TacTok: Semantics-Aware Proof Synthesis

EMILY FIRST, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
ARJUN GUHA, University of Massachusetts Amherst, USA

Formally verifying software correctness is a highly manual process. However, because verification proof scripts often share structure, it is possible to learn from existing proof scripts to fully automate some formal verification. The goal of this paper is to improve proof script synthesis and enable fully automating more verification. Interactive theorem provers, such as the Coq proof assistant, allow programmers to write partial proof scripts, observe the semantics of the proof state thus far, and then attempt more progress. Knowing the proof state semantics is a significant aid. Recent research has shown that the proof state can help predict the next step. In this paper, we present TacTok, the first technique that attempts to fully automate proof script synthesis by modeling proof scripts using both the partial proof script written thus far and the semantics of the proof state. Thus, TacTok more completely models the information the programmer has access to when writing proof scripts manually. We evaluate TacTok on a benchmark of 26 software projects in Coq, consisting of over 10 thousand theorems. We compare our approach to five tools. Two prior techniques, CoqHammer, the state-of-the-art proof synthesis technique, and ASTactic, a proof script synthesis technique that models proof state. And three new proof script synthesis technique we create ourselves, SeqOnly, which models only the partial proof script and the initial theorem being proven, and WeightedRandom and WeightedGreedy, which use metaheuristic search biased by frequencies of proof tactics in existing, successful proof scripts. We find that TacTok outperforms WeightedRandom and WeightedGreedy, and is complementary to CoqHammer and ASTactic: for 24 out of the 26 projects, TacTok can synthesize proof scripts for some theorems the prior tools cannot. Together with TacTok, 11.5% more theorems can be proven automatically than by CoqHammer alone, and 20.0% than by ASTactic alone. Compared to a combination of CoqHammer and ASTactic, TacTok can prove an additional 3.6% more theorems, proving 115 theorems no tool could previously prove. Overall, our experiments provide evidence that partial proof script and proof state semantics, together, provide useful information for proof script modeling, and that metaheuristic search is a promising direction for proof script synthesis. TacTok is open-source and we make public all our data and a replication package of our experiments.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Additional Key Words and Phrases: Formal software verification, Coq, proof script synthesis, automated proof script synthesis

ACM Reference Format:
Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-Aware Proof Synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 231 (November 2020), 31 pages. https://doi.org/10.1145/3428299

Authors' addresses: Emily First, efirst@cs.umass.edu, University of Massachusetts Amherst, 140 Governors Dr., Amherst, Massachusetts, 01003-9264, USA; Yuriy Brun, brun@cs.umass.edu, University of Massachusetts Amherst, 140 Governors Dr., Amherst, Massachusetts, 01003-9264, USA; Arjun Guha, arjun@cs.umass.edu, University of Massachusetts Amherst, 140 Governors Dr., Amherst, Massachusetts, 01003-9264, USA.

2475-1421/2020/11-ART231
https://doi.org/10.1145/3428299

1 INTRODUCTION

Building high-assurance software in a cost-effective manner remains one of the greatest challenges for system-building research. Most programming languages make it difficult to ensure even basic properties, such as the absence of unhandled exceptions. Moreover, industrial verification tools are either unsound by design [Bessey et al. 2010] or place severe restrictions on the programming language to ensure that programs are verifiable [Mauborgne 2004]. Recently, there has been a growing interest in building high-assurance software using programming languages that are designed to support program verification from the ground up [Leino 2010; Swamy et al. 2016; The Coq Development Team 2017; Vazou 2016]. This paper focuses on Coq, which has become a popular language for building verified software systems (e.g., [Gu et al. 2016; Guha et al. 2013; İleri et al. 2018; Jang et al. 2012; Leroy 2009; Morrisett et al. 2012; Sergey et al. 2017; Wilcox et al. 2015]).

At its core, 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. Programmers must prove theorems themselves using a sequence of *proof tactics* and it is up to the programmer to select the right sequence of these tactics. Programmers use an *interactive proof assistant* (e.g., CoqIDE or Proof General) to write *proof scripts*, which provides immediate feedback after each tactic. The proof assistant shows the programmer the current subgoals and available hypotheses, and the programmer uses this information to select the next tactic to try. The proof assistant checks that the chosen tactic is valid and then updates the subgoals and hypotheses that are displayed to the programmer.

Unsurprisingly, writing proof scripts in Coq is a difficult exercise, which can be painstaking and requires deep expertise, e.g., [Jacky et al. 2017; Wilcox et al. 2015]. Moreover, once a system is verified, unless great care is taken, even a small change to the program can require a complete reworking of the proof script [Chlipala 2013]. Researchers have developed a variety of tools and techniques to make writing proof scripts in Coq (and other interactive theorem provers) easier to write [Hellendoorn et al. 2018; Heras and Komendantskaya 2014; Huang et al. 2018; Komendantskaya et al. 2012; Yang and Deng 2019]. Inspired by the success of statistical code completion tools [Hindle et al. 2016, 2012; Vechev and Yahav 2016], Hellerdoorn et al. recently observed that the sequence of tokens in Coq proofs is somewhat predictable [Hellendoorn et al. 2018], and this could hypothetically be used to build statistical proof script completion tools. Other researchers have used the current proof state to build proof script synthesis tools [Huang et al. 2018; Yang and Deng 2019]. For example, ASTactic [Yang and Deng 2019], a deep learning model, takes as input the current goal, local context, and environment, and predicts the next step of the proof script.

In this paper, we observe that when programmers use interactive theorem provers, they use both the feedback from the theorem prover — the proof state — and the partial proof script already written. We argue that models that learn from the proof state and the partial proof script, together, can be more effective at synthesizing proof scripts. When programmers write Coq proof scripts themselves, they critically rely on the feedback detailing the internal proof state provided by the interactive proof assistant to choose the next proof tactic to try. In fact, writing proof scripts without the proof assistant's feedback is often *impossible*. For example, Coq has several sophisticated proof tactics that can make partial progress that is hard to predict. (Section 2.2 shows several concrete examples.) Moreover, even some simple tactics automatically generate new variable names that programmers need to know for subsequent tactics, and thus they rely on the proof assistant to learn these generated names. In a way, the programmer and the proof assistant are engaged in a conversation. However, the existing methods of predicting the next tactic from the internal proof state alone [Yang and Deng 2019] and predicting the next tactic from the previous tactics alone [Hellendoorn et al. 2018] only consider distinct sides of the conversation. In this paper, we show that by carefully modeling and combining the partial proof script already written and the proof assistant's feedback, we can automatically prove different theorems than models that only use one source of input.

We present TacTok, an open-source implementation of our approach to proof script synthesis — available at https://github.com/LASER-UMASS/TacTok — that incorporates both proof state and the partial proof script already written. We apply our approach to a benchmark of 122 open-source software projects in Coq, with over 68K theorems. We demonstrate that the inclusion of both proof state and the partial proof script in a proof synthesis model can improve its effectiveness and allow it to synthesize different proof scripts than other tools.

We compare our approach to five approaches, two existing tools, CoqHammer and ASTactic, and three new ones we create, SeqOnly, WeightedRandom, and WeightedGreedy. CoqHammer [Czajka and Kaliszyk 2018] is a state-of-the-art proof synthesis technique and ASTactic [Yang and Deng 2019] is a proof script synthesis technique that models only proof state. SeqOnly is a proof script synthesis technique that models only the partial proof script and the initial theorem, and WeightedRandom and WeightedGreedy are proof script synthesis techniques that attempt to construct a proof script via metaheuristic search biased by using the frequency distribution of tactics as they appear in the training data.

All of the above techniques, except CoqHammer, are metaheuristic search techniques [Harman 2007] that search through the proof-script space using a model that biases the search toward the likely proof script steps. TacTok's model is built by training on a set of existing, successful proof scripts, and models proof state and the partial proof script already written. ASTactic [Yang and Deng 2019], an existing technique, uses a similar metaheuristic search strategy, but its model only models the proof state. We develop three other novel metaheuristic search techniques for proof synthesis generation: SeqOnly, whose model accounts for the partial proof state so far and the target theorem being proven, and WeightedRandom and WeightedGreedy, whose models account for the frequency with which proof script tactics occur in the existing successful proof scripts, with WeightedRandom selecting randomly adhering to the observed distribution, and WeightedGreedy always selecting the most frequently observed candidates.

Evaluated on a benchmark of 26 software projects in Coq with 10,782 theorems, we find that each of these techniques can successfully generate proof scripts that prove theorems, which is strong evidence for the success of metaheuristic search in proof script synthesis. TacTok can automatically prove more theorems than ASTactic (12.9% versus 12.3%). Both techniques outperform SeqOnly, which can only prove 10.0% of the theorems. Even without explicitly modeling the theorem being proven, WeightedRandom proves 6.2% of the theorems and WeightedGreedy proves 9.7% of the theorems. Importantly, TacTok is complementary to prior tools. When used together with ASTactic, TacTok can prove 20% more theorems than ASTactic alone. CoqHammer, a very different approach, is able to synthesize proofs for more theorems than TacTok (26.6%), but TacTok and CoqHammer together can prove 11.5% more theorems than CoqHammer alone.

