, ASTactic [90] and TacTok [28], which use the same metaheuristic search for proof-script synthesis as Diva. We further compare Diva to the state-of-the-art proof-synthesis tool CoqHammer [21].

Our evaluation answers four research questions:

RQ1: Does diverse-modeling significantly improve proof-script synthesis over state-of-the-art approaches CoqHammer, ASTactic, and TacTok?
RQ2: How much model diversity results from varying the model learning parameters sequence depth, learning rate, number of layers, size of embeddings, and training order, and how does this model diversity affect proof script synthesis effectiveness?
RQ3: How much model diversity results from varying which aspects of the training proofs — tactics, tokens, proof state, Gallina proof terms — are available to the learning process and how does this model diversity affect proof script synthesis effectiveness?
RQ4: How effective is our interrupts mechanism for improving Diva efficiency?

All of our evaluation data and the source code to reproduce our results are available [27].

4.1 Evaluation methodology

We first describe the dataset and metrics we use to evaluate Diva.

4.1.1 Dataset. In our evaluation, we use CoqGym [90], the state-of-the-art benchmark used in prior evaluations of formal verification tools [28, 51, 90]. The benchmark consists of 70,856 theorems from 123 open-source software projects in Coq. The CoqGym benchmark comes with a preselected training set of 96 projects with 57,719 human-written proof scripts, and test set of the remaining 13,137 theorems from 27 projects.

Our earlier TacTok evaluation [28] was unable to reproduce prior results for ASTactic's performance [90] for one project, coq-library-undecidability, due to internal Coq errors when processing the proof scripts. Accordingly, we exclude this project from our evaluation. We were able to reproduce the results for the remaining 26 projects of 10,782 theorems. In total, our training and test sets have 68,501 theorems from 122 projects.

4.1.2 Metrics. We measure four quantities in answering our research questions: success rate, added value, diversity, and mean time to prove a theorem.

Success Rate. The success rate of a tool, widely used in prior evaluations [28, 41, 90], is the fraction of all theorems for which the tool generates a succesful proof script.

Added Value. The added value of tool A over tool B is the number of new theorems tool A proves that tool B does not, divided by the number of theorems tool B proves.

Diversity. Given a set of models, we wish to know how much diversity they yield with respect to their ability to prove theorems. And so, we think of the diversity of a set of models as the diversity of the corresponding sets of theorems that the models prove. Our goal with the diversity measure is to be able to compare how much diversity results from various methods for creating models, so that we can compare the different methods.

Informally, given a set of sets of objects (theorems) we define a family of diversity functions, such that the k^{th} diversity function, d_k, measures the relative increase in objects contained in k sets, as compared to $k-1$ sets. So, for example, for a set of models, d_5 denotes the fraction of the additional theorems (out of all the theorems proved by at least one model) that are able to be proved by adding a fifth model to a set of four models, on average.

More formally, let T be a set of objects and let M be a set of subsets of T such that the union of all sets in M is equal to T. Then, for each $k \in \{1, 2, 3, \ldots, |M|\}$, the k^{th} diversity function $d_k \colon 2^T \to \mathbb{R}$ is the average increase, in terms of the fraction of T, that the union of k elements of M contains over the union of $k-1$ elements of M. Thus, for all $M_k \subseteq M$, such that $|M_k| = k$, and for all $M_{k-1} \subseteq M$, such that $|M_{k-1}| = k-1$, $d_k(M)$ is the average value of $\frac{|M_k \setminus M_{k-1}|}{|T|}$.

Given a set of models, we compute the diversity functions empirically. We use each model to attempt to synthesize proof scripts to prove theorems. We then compute T, the set of all theorems that can be proven by at least one model. Then, to compute d_k, we, for each model, compute how many *additional* theorems it proves compared to each set of $k-1$ models. We then compute the average of those numbers, and divide it by $|T|$ for normalization. In the end, $d_k(M)$ is the average fraction of theorems proven by adding a k^{th} model to a set of $k-1$ models. Note that the sum of d_k for all k is 1, and that diversity is monotonically non-increasing with respect to k (that is, $d_{k-1} \leq d_k$.)

Mean Time to Prove a Theorem. To measure efficiency, we compute the mean time it takes to generate a proof script for a theorem, averaged over all the theorems for which we produce a successful proof script, and over all the possible orderings of the models used in the metaheuristic search.

tool	theorems proven	Diva's value added
ASTactic	1,322 (12.3%)	908 (76.9%)
TacTok	1,388 (12.9%)	842 (68.4%)
CoqHammer	2,865 (26.6%)	781 (27.3%)
all 3 prior tools	3,282 (30.4%)	364 (11.1%)
Diva	2,338 (21.7%)	—
Diva & CoqHammer	3,646 (33.8%)	—

Figure 5: Theorems proven by and the success rate of Diva, ASTactic, TacTok, CoqHammer, and the combination of these tools out of the 10,782 theorems in CoqGym's test dataset. Diva provides value added over each of these tools, and 11.1% value added over the combination of all three.