Overall, our experiments show that there is promise to using metaheuristic search and modeling the partial proof scripts and proof state, together, for synthesizing proof scripts for formal verification. We investigate the effect of using various kinds of information encoded in the partial proof script on verification efficacy and produce guiding information for improving future proof script synthesis tools.

The main contributions of this work are:

- A novel automated proof script synthesis technique, TacTok, that, unlike prior work, models the combination of the partial proof script already written and the proof state to automate proof script synthesis and formal verification. TacTok is open-source.
- Three other new metaheuristic search proof script synthesis techniques, SeqOnly, WeightedRandom, and WeightedGreedy.
- An application of ASTactic, a state-of-the-art proof script synthesis tool, independently reproducing prior evaluation results [Yang and Deng 2019].
- An evaluation of the value added by using both the partial proof script and proof state over using each one alone, and a comparison to two state-of-the-art proof synthesis tools and three new tools, showing that TacTok proves theorems no prior tool can prove. A public release of TacTok — https://github.com/LASER-UMASS/TacTok — and our experimental scripts and data [First et al. 2020].
- An exploration of the information TacTok models to improve verification efficacy.

The rest of this paper is structured as follows. Section 2 illustrates our approach on a simple example. Section 3 details our TacTok approach, Section 4 describes the state of the art of formal verification and the three other proof script synthesis approaches we create, and Section 5 evaluates all these approaches. Finally, Section 6 places our work in the context of related research, and Section 7 summarizes our contributions.

2 ILLUSTRATIVE EXAMPLE

To illustrate our approach, Section 2.1 introduces a small Coq program for adding two numbers and proves that the function is associative, and Section 2.2 shows how TacTok would generate such a proof script automatically. Section 2.3 introduces a more complex theorem expressed in higher-order logic that TacTok is able to prove, whereas prior tools do not.

2.1 Using Coq to Prove Addition Is Associative

We demonstrate Coq's use with a small, simple example. We define the natural numbers and then prove that addition is associative.[1] We define the natural numbers (`nat`) using a unary encoding (Figure 1, lines 1–3), where `Z` is the natural number `0`, and the rest are defined using the successor operator (`S n`).

Lines 5–9 in Figure 1 define a recursive function to add two numbers (`add m n`). Line 10 defines the notation `m + n` as a shorthand for `add m n`. The function itself maps the answer to `m` if `n = 0` (line 7), and otherwise to `S (add p m)`, where `S p = n` (line 8). In other words, the recursive step returns the successor of the sum of `m` and the number whose successor is `n`.

The `assoc` Theorem (lines 1–3) defines the associativity property, `n + (m + p) = (n + m) + p`, for all `n, m, p`.

To prove this property, Coq needs help. Coq maintains an internal *proof state*, which includes the goals that Coq wants to prove and a list of assumptions. When starting to prove a theorem, Coq's proof state is a single *goal*: the theorem itself (shown in Figure 2a). Coq manipulates the proof state (with the help of proof tactics, described next) until the goal is proven and can be removed from the proof state. These manipulations are enabled by *proof tactics*. Some tactics create new subgoals (e.g., **`induction`**) and other tactics manipulate the current goal in various ways (e.g., **`simpl`**, which simplifies complex terms in the goal, and **`rewrite`**, which transforms a term in the goal into an equivalent term). The search space of goal manipulation is too large, and a human helps Coq by

[1]While Coq's standard library already includes these definitions, this basic example is simple enough to effectively demonstrate our approach.

writing a sequence of proof tactics that help Coq manage the exploration. Lines 14–21 are the human-written sequence of tactics, one per line, that help Coq prove associativity.

To write down the appropriate sequence of tactics, the programmer needs to understand Coq's internal proof state. The interactive proof assistant checks that a partially-complete proof script is valid and shows the current proof state to the programmer. This makes it possible for the programmer to incrementally develop a proof script, instead of writing a complete proof script in a single step. The programmer can choose a tactic, examine the output from the proof assistant, choose the next tactic, and so on. If the programmer chooses an invalid tactic, the proof assistant displays an error. (For example, the **induction** tactic signals an error if its argument is not inductively defined.) 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.

We now step through the proof script of associativity and discuss both the proof tactic entered by the programmer and the proof state displayed by the proof assistant. At the start of the proof script, the proof state has a single goal, which is the theorem itself (Figure 2a). Since the goal is a statement that should hold **forall** naturals `n`, `m`, and `p`, the natural next step is to assume the existence of some arbitrary `n`, `m`, and `p`. The **intros** tactic eliminates the **forall** quantifier and introduces three new variables as *assumptions* (line 5 in Figure 2b). Since add is inductively defined, it is natural to perform induction on one of these variables (e.g., **induction** `n`), which leads to two subgoals, one for the base case and the other for the inductive case (Figure 2c). Coq first focuses on proving the base case (line 11 in Figure 2c) by labeling it subgoal 1 of 2 (line 10) and only presenting assumptions (line 9) relevant for the base case. Note that in the base case, `n` is no longer in the list of assumptions, and has been replaced with the value `0` in the subgoal (line 11 in Figure 2c). The **reflexivity** tactic does some basic simplification and solves the base case, since both sides are essentially identical. Therefore, the next proof state has just one goal, which is the inductive case (Figure 2d). In this state, `n` has been replaced with `S n`, and the proof assistant shows the inductive

```
1  Inductive nat : Set :=
2    | O : nat
3    | S : nat -> nat.
4
5  Fixpoint add (n : nat) (m : nat) : nat :=
6    match n with
7      | O -> m
8      | S p -> S (add p m)
9    end
10   where "n + m" := (add n m) : nat_scope.
11
12 Theorem assoc : forall n m p : nat,
13     n + (m + p) = (n + m) + p.
14 Proof.
15   intros.
16   induction n.
17   reflexivity.
18   simpl.
19   rewrite -> IHn.
20   reflexivity.
21 Qed.
```

Fig. 1. The definition of the natural numbers (lines 1–3), the function to add unary natural numbers (lines 5–10), a theorem that the function is associative (lines 12–13), and a proof script of that theorem (lines 14–21).

```
1  1 subgoal
2  ______________________________________(1/1)
3  forall n m p : nat, n + (m + p) = n + m + p
```

(a) Proof state at the start of the proof script (after line 14 in Figure 1).

```
4  1 subgoal
5  n, m, p : nat
6  ______________________________________(1/1)
7  n + (m + p) = n + m + p
```

(b) Proof state after line 15 in Figure 1 (**intros** tactic).

```
8  2 subgoals
9  m, p : nat
10 ______________________________________(1/2)
11 0 + (m + p) = 0 + m + p
12 ______________________________________(2/2)
13 S n + (m + p) = S n + m + p
```

(c) Proof state after line 16 in Figure 1 (**induction** n tactic).

```
14 1 subgoal
15 n, m, p : nat
16 IHn : n + (m + p) = n + m + p
17 ______________________________________(1/1)
18 S n + (m + p) = S n + m + p
```

(d) Proof state after line 17 in Figure 1 (**reflexivity** tactic).

```
19 1 subgoal
20 n, m, p : nat
21 IHn : n + (m + p) = n + m + p
22 ______________________________________(1/1)
23 S (n + (m + p)) = S (n + m + p)
```

(e) Proof state after line 18 in Figure 1 (**simpl** tactic).

```
24 1 subgoal
25 n, m, p : nat
26 IHn : n + (m + p) = n + m + p
27 ______________________________________(1/1)
28 S (n + m + p) = S (n + m + p)
```

(f) Proof state after line 19 in Figure 1 (**rewrite** -> IHn tactic).

Fig. 2. The proof state after the execution of each tactic of the add function's associativity proof script. For example, in (a), at the start of the proof script, there is 1 subgoal, which is the theorem itself, line 3, and in (c), after the **induction** n tactic, there are 2 subgoals, one for the base case (line 11) and one for the inductive case (line 13).

assumption (IHn). However, IHn is not immediately applicable because it contains n instead of S n, and so trying to use it will produce an error. Instead, the **simpl** tactic tries to simplify the goal while keeping it readable (though many Coq users find that **simpl** is a complicated tactic that produces unpredictable results [The Coq Development Team 2017]). After choosing **simpl**, the left-hand side of the goal has an expression that is identical to the left-hand side of the inductive assumption (Figure 2e). Therefore, the **rewrite** tactic can replace that expression with the right-hand side of the inductive assumption (Figure 2f), which leads to a trivial equality that **reflexivity** can solve. Finally, the proof assistant prints *no more subgoals* and **Qed** completes the proof script.