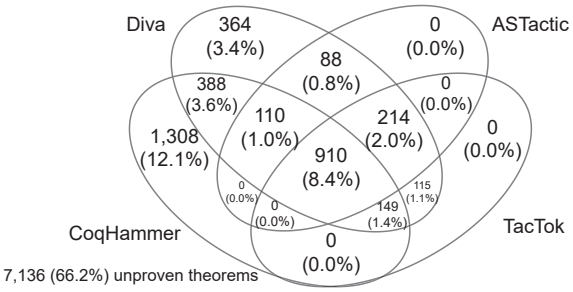


Figure 6: The breakdown of how many theorems are proven by each combination of tools. Diva proves 364 theorems no other tool proves.

4.2 RQ1: Does diversity help Diva outperform the state-of-the-art?

We created models by varying learning parameters and aspects of proof scripts to encode (recall the models described in Sections 3.2 and 3.3). Overall, we generated these 62 models for Diva to use.

We compare Diva to the state-of-the-art synthesis tools, ASTactic [90], TacTok [28], and CoqHammer [21]. ASTactic and TacTok, like Diva, learn from existing proof scripts to predict the next step of the proof script. CoqHammer uses a fundamentally different approach. Whereas CoqHammer produces proofs in Coq's logic (Gallina), Diva searches the proof-script space. When the Coq compiler executes a proof script, it generates a proof. Proofs cannot be wrong, while proof scripts can be (e.g., a proof script that concludes with *Proof completed*, may not lead to a valid proof when it is checked by the Coq compiler). Thus, it is reasonable to compare proof script synthesis tools, such as Diva, to CoqHammer with respect to the theorems they are able to prove. However, since their approaches are so fundamentally different, it is expected that these tools are likely to be complementary, performing well for different theorems. While CoqHammer and Diva are likely to perform similarly well for some simpler classes of theorems, CoqHammer is at a fundamental disadvantage, though, for other classes of theorems, such as ones that require induction to prove.

On our evaluation set of 10,782 theorems, ASTactic proves 1,322 (12.3%) and TacTok proves 1,388 (12.9%) theorems. CoqHammer proves 2,865 (26.6%) theorems. Prior to performing our evaluation, we expected that Diva would prove strictly more theorems than ASTactic and TacTok (though how many more remained an important question), that it would not prove more theorems than CoqHammer, but that it would prove some complementary theorems, thus providing significant added value compared to CoqHammer, as was the case in ASTactic and TacTok evaluations [28, 90].

Figure 5 shows the success rates, as well as the raw number of theorems proven by the four tools, and the value Diva adds over each tool, as well as their combination. Diva proves 2,338 (21.7%) of the theorems. This means Diva proves $\frac{2{,}338-1{,}322}{1{,}322} = 76.9\%$ more theorems than ASTactic and $\frac{2{,}338-1{,}388}{1{,}388} = 68.4\%$ more theorems than TacTok. Since these tools use the same search mechanism, these significant improvements are due entirely to the use of model diversity.

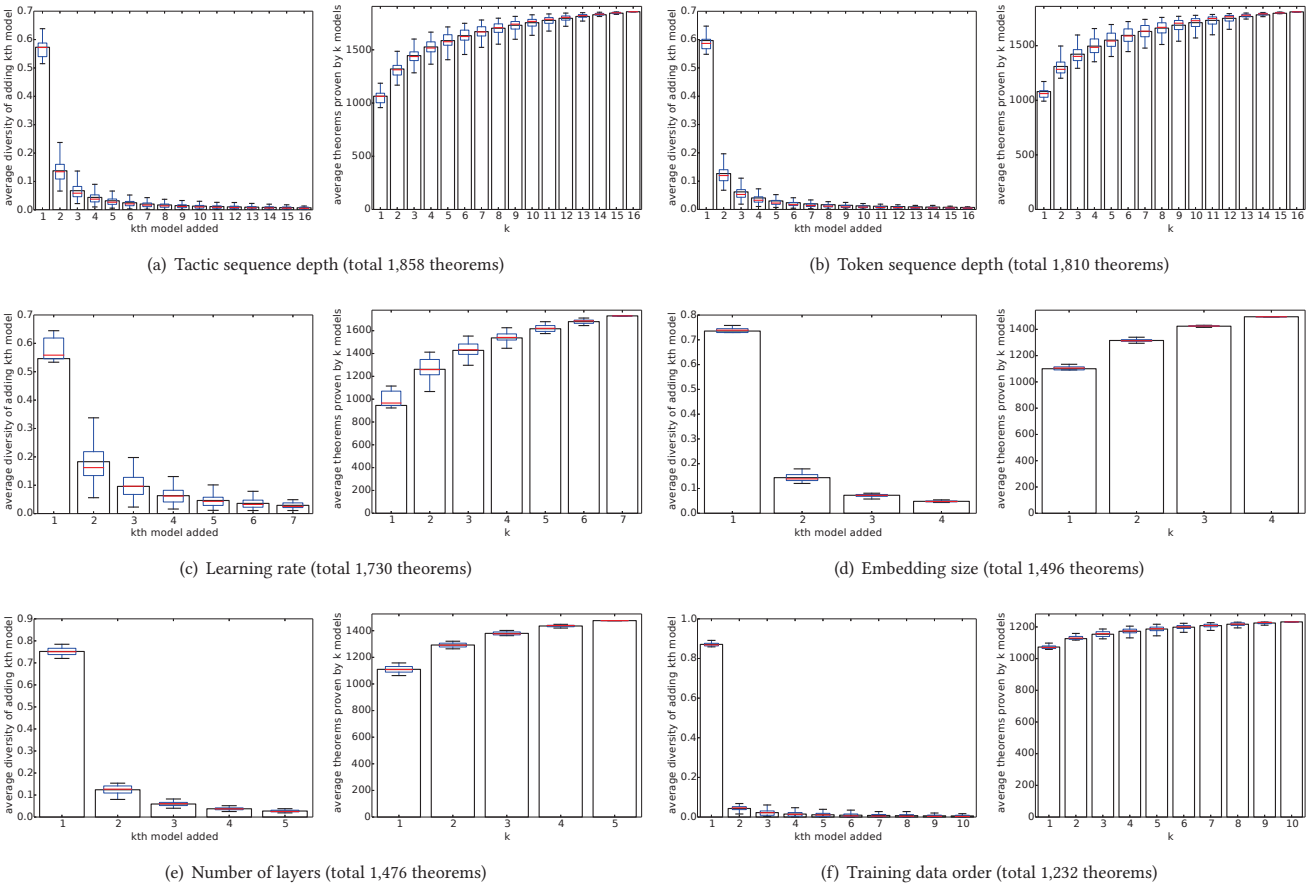
While CoqHammer proves more theorems than Diva, Diva proves 781 theorems that CoqHammer does not, an added value of $\frac{781}{2{,}865} = 27.3\%$. Figure 6 shows a Venn diagram of the theorems Diva, ASTactic, TacTok, and CoqHammer prove. Together, these four tools prove 3,646 theorems, for a success rate of 33.8%, whereas without Diva, the other three tools prove 3,282 theorems. (Because ASTactic and TacTok have an added value of 0% over Diva, CoqHammer and Diva prove the 3,282 theorems on their own, without the other tools' help.) Diva adds a value of 11.1% over the combined state of the art, and proves 364 theorems no tool has previously proven.