2.2 Proving Associativity with TacTok

Note that the proof script (lines 15–20 of Figure 1) is almost unintelligible without examining the internal proof state, for several reasons:

(1) Some tactics generate new variable names (e.g., **induction**), which subsequent tactics (e.g., **rewrite**) use. The programmer needs to know these names to use them in the proof script.
(2) Some tactics apply heuristics that are hard to predict. In the example in Figure 1, **simpl** (line 18) uses heuristics to produce readable output. Without looking at the resulting proof state in Figure 2e, the programmer would not know what form the goal is transformed into and whether `IHn` can be invoked without an error.
(3) When a proof script has multiple goals, it may not be obvious where one goal ends and the next goal begins. In the example in Figure 1, the base case is addressed by the tactic in line 17, while the inductive case is addressed by the tactics in lines 18–20. However, without examining the proof state Figure 2d, it would not be clear that **reflexivity** completely solved the base case.
(4) After a few steps of the proof script, the current goal to prove has likely evolved significantly from the original goal (the theorem). In the example in Figure 2, the programmer is trying to prove the theorem Figure 2a line 3. After three steps in the proof script, the programmer must now prove Figure 2d line 18 using assumptions lines 15–16. The goal has changed significantly and there are now assumptions that didn't exist at the start. Knowing the new goal and the assumption `IHn` are helpful to the programmer at this point in deciding the next tactic: the programmer sees that the goal needs to be transformed using **simpl** to apply the assumption. Without being able to see the current proof state in Figure 2d, the programmer would be proving blindly.

Since the programmer needs to examine the proof state to choose the next tactic, it is a good idea to consider whether models of next tactic prediction might also need and benefit from having access to information in the proof state. However, since proof scripts, like programs, build upon the commands executed thus far [Hellendoorn et al. 2018; Hindle et al. 2016, 2012], a model of next tactic prediction may also benefit from having access to the previous tactics in the proof script.

TacTok builds a model of next-tactic prediction for Coq. The model is learnt using a corpus of existing proof scripts. TacTok decomposes each of the existing proof scripts by stepping through them, one tactic at a time, and computing the resulting intermediate proof states. The model automatically learns an embedding of the proof states and the partially written proof script and maps these to an abstract syntax tree (AST) of the next line in the proof script. Thus, the model's inputs are the partially written proof script and the proof state. For example, one input to the model is the intermediate proof state in Figure 2d and the partial proof script [**intros**, **induction** `n`, **reflexivity**] that achieved that proof state. The model's output is an AST that TacTok decodes to the next predicted tactic and its arguments.

Given a model trained on existing proof scripts (Section 3.2.4 presents the training process), TacTok automatically generates the proof script of associativity (lines 14–21). To start, TacTok takes as input the proof state in Figure 2a and the partially written proof script, which is only [**Proof**]. For the sake of illustration, suppose the model correctly predicts **intros** as the next tactic. TacTok executes this partial proof script using the Coq interactive proof assistant, and checks for two things. First, that executing the partial proof script with the new tactic does not return an error. And second, that the new returned proof state is different from the prior observed proof states in this partial proof script, meaning the tactic had an affect and produced something new. Here, TacTok will see no error and the updated proof state is that in Figure 2b. TacTok will next explore growing the proof script further. TacTok's next input to the model is the proof state in Figure 2b and the partial proof script [**Proof**, **intros**]. Suppose, again, the model correctly predicts the next tactic and argument is **induction** `n`; there is no error adding this tactic to the partial proof script and the new proof state is that in Figure 2c.

If TacTok proceeds in this way to produce the partial proof script [**Proof**, **intros**, **induction** `n`, **reflexivity**] and the proof state in Figure 2d, but as the next tactic and arguments, TacTok predicts **rewrite** `->` `IHn`. Adding this tactic results in an error. TacTok goes back to the model and asks it for the next most likely tactic and arguments (TacTok would do the same thing if there were no error but the proof state was a duplicate of an earlier state). Suppose the model suggests **simpl**, which turns out to return no error and changes the state.

TacTok will proceed in this way, performing a depth-first search through the space of proof scripts until, either, it reaches the proof state "no more subgoals" (meaning the proof script is completed successfully) and adds **Qed** to the proof script, or it times out and fails to complete the proof script.

Of course, TacTok can only be successful if its model is fairly accurate in its predictions.

2.3 Proving Higher-Order Logic Theorems

The associativity theorem we just discussed is an example of a theorem expressed in first-order logic. Theorems expressed in higher-order logic are likely to be more complex, and so their proof scripts are more difficult to generate. One example of higher-order logic in Coq is the presence of a nested **forall**. Figure 3 defines the function `prod n f`, which calculates $1 \cdot f(0) \cdot \cdots \cdot f(n-1)$. The same figure defines the `sameProd` theorem, which states an intuitive property: for functions f and g, if $f(m) = g(m)$ for all $m < n$, then $prod\ n\ f = prod\ n\ g$. TacTok is able to generate a proof script for `sameProd`. CoqHammer [Czajka and Kaliszyk 2018] fails to find a proof in ten minutes, and ASTactic [Yang and Deng 2019] cannot find a proof script either. Section 5.5 will further discuss theorems that TacTok proves that prior tools cannot.

We next describe prior work on training language models and our improvements over the state of the art.

```
1  Definition prod : forall (n : nat) (f : nat -> nat), nat.
2  intros.
3  induction n as [| n Hrecn].
4  exact 1.
5  exact (f n * Hrecn).
6  Defined.
7
8  Lemma sameProd : forall (n : nat) (f g : nat -> nat),
9          (forall m : nat, m < n -> f m = g m) ->
10         prod n f = prod n g.
```

Fig. 3. Lines 1–6 defines the function $prod\ n\ f$, which calculates $1 \cdot f(0) \cdot \cdots \cdot f(n-1)$. Lines 8–10 define the `sameProd` theorem, which states that if $f(m) = g(m)$ for all $m < n$, then $prod\ n\ f = prod\ n\ g$.

3 THE TacTok APPROACH

Section 3.1 describes language modeling methods and Section 3.2 details how TacTok builds on language modeling approaches to generate proof scripts. Our TacTok implementation is publicly available at https://github.com/LASER-UMASS/TacTok/.

3.1 Language Modeling

Language models estimate the probability of a particular instance in a language. For example, a language model can estimate the likelihood that the words "I went to the" are followed by "store". When applied to natural languages, these models have aided speech recognition, machine translation, spelling correction, and other natural language processing tasks [Brill and Moore 2000; Koehn et al. 2007; Stolcke 2002].

While language models can be created in many ways, including manually, a typical approach is to train a language model on a large corpus of example sentences in a language. There are two main classes of language models widely used in modern natural language processing research. The first consists of statistical language models, often called n-gram language models. These count-based models work on the Markov assumption that the conditional probability of a word is dependent on a fixed number (n) of previous words in a sentence. The second class of models are based on neural network architectures. Neural models often perform better on natural language applications than statistical models, though are less human-readable [Mikolov et al. 2010; Sundermeyer et al. 2012].

3.1.1 Statistical Language Models. N-gram models predict the next word based on a sequence of the previous $n-1$ words [Katz 1987]. Given a vocabulary of words, during training, an n-gram model counts the number of times in the training corpus that each word follows each possible sequence of $n-1$ words. At prediction time, given a sequence of $n-1$ words, the model returns the probability distribution of words that follow that sequence. Formally, given a sequence S of $n-1$ words, $S = \langle w_1, w_2, \ldots, w_{n-1}\rangle$, the n-gram model estimates $P(w_n|w_1, w_2, \ldots, w_{n-1})$, the probability distribution of words in the nth place.

The maximum likelihood estimate for this probability, $P_{ML}(w_n|w_1, w_2, \ldots, w_{n-1})$, is the number of times $\langle w_1, w_2, \ldots, w_n\rangle$ appears in the training corpus, divided by the number of times $\langle w_1, w_2, \ldots, w_{n-1}\rangle$ appears in the corpus.

$$P_{ML}(w_n|w_1, w_2, \ldots, w_{n-1}) = \frac{count(w_1, w_2, \ldots w_n)}{count(w_1, w_2, \ldots, w_{n-1})}$$

This can be used as an estimate but because training data may not contain every possible sequence $\langle w_1, w_2, \ldots, w_n\rangle$, it would end up with a lot of zero probability estimates. So rather than underestimate all $P(w_n|w_1, w_2, \ldots, w_{n-1})$ as 0, linear interpolation [Stolcke 2002] uses smaller subsequences of length 1 to $n-1$ to estimate the probability:

$$P(w_n|w_1, w_2, \ldots, w_{n-1}) = \sum_{i=1}^{n} \lambda_i \times P_{ML}(w_n|w_{n-i+1}, \ldots, w_{n-1})$$

where λ_i is a normalized weight given to the value $P_{ML}(w_n|w_{n-i+1}, \ldots, w_{n-1})$. These weights can be considered as *fallback* weights to fallback and look at smaller subsequences if the larger subsequence is not present in the training corpus.

Discounting smooths the probability distribution over the vocabulary by reallocating the probability mass from the training dataset by subtracting a fixed discount d from the counts of each word in the training dataset and then reassigning it to the fallback probability. The state-of-the-art n-gram language model is the Modified Kneser-Ney model [Kneser and Ney 1995], which implements a more involved discounting technique.

As described in Section 1, recent work has applied n-gram models (as well as other models) to Coq proof scripts to predict tokens, which can, in theory, be used to build statistical proof script completion tools.

3.1.2 Neural Language Models. Feed-forward neural networks [Rumelhart et al. 1986] can also model language. These models are also trained to predict the next word given a sequence of previous words but they do this using a different architecture than that of n-gram models.