> RA1: Our Diva diversity mechanisms are successful in creating model diversity sufficient to significantly improve the proving power of metaheuristic-search-based tools (68%–77% added value). Diva also generates 27.3% added value over CoqHammer, and proves 364 theorems no prior tool has proven. Together with CoqHammer, Diva reaches a new milestone, proving over one third of all theorems completely automatically.

4.3 RQ2: Learning-parameter diversity

To investigate the effectiveness of varying learning parameters on generating diverse models, we conduct a series of experiments by generating models varying those parameters, using the resulting models to synthesize proof scripts, and then measuring the *diversity* of the sets of theorems the models prove. As Section 3.2 described, the factors we investigate are sequence depth, learning rate, number of layers, embedding size, and training order. Figure 7 details how much diversity Diva produces by varying learning parameters in training its models.

Tactic depth diversity. We vary the tactic sequence depth, considering depths of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30; a total of 16 models. (Note that the depth 0 model is equivalent to ASTactic, and the depth 3 model is equivalent to the Tac model in TacTok.) Overall these 16 models prove 1,858 theorems, whereas



(a) Tactic sequence depth (total 1,858 theorems)

(b) Token sequence depth (total 1,810 theorems)

(c) Learning rate (total 1,730 theorems)

(d) Embedding size (total 1,496 theorems)

(e) Number of layers (total 1,476 theorems)

(f) Training data order (total 1,232 theorems)

Figure 7: The diversity exhibited by altering learning parameters tactic sequence depth (a), token sequence depth (b), learning rate (c), embedding size (d), number of layers (e), and training data order (f). The left graph in each pair shows the diversity measure, as a function of the number of models (e.g., the $k = 5$ bar is the mean fraction of additional theorems proven by picking a random 5^{th} model that a random disjoint set of 4 models has not proven). The right graph in each pair shows the mean number of theorems proven by k models. The box-and-whiskers indicate the maximum, 75%-, 50%-, and 25%-tiles, and minimum values.

on average, a single model proves 1,064 theorems. Diva's diversity is responsible for a 74.6% increase in proving power! The left graph in Figure 7(a) shows the diversity of the set of tactic sequence depth models (recall the diversity metric from Section 4.1.2). The k^{th} bar shows d_k for the 16 models. That is, the k^{th} bar states the fraction of *extra* theorems proven by k random models, that a random set of $k-1$ models does not prove. For example, the $k = 1$ bar is simply the effectiveness of using a single model, 0.573 (on average, 57.3% of the theorems proven by all models together are proven by using one random model). The remaining 42.7% need Diva's diversity mechanism. For $k = 2$, the diversity is 0.138, meaning that adding the second model, on average, adds an additional 13.8% of the total theorems proven. Two randomly chosen models prove, on average, 57.3% + 13.8% = 71.1% of all the theorems proven by at least one model. The right graph in Figure 7(a) shows the average number of theorems that k of the tactic sequence depth models prove. The box-and-whiskers indicate the variability in the choice: how important is it to select specific k models, or can they simply be selected at random. For example, a single model can prove between 957 (8.9%) and 1,322 (12.3%) theorems from the test set. We leave developing mechanisms for selecting models to future work.