Given the previous $n-1$ words $\langle w_1, \ldots, w_{n-1}\rangle$, these models first create real-valued vectors $\langle r_1, \ldots, r_{n-1}\rangle$ that represent the words. This is commonly done by converting each of the words into a one-hot vector representation and then passing them through a linear embedding layer. A one-hot vector representation is constructed by first initializing a vector of the size of vocabulary V of the language and then setting the element at the index corresponding to the given word to 1. The linear embedding layer consists of $|V|$ weights W_e and a bias term b_e. The model then concatenates the real valued embeddings to form a summary vector r. The model passes r through a predetermined number of neural network layers, together called NN. The output $h = NN(r)$ passes through a softmax layer to produce a probability distribution over the entire vocabulary. In summary, the training algorithm is:

```
$o_i = onehot(w_i)$
$r_i = o_i.W_e + b_e$
$r = \oplus(r_1 \ldots r_{n-1})$
$h = NN(r)$
$P = softmax(h)$
```

Recently, recurrent neural network (RNN) architectures have been shown to perform well in language modeling tasks [Bengio et al. 2003; Mikolov et al. 2010; Sundermeyer et al. 2012]. The main difference between this approach from a simple feed-forward neural network is the way the sequence of real valued vector representations, $\langle r_1, \ldots, r_{n-1}\rangle$, are handled to obtain the final hidden layer representation h. An RNN performs a series of steps that each process the sequence up until an index. A simplified version of this process can be described as follows. At each index i of the sequence, an RNN maintains a hidden state h_i, which is a vector representing the summary of the real valued vectors before index i. It then concatenates h_i with r_i and sends the resultant vector through a linear neural network layer with weights W_r and W_h and bias b_r. The output is used as the new summary and hidden state at index $i+1$, h_{i+1}. The summary of the whole sequence is hence the hidden state after processing all the $n-1$ real valued vectors. Note that h_1 is a zero vector since there are no elements to summarize before the first element in the sequence. Following those steps, we obtain h and then, a softmax layer converts this into the probability distribution over the vocabulary. In summary, the RNN training algorithm is:

```
$o_i = onehot(w_i)$
$r_i = o_i.W_e + b_e$
$h_{i+1} = RNN(r_i, h_i) = r_i.W_r + h_i.W_h + b_r$
$h = RNN(r_{n-1}, h_{n-1})$
$P = softmax(h)$
```

Long short-term memory networks (LSTMs) extend recurrent neural networks to learn long-term dependencies by tackling a problem that arises with basic RNNs called the vanishing gradient problem [Gers et al. 1999; Greff et al. 2017]. For long sequences, the RNN framework scales down the gradient over each word going backwards and the value of the gradient become negligible, making it difficult for the model to learn. To address this problem, LSTMs redesign the basic building block of the neural network to include components called *gates*. At each step in the sequence, the model makes decisions based on three gates: the input gate, the output gate, and the forget gates. These gates control whether the model should remember or forget the summary of the sequence so far

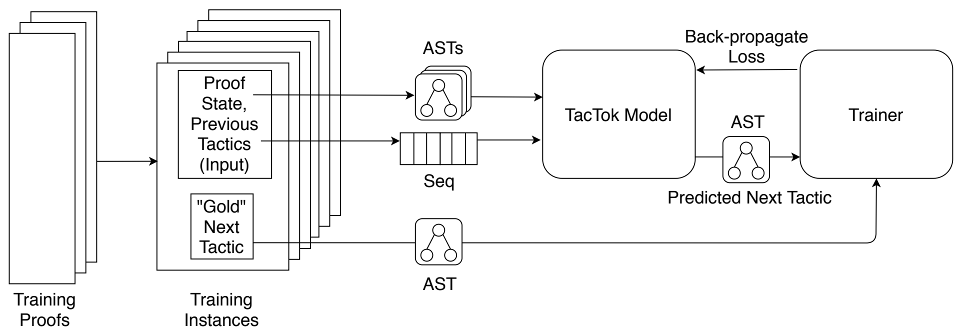


Fig. 4. TacTok training process. Given a set of proof scripts, TacTok breaks down each proof script into training instances, consisting of the input to the model (proof state and the proof script thus far up to this state), and the next step of the proof script. The TacTok model jointly learns embeddings for the proof state ASTs and the proof script thus far, and uses these embeddings to predict the next tactic in the form of its AST. The learning process back-propagates the difference between the predicted tactic AST and the (expected) next step as the loss.

through a set of parameters that are learned jointly with the parameters of the RNN during training. Note that the hidden state h is still calculated sequentially like a normal RNN, except with the model choosing to ignore the summary vector sometimes. The exact implementation of the gates can be found in literature and are beyond the scope of this paper.

A Bidirectional LSTM [Peters et al. 2018] runs the sequence input in two ways, backwards and forward, allowing the output layer to see both directions of the information simultaneously. This improves upon the LSTM by capturing more information.

Transformers [Devlin et al. 2018] are the current state-of-the-art in neural language models, but they only perform well when there is a very large amount of data for training. Thus, LSTMs and variants like gated recurrent neural networks (GRUs) [Chung et al. 2014] are sufficient for neural language models that do not have millions of sequences for training. They have been shown to do well in machine translation and sentiment analysis [Bahdanau et al. 2015; Socher et al. 2013].

While RNN language models use sequences of words to predict the next word in the sequence, RNNs in general can be used to model other types of sequential objects for any prediction task. In particular, we are interested in using sequential data — sequences of proof script tokens — to predict the next tactic, which is from a different vocabulary. An RNN allows encoding a sequence of n real-valued vectors as one representative vector h through the following steps.

```
$h_1 = [0, \ldots, 0]$
$h_{i+1} = RNN(r_i, h_i) = r_i.W_r + h_i.W_h + b_r$
$h = RNN(r_n, h_n)$
```

LSTMs and GRUs can hence be used to effectively model general sequences of data when there are only thousands of training sequences, and so we use them to model the sequence of proof script tokens.

3.2 TacTok Language Modeling

TacTok first trains a model on existing proof scripts, and then applies the model to synthesize new proof scripts. Figure 4 details how TacTok trains a model on a set of proof scripts. Given a set of proof scripts, each proof script is broken down into training instances. A training instance consists of the input to the model, which is the proof state after a tactic in the proof script is executed

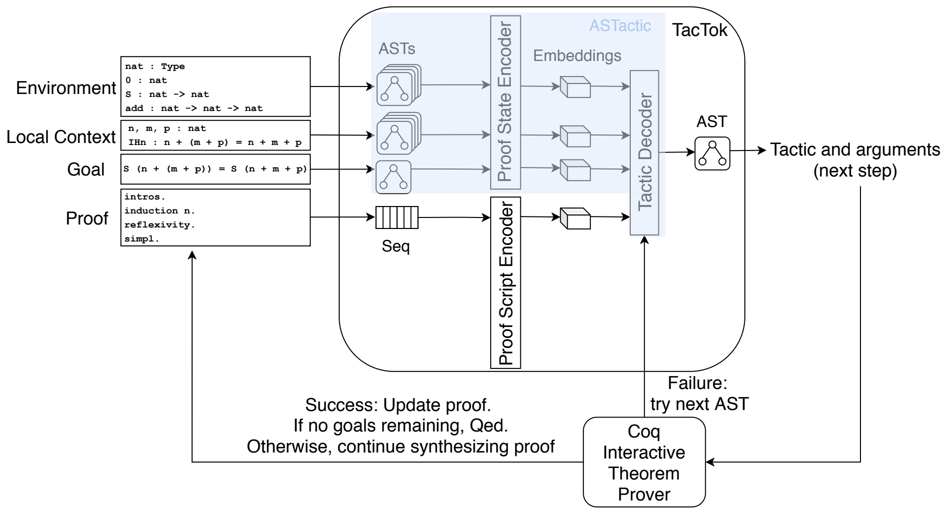
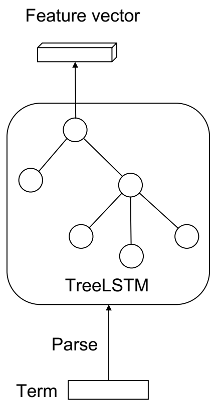


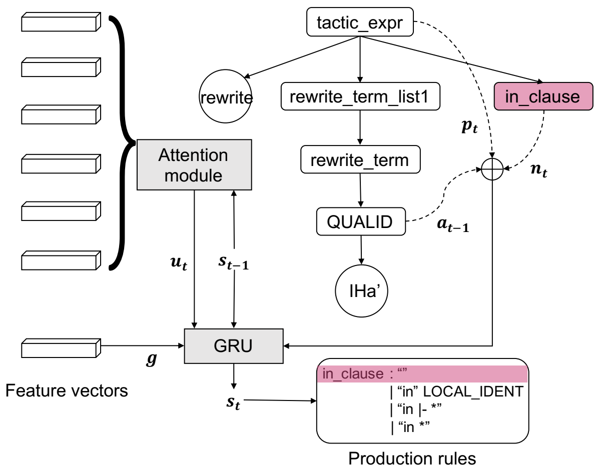
Fig. 5. TacTok architecture. TacTok, in the process of completing the proof script of associativity for the add function (recall Section 2), after the execution of **simpl**. TacTok's input is the proof state (goal, local context, and environment) and the partial proof script synthesized so far. Each term in the proof state has an underlying AST, and the proof state encoder generates embeddings for each of these ASTs, as done in ASTactic. The proof script is represented as a sequence, and the proof script encoder generates embeddings from this sequence. The tactic decoder, modified from ASTactic, uses these embeddings to generate the next predicted tactic and its arguments in the form of an AST. TacTok interfaces with the Coq Interactive Theorem Prover to execute the higher level expression associated with this predicted tactic. Upon failure, TacTok resamples the tactic and its arguments and tries again. Upon success, TacTok updates the proof script and proof state to reflect the execution of the tactic. This process continues until the proof script is complete or times out.

and the proof script up to the point of the executed tactic, and the next step of the proof script. The proof state is made up of the current goal, local context, and environment. Each term in the proof state has an underlying AST. The proof script is represented as a sequence of tokens. The TacTok model jointly learns embeddings for these ASTs and sequences. The TacTok model uses these embeddings to output a predicted next proof script step, in the form of an AST, and sends that along with the AST form of the ground-truth next tactic to the trainer. The trainer then compares these tactics and back-propagates the loss.