Token depth diversity. Similar to tactic depth, we considered token depths of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 29 and 30; a total of 16 models. (Note that the depth 0 model is, again, equivalent to ASTactic, and the depth 30 model is equivalent to the Tok model in TacTok.) Overall these 16 models prove 1,810 theorems (slightly fewer than tactic depth diversity models did), whereas on average, a single model proves 1,080 theorems. Diva's diversity is responsible for a 67.6% increase in proving power. The left graph in Figure 7(b) shows the diversity of the token depth models. A single random model proves a slightly larger fraction, 59.7%, of all the proven

theorems than was the case for tactic depth models, indicating again that token depth provides slightly less useful diversity. Still, the remaining 40.3% of the theorems require Diva's diversity to be proven. The right graph in Figure 7(a) shows the variability in a selected k models. Here, a single model can prove between 992 (9.2%) and 1,322 (12.3%) theorems from the test set. Overall, token depth provides significant diversity, but less than tactic depth did.

Learning rate. We explore 7 different learning rates: 3×10^k for $k \in \{-2, -3, -4, -5, -6, -7, -8\}$. Overall these 7 models prove 1,730 theorems (slightly fewer than the depth diversity models did), whereas on average, a single model proves 945 theorems. Diva's diversity is responsible for a 83.1% increase in proving power. The left graph in Figure 7(c) shows the diversity of the learning rate models. A single random model proves 54.6% of all the proven theorems. The remaining 45.4% of the theorems require Diva's diversity to be proven. The right graph in Figure 7(c) shows the variability in a selected k models. Here, a single model can prove between 505 (4.7%) and 1,115 (10.3%) theorems from the test set. Overall, learning rate provides significant diversity, and the models are more diverse from one another than the sequence depth models, but, overall, result in slightly less proving power.

Embedding size. We explore 4 different embedding sizes: 64, 128, 256, 512. Overall these 4 models prove 1,496 theorems (fewer than the already discussed models), whereas on average, a single model proves 1,100 theorems. Diva's diversity is responsible for a 36.0% increase in proving power. The left graph in Figure 7(d) shows the diversity of the embedding size models. A single random model proves 73.5% of all the proven theorems. The remaining 26.5% of the theorems require Diva's diversity to be proven. The right graph in Figure 7(d) shows the variability in a selected k models. Here, a single model can prove between 1,056 (9.8%) and 1,134 (10.5%) theorems from the test set. Overall, embedding size provides some diversity, though less than sequence depth and learning rate.

Number of layers. We explore 5 different numbers of layers: 1, 2, 3, 4, and 5. Overall these 5 models prove 1,476 theorems (similar to the embedding size), whereas on average, a single model proves 1,109 theorems. Diva's diversity is responsible for a 33.1% increase in proving power. The left graph in Figure 7(e) shows the diversity of the number of layers models. A single random model proves 75.2% of all the proven theorems. The remaining 24.8% of the theorems require Diva's diversity to be proven. The right graph in Figure 7(e) shows the variability in a selected k models. Here, a single model can prove between 1,063 (9.9%) and 1,158 (10.5%) theorems from the test set. Overall, varying the number of layers provides a similar amount of diversity as embedding size. Both parameters effect the size of the learnt model.

Training data order. We explore 10 randomly chosen orderings of the training data. Overall these 10 models prove 1,232 theorems, the smallest number of all the learning parameters, whereas on average, a single model proves 1,073 theorems. Diva's diversity is responsible for a 14.8% increase in proving power. The left graph in Figure 7(f) shows the diversity of the training data order models. A single random model proves 87.1% of all the proven theorems. The remaining 12.9% of the theorems require Diva's diversity to be proven. The right graph in Figure 7(f) shows the variability in a selected k models. Here, a single model can prove between 1,058 (9.8%) and 1,098 (10.2%) theorems from the test set. Overall, even just varying the training data order provided some useful diversity and enabled proving more theorems, though the diversity benefits were much smaller than those of the other parameters.

> RA2: Varying learning parameters resulted in significant diversity, which, in turn, led to significant improvement in proving power. Varying the depth of the tactics and tokens the model learnt from and the learning rate led to the greatest diversity, while varying the size of the model led to moderate diversity. Varying the order of the training data marginally increased the proving power.

4.4 RQ3: Training-data diversity

Recall from Section 3.3 that there are three types of data in the training proof scripts: the proof state, the proof script tactics and tokens, and the Gallina proof term. We train models for all possible combinations of these data types, except no model includes both tactics and tokens, and we exclude the model that is the empty combination. In total, we learn 11 models.

We first measure the value added by adding each of the three types of information. The value of adding proof script tactics to a model already encoding the proof state and the Gallina proof term is 134.2%, proving an additional 345 theorems. (The value of adding proof script tokens instead of tactics is similar, 136.6%, 351 theorems). The value of adding Gallina proof term to a model already encoding the proof script and the proof state is much smaller, 8.0%, proving an additional 89 theorems. (If using tokens instead of tactics, the added value is 10.6%, 124 theorems.) Finally, the value of adding proof state to a model already encoding the proof script and the Gallina proof term is 21.8%, proving an additional 135 theorems. (If using tokens instead of tactics, the added value is 53.5%, 281 theorems. We observe that while in previous scenarios, tactics and tokens behaved similarly, here, tokens exhibit much more diversity than tactics.) In all three cases, tokens exhibited greater diversity than tactics in encoding the proof script, suggesting that tokens are a more different representation than tactics of the other types of information. The Gallina proof term contained the least diversity compared to the other types of data, whereas the proof script contained the most.