Figure 5 details how TacTok synthesizes proof scripts, using the proof script for the proof of associativity of the add function (recall Section 2.2) as an example. The figure shows TacTok at the point where it has already generated [**intros**, **induction** n, **reflexivity**, **simpl**] of the proof script. TacTok takes as input the proof state and the partial proof script so far. The proof state encoder (Section 3.2.1) takes the AST form of the proof state inputs and outputs embeddings for each. The proof script encoder (Section 3.2.2) takes the sequence of tokens in the proof script and outputs an embedding for this sequence. The tactic decoder (Section 3.2.3) uses these embeddings to predict an AST, which represents a tactic and its arguments. TacTok then interfaces with the Coq ITP to execute this tactic and its arguments and interprets the Coq ITP output as either a *failure* or a *success*. A failure is when the Coq ITP produces an error or the execution of the tactic produces a duplicate proof state. In this case, TacTok resamples the tactic and its arguments and interfaces with the Coq ITP again. Otherwise, it is a success and the proof script and proof state



(a) Proof State Encoder.



(b) Tactic Decoder.

Fig. 6. ASTactic architecture from [Yang and Deng 2019]. The proof state encoder (a), takes as input the goal, local context, and environment terms in AST form and generates embeddings (feature vectors) for each term. The tactic decoder (b) concatenates the input embeddings and generates a tactic in the form of an AST, conditioned on these inputs.

update to include the tactic and its arguments and resulting proof state from the tactic's execution. This process continues until the proof script is complete or times out.

We now describe each of the TacTok components.

3.2.1 *Proof State Encoder.*

Figure 6a details the encoder in ASTactic [Yang and Deng 2019], which is the proof state encoder in TacTok. The inputs are the goal, local context, and environment in AST form. It uses a TreeLSTM network [Tai et al. 2015], which allows for encoding a tree, to generate embeddings for each proof state term.

3.2.2 *Proof Script Encoder.*

The proof script consists of tokens that are either tactics, arguments to tactics, or other symbols. The proof script encoder parses the sequence of these tokens in two different modes. One mode of parsing the sequence is to include the most common Coq tactic tokens appearing in the training proof scripts, excluding custom tactics, and obscure arguments so that their names are not learned. When the proof script encoder parses in this way, the model it trains is the *Tac* model. Another mode of parsing the sequence is to include the entire token sequence, only excluding punctuation. When the proof script encoder parses in this way, the model it trains is the *Tok* model. TacTok is comprised of both the Tac and Tok models, trained separately, and uses either one when attempting to synthesize a proof script.

We encode the parsed sequence of previous tokens using a Bidirectional LSTM, as described in Section 3.1.2, which generates an embedding for the sequence. Figure 7 shows the proof script encoder in TacTok.

3.2.3 *Tactic Decoder.*

Figure 6b shows the tactic decoder in ASTactic [Yang and Deng 2019], which is the tactic decoder in TacTok. It is conditioned on the sequence of input embeddings. In TacTok, the embeddings are a concatenation of the embeddings generated from both the proof state and proof script encoders. The decoder generates a tactic in the form of a program by sequentially growing an AST [Yin and Neubig 2017]. At a non-terminal node in the AST, it chooses a production rule

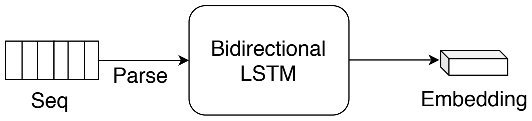


Fig. 7. Proof Script Encoder. The proof script encoder's input is a sequence of proof script tokens, which it parses and then uses to generate an embedding using a Bidirectional LSTM.

from the specified context free grammar (CFG) of the tactic space. At a terminal node, it synthesizes an argument based on semantic constraints. This process of growing the tree is controlled by a GRU [Cho et al. 2014; Chung et al. 2014], which uses the input embeddings of the partially generated AST to update its hidden state.

The tactic decoder has learnable embeddings for all production rules and symbols in the tactic CFG. For example, at time t, n_t is the symbol of the current node, a_{t-1} is the production rule for expanding the prior node, p_t is the production rule for expanding the parent node concatenated with the parent's state, g is the concatenation of the input embeddings, and u_t is the weighted sum of the premises in the environment and local context. The decoder concatenates a_{t-1}, p_t, n_t, g, and u_t, and uses a GRU controller to combine them with the information from the partial tree s_{t-1} to update the decoder state s_t. Then, the decoder uses s_t to predict which production rule to apply.

3.2.4 Training and Search. TacTok is trained on a set of proof scripts. Each proof script in the set is broken down into training instances. A training instance details the proof state after a tactic is executed and the proof script up to the point of the executed tactic. TacTok is trained in same way as the ASTactic model, except for that it also jointly trains a language model over the previous tokens in a proof script.

For automated theorem proving, TacTok uses beam search, just like ASTactic. TacTok samples a fixed number (beam width) of the most likely tactics across all search tree nodes at the same level, and then uses these tactics to search for a complete proof script. TacTok backtracks when it detects a duplicate proof state, or when the Coq compiler fails to check the new attempted proof script step.

3.3 Design Space of Proof Script Synthesis

This section explores the design space of proof script synthesis tools that model a set of completed proof scripts to then generate new ones. We limit our discussion to tools whose models predict a next possible tactic and arguments, and use some form of search to synthesize proof scripts using these predictions. The variability within this design space comes from how a model is built, and how that model is then used to synthesize proof scripts. For example, Figure 8 shows how beam search can use a model to bias a metaheuristic search for a proof script. Here, the model predicts 3 likely next proof script steps (the beam width is 3). The search then checks if adding the step allows the proof script to compile and if the resulting proof state has not been previously observed within this proof script. If these criteria are satisfied, the search keeps the candidate proof script and iterates to continue the search further. Once the proof state contains no subgoals, the proof script can be completed with the application of **Qed**. The search fails if it reaches a timeout.

Some tools, such as CoqHammer, use Automated Theorem Provers (ATP) rather than metaheuristic search. These tools are outside the scope of this design space because they do not model a set of existing proof scripts.

3.3.1 Modeling Proof Scripts. There are different ways of modeling proof scripts. For example, a model could compute the frequency distribution of tactics as they appear in a set of existing,

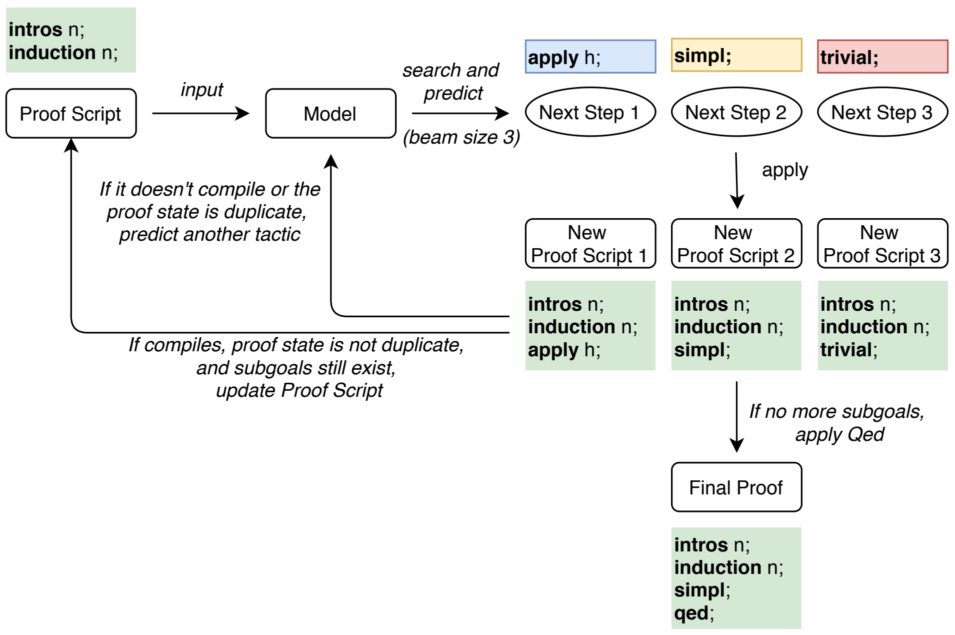

intros n;
induction n;
Proof Script
input
Model
search and predict
(beam size 3)
apply h;
simpl;
trivial;
Next Step 1
Next Step 2
Next Step 3
apply
If it doesn't compile or the proof state is duplicate, predict another tactic
New Proof Script 1
New Proof Script 2
New Proof Script 3
intros n; induction n; apply h;
intros n; induction n; simpl;
intros n; induction n; trivial;
If compiles, proof state is not duplicate, and subgoals still exist, update Proof Script
If no more subgoals, apply Qed
Final Proof
intros n; induction n; simpl; qed;