Overall, these 11 models prove 2,053 theorems, which is significantly more than any of the learning parameter models from Section 4.3. A single model, on average, proves 785 theorems. Diva's diversity is responsible for a 161.5% increase in proving power! The left graph in Figure 8 shows the diversity of the training-data-types models. A single random model proves 38.3% of all the proven theorems. The remaining 61.7% of the theorems require Diva's diversity to be proven. The right graph in Figure 8 shows the variability in a selected k models. Here, a single model can prove between 257 (2.4%) and 1,322 (12.3%) theorems from the test set. Overall, training data types provide the most diversity of all the dimensions we explored, leading to the greatest proving power.

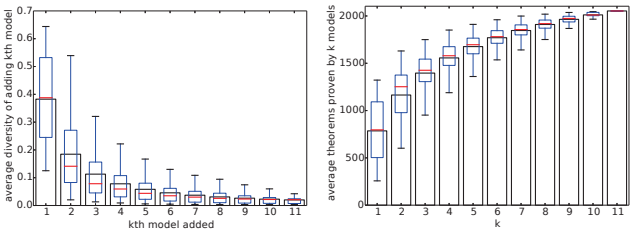


Figure 8: Training data aspects (total 2,053 theorems)

RA3: Including different data types in training resulted in the most diversity of all the dimensions we considered, leading to the greatest proving power increase. Adding proof script tactics or tokens provided the most diversity, followed by the proof state.

4.5 RQ4: Synthesis efficiency

To explore improving Diva's efficiency, we implement model interrupts described in Section 3.4. We evaluate the efficiency improvement of model interrupts by measuring the mean time to prove a theorem with and without interrupts. Of course, the order in which Diva considers the models matters. Without interrupts, in the worst case, the last model produces the successful proof script, and Diva wastes 10 minutes on each of the other models, before they time out. For our evaluation, we measure the mean time over a random sample of 20 possible model orderings.

Without interrupts, the mean time to prove a theorem is 685.5 seconds. However, we observe that most models either synthesize the proof script relatively quickly, or don't at all, though with some notable exceptions. Using model interrupts allows us to benefit from proving theorems quickly in the initial burst of each model, without spending the long time in the tail of each model's distribution, unless it is necessary. With model interrupts, we explore 15 different switching times: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, and 60 seconds.

We explore two different interrupt schemes. First, we attempt to synthesize a proof script using each model for X seconds. If none of the models find a proof script in that time, we return and give each model another X seconds. And so on, until each model has attempted its search for 10 minutes. The left graph in Figure 9 shows the mean time to prove a theorem for this interrupt scheme. For $X = 1$ second, this interrupt scheme achieves the minimal mean time to prove a theorem of 17.2 seconds, and the proving time increases monotonically for larger X. For $X = 1$, the speed up compared to not using interrupts is 97%, or 40×. This suggests that many theorems are proven very early in the synthesis process, and while some theorems do get proven after a lengthy synthesis search, prioritizing the first seconds of synthesis using the diverse models greatly improves synthesis efficiency.

Second, we allow each model to attempt to synthesize a proof script for X seconds, and then give each model the remainder of its $600 - X$ seconds, thus switching only once per model. The right graph in Figure 9 shows the mean time to prove a theorem for this interrupt scheme. For $X = 5$ seconds, this interrupt scheme

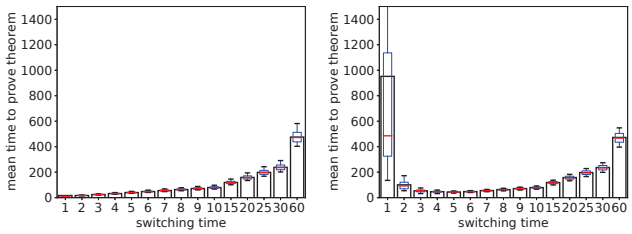


Figure 9: Mean time to prove a theorem using the model interrupts optimization for different switching times (in seconds). Executing each model's search for X seconds, and then again each model's search for X seconds, and so on until each model's search has been executed for 600 seconds (left graph), achieves the minimal mean time to prove a theorem of 17.2 seconds when $X = 1$ second. Interrupting each model's search once, first executing each model for X seconds, and then each model for $600 - X$ seconds, (right graph), achieves the minimal mean time to prove a theorem of 44.7 seconds when $X = 5$ seconds. The box-and-whiskers indicate the maximum, 75%-, 50%-, and 25%-tiles, and minimum values over 20 different model orderings.

achieves the minimal mean time to prove a theorem of 44.7 seconds, a speed up of 93%, or 15×, compared to not using interrupts.

RA4: Model interrupts is incredibly effective, cutting down the mean time to prove a theorem by up to 97%.