Fig. 8. The process of synthesizing a proof script using metaheuristic search, biased by a predictive model. Given an incomplete proof script, a model can predict the next proof step. Here, to use a beam search of width 3, the model predicts 3 likely steps. If adding the step satisfies certain criteria, here, the proof script must compile and the resulting proof state must not have been previously observed within this proof script, the process iterates until, either, the proof state contains no subgoals and the proof script can be completed with the application of `Qed`, or the search reaches a timeout.

successful proof scripts. Or, a model could involve encoding aspects of the proof script as features that are then used to inform a model. These proof script aspects include the goals, the proof context, and proof script written thus far, and perhaps others. For our model, we focus on encoding the proof script and the proof state, which, in turn, includes the goals, context, and environment.

Encoding proof script features. The proof script is a sequence of tokens in the Coq tactic language. Traditionally, this problem calls for a language model, which can predict the next token given sequence of token (recall Section 3.1). The two overarching classes of language models are n-gram models (Section 3.1.1) and neural language models (Section 3.1.2). Prior work [Hellendoorn et al. 2018] has evaluated the use of n-grams and RNNs for predicting the next token in a sequence of tokens in Coq. N-gram models have no trainable parameters; instead they keep counts of previously seen common sequences during training and perform a table lookup during inference to predict the next token given a sequence of tokens. In our target tools, the sequence of tokens is used as input along with the proof state. So we want to encode the token sequence using a trainable model with parameters, which requires that we use some neural model to perform this encoding. Within neural language models, transformers [Devlin et al. 2018] and RNNs are widely used [Peters et al. 2018]. Vanilla RNNs are known to have the problem of vanishing gradients because the token sequences are processed one-by-one in a sequential fashion. To mitigate this issue, researchers have proposed different variants of RNNs, such as LSTMs and GRUs. LSTMs and GRUs have similar performance on many language modeling tasks, so we chose LSTMs for our experiments. More recently, transformers have become very popular for language modeling tasks. These are massive

parameter	value
TacTok model	
proof script terms embedding size	256
proof script encoder LSTM hidden size	256
number of hidden layers in LSTM	1
Tac sequence length (including start token)	3
Tok sequence length (including start token)	30
learning rate	3×10^{-5}
batch-size	128
training epochs	4
Search and Synthesis	
beam width	20
search depth limit	5
timeout	10 minutes

Fig. 9. Experimental hyper-parameter values.

attention-based neural models that can process the entire token sequence in one shot during training, making them faster to train. However, they are known to be tricky to train and require large sets of data, typically in the order of millions of lines of text that one typically sees for natural language corpora, such as Wikipedia, to outperform LSTM-based models. We use LSTMs for our task due to the constraint of the benchmarking dataset size and ease of modeling, leaving the exploration of transformers to future work. TacTok encodes the proof script (see Section 3.2.2) using a Bidirectional LSTM to capture more contextual information by processing the sequence forward and backward. We configure the proof script encoder's term embedding size, LSTM hidden size, number of hidden layers in the LSTM, batch-size, sequence length (including start token), and learning rate by performing a random grid search over these hyperparameters. Figure 9 lists the hyperparameters we made for TacTok.

It is also possible to preprocess the raw sequence of tokens in the proof script prior to learning a model. The tokens in the raw sequence can be categorized into tactics, their arguments, and punctuation. This preprocessing can include every token in the proof script, or can exclude some, e.g., the punctuation or arguments. Preprocessing can also obscure or rename arguments. Section 3.2.2 describes our preprocessing, parsing choices.

Encoding the proof state. The proof state is a sequence of proof state terms. The proof state terms can be expressed in a specific format, which is displayed to the programmer during interactive proving (see Section 2), and is essentially a sequence of tokens. The proof state also has an underlying tree-structured representation consisting of Coq terms. A neural architecture can encode the terms in both formats. The use of the underlying tree-structure representation is preferable since some information is lost in the conversion to the human-readable representation. ASTactic [Yang and Deng 2019] accesses the underlying Coq terms of the proof state and uses a TreeLSTM to encode these Coq terms to represent the proof state. TacTok inherits ASTactic's proof state encoding design choices.

3.3.2 Search-Based Proof Script Synthesis. To synthesize a complete proof script, the tools in our design space employ a search-based strategy. The space of all possible proof scripts is, of course, infinite, but a precise model from Section 3.3.1 can help navigate this space towards the proof script that makes proving the theorem possible.

Search is a step-by-step process. The space of proof scripts can be represented as a tree, where the nodes are (partial) proof scripts, and a path from the root to a leaf is a potential complete proof script. Standard search algorithms, such as depth-first search and breadth-first search, are applicable to this space. Though breadth-first search allows for all paths to be explored in parallel, this is a vast search space to cover, and so it is more efficient to prune the search space. Beam search limits the width of a search, considering a fixed number of tactics across all search tree nodes at the same level. This number of tactics being considered is called the beam width.

A proof script that does not compile or is incomplete, can be mutated it in different ways, including insertion, deletion, and modification of tactics, to produce candidate proof scripts. These candidates can be executed, pruned, and further modified as the algorithm searches for a working proof script. For example, genetic algorithms can potentially be used to search for a proof script given a set of existing candidate scripts, e.g., generated using the step-by-step search process. This combination may be a fruitful direction for future work in search-based proof script synthesis strategies.

ASTactic [Yang and Deng 2019] is an example of a tool that uses beam search to sample tactics, and use these tactics to attempt to construct a complete proof script. The authors conducted experiments on different beam widths and found 20 to be most effective. TacTok inherits this design choice.

4 PROOF AND PROOF SCRIPT SYNTHESIS TOOLS

Our evaluation will compare TacTok to five tools, CoqHammer, ASTactic, SeqOnly, WeightedRandom, and WeightedGreedy. This section describes these tools.

The most powerful, general-purpose techniques for automating verification in an ITPs, such as the Coq ITP, are called *hammers* [Blanchette et al. 2016]. A hammer performs efficient automated reasoning using facts from a preexisting library. It uses machine-learning techniques to select the facts that are likely to be needed to prove a theorem. Then, it translates these selected facts and the theorem from the ITP logic to a form accepted by ATPs, which are theorems provers for first-order logic. The ATPs take the resulting translation and try to find a proof. Lastly, the hammer processes the proof found by an ATP, and tries to reconstruct the proof of the theorem in the ITP logic. CoqHammer [Czajka and Kaliszyk 2018] is one such hammer for Coq. As in prior evaluations of automated verification tools that compare to CoqHammer [Yang and Deng 2019] our experiments configure CoqHammer to use four ATPs: Z3 [de Moura and Bjørner 2008], Vampire [Kovács and Voronkov 2013], CVC4 [Barrett et al. 2011], and E Prover [Schulz 2013].

ASTactic is a search-based automated verification tool that uses deep learning to learn to predict the next step of a proof script solely from the current proof state [Yang and Deng 2019]. To make a prediction for a given, incomplete proof script, ASTactic uses the current goal, local context, and environment, and generates tactics in the form of ASTs (just like TacTok).

Our central goal in developing TacTok is to demonstrate the effect of modeling both the proof state and the partial proof script on proof script synthesis. Since ASTactic [Yang and Deng 2019] models proof state alone to synthesize proof scripts, a direct comparison between TacTok and ASTactic demonstrates the effect of adding the partial proof script to the model. A natural complement is then a tool that models only the partial proof script, to measure the effect of adding the proof state to the model. However, such a tool, without knowing any of the goals, including the theorem being proven, would follow the same pattern for every theorem and would prove very few, if any, of them. (Anecdotally, we implemented such a tool and confirmed that it proves a small number of theorems.) Instead, we developed SeqOnly, a tool that follows the same basic structure as TacTok and models the partial proof script, but instead of modeling the entire proof state, it only models the theorem being proven. SeqOnly uses the TacTok proof script encoder (recall Section 3.2.2) to generate embeddings for the proof script sequence, but its tactic decoder (recall Section 3.2.3) is conditioned only on the proof script sequence embedding and the initial theorem. SeqOnly has the potential to be effective because language models (n-grams and neural networks) have been shown to be able to predict next tokens in Coq proof scripts [Hellendoorn et al. 2018]. For example, using an RNN to represent the sequence of tokens in a proof script allows for 56.6% accuracy in predicting the next token.