4.6 Threats to validity

The CoqGym benchmark we evaluate our work on has been used by prior evaluations of proof-script synthesis [28, 90] and uses theorems from 122 open-source projects, improving the likelihood that our results generalize. Our analysis focuses on the Coq interactive proof assistant and may not extend to other assistants, such as HOL4 [72] and HOL Light [36]. Transformers have outperformed Bidirectional LSTM in some natural language tasks [23], and may be able to improve Diva's performance beyond what we find here, but they require significantly larger training sets than what is available today in projects written in Coq. Accordingly, future work should explore other neural modeling architectures.

5 RELATED WORK

We now place our research in the context of related work.

Interactive Theorem Provers (ITPs). ITPs, such as Coq [79], Agda [84], Dafny [49], F* [77], Liquid Haskel [83], Mizar [80], Isabelle [61], HOL4 [72], and HOL Light [36] are semi-automated systems for formally proving theorems. We focus on Coq, but our approach is applicable to other ITPs. Coq has been used to build and verify a C compiler [50], an operating system kernel [32], an x86 model [57], a file system [42], distributed protocols [71] and systems [88], a browser [45], and network controllers [33].

Automation for Proof Systems. Heuristic-based search can partially automate ITPs [5, 12, 14, 15]. *Hammers* use external ATPs

to automatically find proofs for ITPs [21]. Classical search algorithms, such as A*, can also search for proofs in HOL4 [30], as can reinforcement-learning-based methods [89]. By contrast, Diva models existing proof scripts, uses native tactics, and proves theorems within the ITP framework.

Software Engineering for Interactive Proof Assistants. Pumpkin Patch generates proof patches when software evolves [67] by learning from a template of a human-written fix to a similar evolution. Unlike Pumpkin Patch, Diva does not require a nearly-working proof script and generates proof scripts from scratch. Pumpkin Pi repairs the proof term of a broken proof and then uses a decompiler to generate a proof script [66]. Pumpkin Pi's proof script consists of predefined tactics, whereas Diva predicts tactics from a learnt model.

iCoq [16] finds failing proof scripts in evolving projects by prioritizing proof scripts affected by a revision. iCoq tracks fine-grained dependencies between Coq definitions, propositions, and proof scripts to narrow down the potentially affected proof scripts. Diva does not require a failing proof script, but our ideas could potentially be used to repair proof scripts. QuickChick [48], a random Coq testing tool, searches for counterexamples to executable theorems and helps a programmer gain confidence that a theorem is correct.

Language Models for Code. Language modeling of source code can detect bugs and generate tests [1, 26, 64]. Modeling code with n-grams can help code completion [39, 40]. Modified n-grams can be used as a cache to capture local dependencies in code [81]. However, such applications have not been applied to ITPs. Applying language models to Coq and HOL4 proof scripts showed that n-gram models outperform recurrent neural networks [37]. Unlike Diva, this approach did not consider the proof state or proof term and does not synthesize complete proof scripts.

Machine Learning in Formal Verification. Machine learning can simplify formal verification: ML4PG helps Coq users construct proof scripts by showing proof scripts of similar theorems [38, 47]. Machine learning can similarly help with premise selection, the task of selecting lemmas that are relevant to a given theorem [3, 43, 86]. NeuroTactic represents theorems and premises with graph neural networks for prediction [51]. GamePad [41] and Proverbot9001 [69] model the proof state in Coq using RNNs. Diva similarly captures the proof state, but unlike GamePad and Proverbot9001, also models the proof script and Gallina proof term for script synthesis. Diva is a generalization of ASTactic [90] and TacTok [28]. These tools use the CoqGym [90] benchmark for evaluation, which is also a learning environment. Large transformer models can be applied to theorem proving in the MetaMath formalization language and the Lean interactive proof assistant [34, 63]. However, these powerful models require much larger training sets than what is available today in Coq projects.

Metaheuristic Search. Metaheuristic-search-based software engineering [35] has been used for developing test suites [55, 85], finding safety violations [4], refactoring [70], project management and effort estimation [8], and automated program repair [2, 46, 87]. In search, low-quality fitness functions can lead to low-quality results, such as, for example, incorrect bug patches [58, 73]. With Diva, the interactive theorem prover provides a strong assurance that the final produced proof script leads to a correct proof, and thus, proof script synthesis is particularly well suited for metaheuristic-search-based methods.

Ensemble Learning. Ensemble learning is the generation and combination of multiple models to make a decision. This is typically used in supervised machine learning tasks [68]. The idea is that weighing and combining several opinions is better than simply choosing a single one. When generating a model to be used in an ensemble learning method, the model should be sufficiently diverse for the ensemble to achieve a desired predictive performance [22], and the individual model's predictive performance should be as high as possible. There are several approaches to generating diverse models, including input manipulation [18], manipulation of the learning algorithm [13, 52, 54], and combinations of strategies.

Ensemble learning methods either have dependent models, where the output of each model affects the generation of the next, or independent models, where each model is constructed independently from the others [68]. Another way to combine classifiers is through stacking [24], which uses a classifier to decide which model to apply to each input. Diva differs from these methods by using independent models in separate searches of the proof script space since the Coq proof assistant serves as an oracle for whether the resulting proof scripts are valid.

6 CONTRIBUTIONS

We have identified a method for using diversity to significantly improve the proving power of proof-script-synthesis tools. We create Diva, implementing our diversity-based approach, which proves 68% more theorems than TacTok and 77% more than ASTactic, two state-of-the-art proof-script-synthesis tools. Diva automatically proves 364 theorems no existing tool has proved. Together with CoqHammer, Diva proves more than a third of all the theorems in our benchmark of 122 open-source projects, the largest fraction to date. Our model interrupts optimization improves Diva's running time by 40×. Along the way we identify a killer app for ensemble learning, by using the theorem prover as an oracle for optimally aggregating learnt model results. Our findings strongly suggest that using diversity for improving automated formal verification is fruitful and warrants further research.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grant no. CCF-1763423, and by Amazon. This work was performed in part using high performance computing equipment obtained under a grant from the Collaborative R&D Fund managed by the Massachusetts Technology Collaborative.

DATA AVAILABILITY

All of our data and source code to reproduce our results are available [27]. Diva is open-source and is available at https://github.com/LASER-UMASS/Diva/.

REFERENCES

[1] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. 2004. N-gram-based detection of new malicious code. In *Annual International IEEE Computer Software and Applications Conference*, Vol. 2. 41–42. https://doi.org/10.1109/CMPSAC.2004.1342667

[2] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering (TSE)* 47, 10 (October 2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

[3] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. 2014. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning* 52, 2 (2014), 191–213. https://doi.org/10.1007/s10817-013-9286-5

[4] Enrique Alba and Francisco Chicano. 2007. Finding safety errors with ACO. In *Conference on Genetic and Evolutionary Computation (GECCO)*. London, England, UK, 1066–1073. https://doi.org/10.1145/1276958.1277171

[5] Peter B Andrews and Chad E Brown. 2006. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic* 4, 4 (2006), 367–395. https://doi.org/10.1016/j.jal.2005.10.002

[6] AWS [n.d.]. AWS Provable Security. https://aws.amazon.com/security/provable-security.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations (ICLR)*. San Diego, CA, USA. https://arxiv.org/abs/1409.0473

[8] Ahilton Barreto, Márcio Barros, and Cláudia Werner. 2008. Staffing a software project: A constraint satisfaction approach. *Computers and Operations Research* 35, 10 (2008), 3073–3089.

[9] BedRock [n.d.]. BedRock Systems Inc. https://bedrocksystems.com.

[10] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A Neural Probabilistic Language Model. *Journal of Machine Learning Research* 3, Feb. (2003), 1137–1155.

[11] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[12] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. 2011. Automatic proof and disproof in Isabelle/HOL. In *International Symposium on Frontiers of Combining Systems*. Springer, 12–27. https://doi.org/10.1007/978-3-642-24364-6_2

[13] Gavin Brown, Jeremy L Wyatt, Peter Tino, and Yoshua Bengio. 2005. Managing diversity in regression ensembles. *Journal of machine learning research (JMLR)* 6, 9 (2005).

[14] Alan Bundy. 1998. A science of reasoning. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 10–17. https://doi.org/10.1007/3-540-69778-0_2

[15] Alan Bundy, Frank Van Harmelen, Christian Horn, and Alan Smaill. 1990. The $O^{Y}S^{T}ER$-$CL^{A}M$ system. In *International Conference on Automated Deduction (CADE)*. Springer, 647–648. https://doi.org/10.1007/3-540-52885-7_123

[16] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. ICoq: Regression proof selection for large-scale verification projects. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana-Champaign, IL, USA, 171–182. https://doi.org/10.1109/ASE.2017.8115630

[17] Certora [n.d.]. Certora. https://www.certora.com.

[18] Philip K Chan and Salvatore J Stolfo. 1995. A comparative evaluation of voting and meta-learning on partitioned data. In *Machine Learning Proceedings*. Elsevier, 90–98.

[19] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[20] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Deep Learning and Representation Learning Workshop (DL&RL)*. http://arxiv.org/abs/1412.3555

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

[22] Houtao Deng, George Runger, Eugene Tuv, and Martyanov Vladimir. 2013. A time series forest for classification and feature extraction. *Information Sciences* 239 (2013), 142–153.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. Minneapolis, MN, USA, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[24] Saso Džeroski and Bernard Ženko. 2004. Is combining classifiers with stacking better than selecting the best one? *Machine learning* 54, 3 (2004), 255–273.

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

[26] Michael D. Ernst. 2017. Natural Language is a Programming Language: Applying Natural Language Processing to Software Development. In *Summit on Advances in Programming Languages (SNAPL)*, Vol. 71. Dagstuhl, Germany, 4:1–4:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.4

[27] Emily First and Yuriy Brun. 2022. Replication package for "Diversity-Driven Automated Verification". https://doi.org/10.5281/zenodo.5903318.

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

[29] Galois [n.d.]. Galois, Inc. https://galois.com.

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

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[32] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

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

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

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

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

[37] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. 2018. On the naturalness of proofs. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results track*. Orlando, FL, USA, 724–728.