Additionally, we build two more proof script synthesis tools, WeightedRandom and WeightedGreedy, that use the frequencies of tactic ASTs occurring in the training proofs to predict the next likely proof step. WeightedRandom computes a probability distribution over the entire set of tactic ASTs by measuring the frequency of each tactic AST in the training set of proof scripts. It then performs a search, just as TacTok, but using only that probability distribution to make predictions over the next tactic AST. To make the top-k predictions at each step of the search algorithm, WeighedRandom samples k tactic ASTs from this distribution without replacement.

WeightedGreedy uses the same probability distribution as WeightedRandom, but always selects the top-k most frequent tactic ASTs from the training set of proof scripts, as opposed to sampling the distribution. Otherwise, it uses search to construct a proof script in the same way as WeightedRandom.

5 EVALUATION

We evaluate TacTok by comparing its ability to fully automatically synthesize proof scripts to that of two state-of-the-art synthesis tools, CoqHammer [Czajka and Kaliszyk 2018] and ASTactic [Yang and Deng 2019], and our own three techniques, SeqOnly, WeightedRandom, and WeightedGreedy. CoqHammer attempts to automatically prove theorems using external Automatic Theorem Prover (ATP) systems (recall Section 4). ASTactic, like TacTok, learns from existing proof scripts but uses only the proof state to predict the next step of the proof script. Finally, SeqOnly is a tool we created ourselves that learns from existing proof scripts but uses only the partially written proof script and the initial theorem to predict the next step of the proof script. WeightedRandom is a model that constructs a probability distribution over the set of tactics in the training dataset, while WeightedGreedy is a model of the maximum likelihood prediction based on the distribution of tactics in the training dataset.

Our implementations, experimental scripts, and data are publicly available in our replication package [First et al. 2020].

TacTok uses search in the proof-script space, whereas CoqHammer produces proofs in Coq's logic (Gallina) using a fundamentally different approach. When the Coq compiler executes a proof script, it generates a proof. Proof scripts can be wrong — e.g., it is possible to write a proof script that concludes with *Proof completed*, but that is not a valid proof when it is checked by the Coq compiler — but proofs cannot. As such, it is reasonable to compare TacTok (and other proof script synthesis tools) to CoqHammer in terms of the theorems they are able to prove, but as their approaches are so fundamentally different, it should be no surprise that the tools are likely to excel for different theorems, and be complementary. For some simpler classes of theorems, CoqHammer and TacTok are likely to perform similarly well, whereas for others, (e.g., proofs that require induction) CoqHammer will be at a fundamental disadvantage. CoqHammer will be perhaps more predictable in the types of theorems it is able to prove, whereas proof script synthesis tools may provide more surprises. Section 5.5 discusses some theorems TacTok was able to prove but other prior tools, including CoqHammer, could not.

Our evaluation uses the state-of-the-art CoqGym benchmark [Yang and Deng 2019] and answers three research questions:

RQ1: Does modeling proof state and the partially written proof script, together, improve automatic verification efficacy, as compared to modeling only the proof state or only the partial proof script?
RQ2: Does modeling the proof state and partial proof script, together, improve automatic verification efficacy over state-of-the-art ATP-using CoqHammer?
RQ3: Does modeling of other (non-tactic) tokens together with the proof state improve automatic verification efficacy as compared to modeling the tactic sequences together with proof state?

We next describe our evaluation methodology (Section 5.1), and then answer each of the research questions (Sections 5.2–5.4). We end with a discussion of proof scripts of complex theorems that TacTok generates (Section 5.5).

5.1 Evaluation Methodology

This section describes the dataset and metrics our experiments use, and the tools to which we compare TacTok's performance.

5.1.1 Dataset. In our evaluation, we use the CoqGym benchmark [Yang and Deng 2019]. In total, we use 68,501 theorems from 122 open-source software projects in Coq. CoqGym is the state-of-the-art benchmark used in prior evaluations of formal verification tools [Yang and Deng 2019].

The CoqGym benchmark comes with a preselected training set of 96 projects with 57,719 human-written proof scripts. The preselected testing set in CoqGym is comprised of 27 projects. We were unable to reproduce prior results for ASTactic's performance [Yang and Deng 2019] for one project, coq-library-undecidability, due to internal Coq errors when processing the proof scripts within that project. Accordingly, we exclude this project from our evaluation. We were able to reproduce the results for the remaining 26 projects of 10,782 theorems, and this is the testing set our evaluation uses. Each project in the testing set has between 2 and 2.1K theorems, summarized in Figure 13. Following the methodology in prior evaluations [Yang and Deng 2019], our experiments use the training set to train our models and the testing set to measure efficacy of automated verification. When the training instances are extracted from the proof scripts, there are 189,824 training instances.

5.1.2 Metrics. Our experiments measure two key metrics, *success rate* and *added value*. The success rate of a tool is the fraction of all theorems that tool fully automatically generate a proof script for. Success rate is widely used in prior evaluations [Huang et al. 2018; Yang and Deng 2019]. The added value of tool A as compared to tool B is the relative increase in the fraction of *new* theorems tool A proves that tool B fails to prove, as compared to the total number of theorems tool B proves. For example, if Tool B proves 10 theorems, and tool A only proves 5, but tool B cannot prove any one of these 5, then tool A's added value is 50% (5 new theorems out of 10 previously provable theorems). In other words, together, tools A and B can prove 15 theorems, a 50% improvement over tool B's 10.

Because we view our approach as complementary to the prior work, our hope is that TacTok will prove some theorems the prior approaches fail to prove (as opposed to strictly improving the absolute success rate). The added value metric captures this measure.

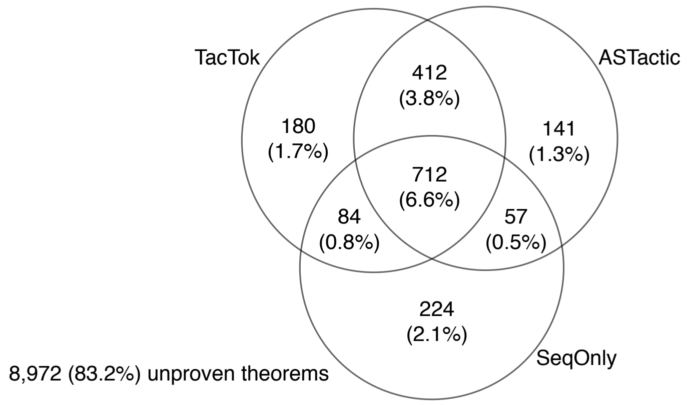


Fig. 10. Theorems proven by TacTok, ASTactic, and SeqOnly out of the 10,782 theorems in CoqGym's test dataset. Figure 13 shows the by-project breakdown of the data. TacTok has the highest success rate of the three tools, proving 1,388 (12.9%) theorems, proving 264 theorems that ASTactic cannot (an added value of 20.0%). TacTok and ASTactic outperform SeqOnly, each with an added value of about 50%. This suggests that proof state together with the partial proof script can achieve better automated verification than either alone.

5.2 RQ1: Does Modeling Proof State and Partial Proof Script Improve Verification Efficacy?

TacTok uses both the partial proof script and the proof state to predict the next step in a proof script. To understand if this combination of information improves the efficacy of automated proof script verification, we compare TacTok to ASTactic, which uses only proof state, and SeqOnly, which uses only the partial proof script and the initial theorem, as well as to our WeightedRandom and WeightedGreedy approaches.

Figure 10 compares the number of theorems proven by TacTok, ASTactic, and SeqOnly. TacTok proves 1,388 theorems, for a success rate of $\frac{1{,}388}{10{,}782} = 12.9\%$. ASTactic, trained only on the proof state, has a success rate of $\frac{1{,}322}{10{,}782} = 12.3\%$, and SeqOnly has a success rate of $\frac{1{,}077}{10{,}782} = 10.0\%$.

TacTok proves 264 theorems that ASTactic fails to prove, so its value added compared to ASTactic is $\frac{264}{1{,}322} = 20.0\%$.

TacTok proves 592 theorems that SeqOnly fails to prove, so its value added compared to SeqOnly is $\frac{592}{1{,}077} = 55.0\%$. ASTactic proves 553 theorems that SeqOnly fails to prove, so its value added is $\frac{553}{1{,}077} = 51.3\%$.

SeqOnly is able to prove theorems that ASTactic and TacTok fail to prove, for an added value of $\frac{308}{1{,}322} = 23.3\%$ and $\frac{281}{1{,}388} = 20.2\%$, respectively. This suggests that proof state and the partial proof script are helpful for automated verification when used together and separately.

tool	theorems proven	TacTok value added over tool
WeightedRandom	671 (6.2%)	799 (120%)
WeightedGreedy	1,049 (9.7%)	504 (48%)

Fig. 11. Theorems proven by the WeightedRandom and WeightedGreedy, out of the 10,782 theorems in CoqGym's test dataset, and TacTok's added value over these baselines. Both WeightedRandom and WeightedGreedy underperform TacTok, which proves 1,388 theorems (12.9%).