[38] Jónathan Heras and Ekaterina Komendantskaya. 2014. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science* 8, 1 (2014), 99–116. https://doi.org/10.1007/s11786-014-0173-1

[39] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Communications of the ACM (CACM)* 59, 5 (April 2016), 122–131. https://doi.org/10.1145/2902362

[40] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. 837–847. https://doi.org/10.1109/ICSE.2012.6227135

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

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

[43] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems (NeurIPS)*. Barcelona, Spain, 2235–2243. https://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection

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

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

[46] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (9–13). Lincoln, NE, USA, 295–306. https://doi.org/10.1109/ASE.2015.60

[47] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. 2012. Machine learning in proof general: Interfacing interfaces. In *International Workshop on User Interfaces for Theorem Provers (UITP)*, Vol. 118. Bremen, Germany. https://doi.org/10.4204/EPTCS.118.2

[48] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proceedings of the ACM*

on Programming Languages (PACMPL) 2, POPL (Dec. 2017), 45:1–45:30. https://doi.org/10.1145/3158133

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

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

[51] Zhaoyu Li, Binghong Chen, and Xujie Si. 2021. Graph Contrastive Pre-training for Effective Theorem Reasoning. In *International Conference on Machine Learning (ICML)*, Vol. PLMR 139. http://arxiv.org/abs/2108.10821

[52] Shih-Wei Lin and Shih-Chieh Chen. 2012. Parameter determination and feature selection for C4.5 algorithm using scatter search approach. *Soft Computing* 16, 1 (2012), 63–75.

[53] Laurent Mauborgne. 2004. AstrÉe: Verification of Absence of Runtime Error. In *Building the Information Society*. 385–392. https://doi.org/10.1007/978-1-4020-8157-6_30

[54] Jesús Maudes, Juan J Rodríguez, and César García-Osorio. 2009. Disturbing neighbors diversity for decision forests. In *Applications of supervised and unsupervised ensemble methods*. Springer, 113–133.

[55] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. 2001. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering (TSE)* 27, 12 (Dec. 2001), 1085–1110. https://doi.org/10.1109/32.988709

[56] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent Neural Network Based Language Model. In *Annual Conference of the International Speech Communication Association (INTERSPEECH)*. Makuhari, Chiba, Japan. https://doi.org/10.1109/IALP.2016.7875937

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

[58] Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2021. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering (TSE)* (2021). https://doi.org/10.1109/TSE.2020.2998785 DOI: 10.1109/TSE.2020.2998785.

[59] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy (S&P)*. San Francisco, CA, USA, 415–429.

[60] David F. Nettleton, Albert Orriols-Puig, and Albert Fornells. 2010. A study of the effect of different types of noise on the precision of supervised learning techniques. *Artificial Intelligence Review* 33 (2010), 275–306. https://doi.org/10.1007/s10462-010-9156-z

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

[62] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, Vol. 1. Association for Computational Linguistics, New Orleans, LA, USA, 2227–2237. https://doi.org/10.18653/v1/N18-1202

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

[64] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. Austin, TX, USA, 428–439. https://doi.org/10.1145/2884781.2884848

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

[66] Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof Repair Across Type Equivalences. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)* (20–26). 112–127. https://doi.org/10.1145/3453483.3454033

[67] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. Los Angeles, CA, USA, 115–129. https://doi.org/10.1145/3167094

[68] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8, 4 (2018), e1249.

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

[70] Olaf Seng, Johannes Stammel, and David Burkhart. 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and Evolutionary Computation (GECCO)*. Seattle, WA, USA, 1909–1916. https://doi.org/10.1145/1143997.1144315

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

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

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

[74] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1631–1642. https://www.aclweb.org/anthology/D13-1170

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

[76] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM Neural Networks for Language Modeling. In *Annual Conference of the International Speech Communication Association (INTERSPEECH)*. Portland, OR, USA. https://doi.org/10.21437/Interspeech.2012-65

[77] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Vol. 51. St. Petersburg, FL, USA, 256–270. https://doi.org/10.1145/2914770.2837655

[78] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, Vol. 1. Beijing, China, 1556–1566. https://doi.org/10.3115/v1/P15-1150

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

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

[81] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Hong Kong, China, 269–280. https://doi.org/10.1145/2635868.2635875

[82] Brendan van Rooyen, Aditya Menon, and Robert C Williamson. 2015. Learning with Symmetric Label Noise: The Importance of Being Unhinged. In *Advances in Neural Information Processing Systems*, Vol. 28. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2015/file/45c48cce2e2d7fbdea1afc51c7c6ad26-Paper.pdf

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

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

[85] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*. Portland, ME, USA, 1–12. https://doi.org/10.1145/1146238.1146240

[86] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. 2017. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems (NeurIPS)*. Long Beach, CA, USA, 2786–2796. https://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding

[87] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. Vancouver, BC, Canada, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

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

[89] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. 2021. TacticZero: Learning to Prove Theorems from Scratch with Deep Reinforcement Learning. *CoRR* abs/2102.09756 (2021). http://arxiv.org/abs/2102.09756

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

[91] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, Vol. 1. Vancouver, BC, Canada, 440–450. https://doi.org/10.18653/v1/P17-1041