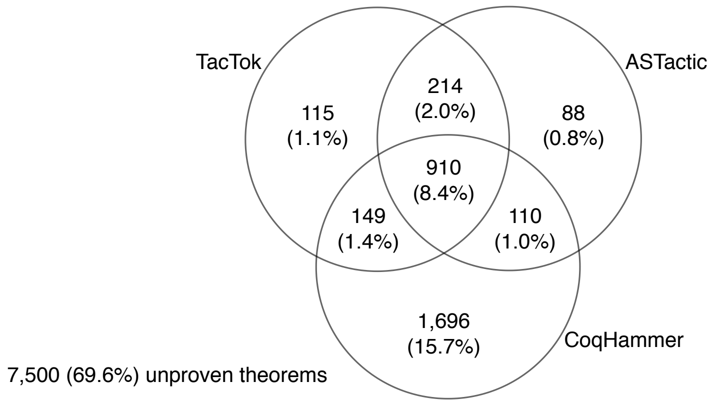


Fig. 12. Theorems proven by TacTok, CoqHammer, and ASTactic out of the 10,782 theorems in CoqGym's test dataset. Figure 13 shows the by-project breakdown of the data. CoqHammer proves more theorems than TacTok and ASTactic, but they each prove different theorems. Together, they can prove more theorems overall.

Figure 11 shows the success rates of WeightedRandom and WeightedGreedy. WeightedRandom proves 671 theorems, for a success rate of $\frac{671}{10{,}782} = 6.2\%$. WeightedGreedy proves 1,049 theorems, for a success rate of $\frac{1{,}049}{10{,}782} = 9.7\%$. Both WeightedRandom and WeightedGreedy underperform TacTok, which proves 1,388 theorems (12.9%).

TacTok proves 799 theorems that WeightedRandom fails to prove, so its added value compared to WeightedRandom is $\frac{799}{671} = 120\%$. WeightedRandom adds little beyond TacTok, proving 82 theorems that TacTok fails to prove, for an added value of $\frac{82}{1{,}388} = 5.9\%$.

TacTok proves 504 theorems that Greedy fails to prove, so its added value compared to Greedy is $\frac{504}{1{,}049} = 48\%$. Greedy proves 165 theorems that TacTok fails to prove, for an added value of $\frac{165}{1{,}388} = 11.9\%$.

RA1: The data suggest that using proof state together with the partial proof script creates significant added value and improves verification efficacy beyond using either kind of information alone. TacTok proves more theorems on its own than ASTactic. Together, they prove 20.0% more than ASTactic alone, and are, therefore, complementary. Meanwhile TacTok and ASTactic outperform SeqOnly, each with an added value of about 50%. WeightedRandom and WeightedGreedy also underperform TacTok.

5.3 RQ2: Does TacTok Improve CoqHammer's Verification Efficacy?

To evaluate whether TacTok improves automatic verification efficacy over other state-of-the-art tools, we compare TacTok to CoqHammer. We also compare to the combination of CoqHammer and ASTactic to understand if TacTok provides value beyond the combination of prior tools.

Figure 12 compares the number of theorems proven by TacTok, CoqHammer, and ASTactic. CoqHammer has a success rate of $\frac{2{,}865}{10{,}782} = 26.6\%$, which dominates the success rates of TacTok and ASTactic, 12.9% and 12.3%, respectively. The success rate of CoqHammer and ASTactic together is $\frac{3{,}167}{10{,}782} = 29.4\%$.

However, TacTok proves 115 theorems that CoqHammer and ASTactic fail to prove, so its added value compared to the combination of CoqHammer and ASTactic is $\frac{115}{3{,}167} = 3.6\%$.

success rate: project	TacTok	ASTactic & CoqHammer	ASTactic	CoqHammer	Tac	Tok	total theorems
weak-up-to	18 (12.9%)	36 (25.9%)	23 (16.5%)	30 (21.6%)	12 (8.6%)	12 (8.6%)	139
buchberger	76 (10.5%)	192 (26.5%)	70 (9.7%)	166 (22.9%)	70 (9.7%)	65 (9.0%)	725
jordan-curve-theorem	22 (3.5%)	168 (26.8%)	19 (3.0%)	165 (26.3%)	16 (2.5%)	22 (3.5%)	628
dblib	45 (25.0%)	67 (37.2%)	41 (22.8%)	55 (30.6%)	41 (22.8%)	35 (19.4%)	180
disel	89 (14.0%)	194 (30.6%)	83 (13.1%)	185 (29.2%)	63 (9.9%)	72 (11.4%)	634
zchinese	5 (11.6%)	13 (30.2%)	5 (11.6%)	12 (27.9%)	4 (9.3%)	4 (9.3%)	43
zfc	33 (13.9%)	70 (29.5%)	33 (13.9%)	64 (27.0%)	28 (11.8%)	28 (11.8%)	237
dep-map	11 (25.9%)	16 (37.2%)	9 (20.9%)	14 (32.6%)	7 (16.3%)	8 (18.6%)	43
chinese	35 (26.7%)	58 (44.3%)	31 (23.7%)	56 (42.7%)	27 (20.6%)	30 (22.9%)	131
UnifySL	180 (18.6%)	367 (37.9%)	189 (19.5%)	303 (31.3%)	126 (13.0%)	153 (15.8%)	968
hoare-tut	5 (27.8%)	6 (33.3%)	1 (5.5%)	6 (33.3%)	3 (16.7%)	3 (16.7%)	18
huffman	28 (8.9%)	81 (25.8%)	25 (7.9%)	74 (23.6%)	22 (7.0%)	21 (6.7%)	314
PolTac	112 (30.9%)	308 (84.9%)	118 (32.5%)	289 (79.6%)	87 (24.0%)	110 (30.3%)	363
angles	4 (6.5%)	15 (24.2%)	4 (6.5%)	15 (24.2%)	3 (4.8%)	4 (6.5%)	62
coq-procrastination	6 (75.0%)	5 (62.5%)	5 (62.5%)	3 (37.5%)	5 (62.5%)	6 (75.0%)	8
tree-automata	111 (13.4%)	311 (37.6%)	96 (11.6%)	292 (35.3%)	83 (10.0%)	96 (11.6%)	828
coquelicot	100 (6.8%)	299 (20.4%)	95 (6.5%)	273 (18.6%)	77 (5.2%)	78 (5.3%)	1,467
fermat4	10 (7.7%)	47 (36.2%)	13 (10.0%)	47 (36.2%)	9 (6.9%)	8 (6.2%)	130
demos	53 (77.9%)	55 (80.9%)	50 (73.5%)	54 (79.4%)	49 (72.1%)	52 (76.5%)	68
coqoban	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	2
goedel	67 (11.1%)	128 (21.1%)	53 (8.7%)	120 (19.8%)	57 (9.4%)	58 (9.6%)	606
verdi-raft	121 (5.7%)	351 (16.5%)	117 (5.5%)	337 (15.8%)	99 (4.7%)	97 (4.5%)	2,127
verdi	47 (8.2%)	127 (24.7%)	37 (7.2%)	122 (23.7%)	37 (7.2%)	42 (8.2%)	514
zorns-lemma	12 (8.1%)	21 (14.1%)	10 (6.7%)	18 (12.1%)	10 (6.7%)	7 (4.7%)	149
coqrel	183 (71.5%)	191 (74.6%)	184 (71.9%)	128 (50.0%)	163 (63.7%)	146 (57.0%)	256
fundamental-arithmetic	15 (10.6%)	41 (28.9%)	11 (7.8%)	37 (26.1%)	10 (7.0%)	9 (6.3%)	142
total	1,388 (12.9%)	3,167 (29.4%)	1,322 (12.3%)	2,865 (26.6%)	1,108 (10.3%)	1,166 (10.8%)	10,782

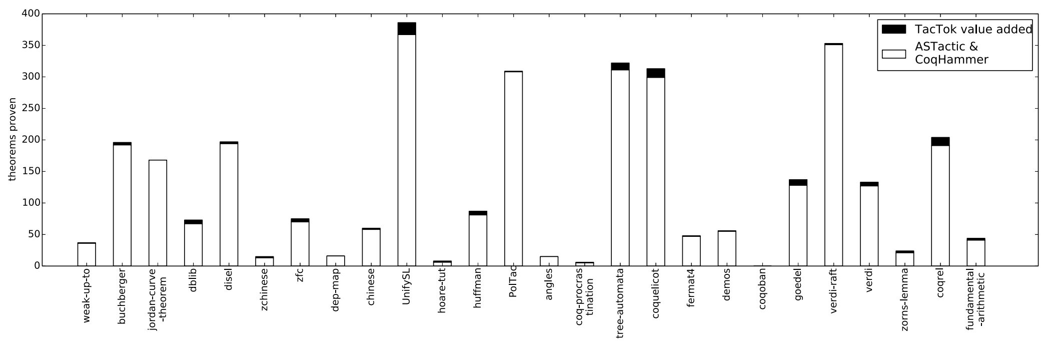


Fig. 13. Success rates, as evaluated on the CoqGym benchmark [Yang and Deng 2019], the three tools, TacTok, ASTactic [Yang and Deng 2019], and CoqHammer [Czajka and Kaliszyk 2018], prove more theorems together than each does on its own. For 22 of the 26 subjects, TacTok proves theorems the other two tools cannot. Overall, together, TacTok proves 115 more theorems than the prior tools combined, a 3.6% relative improvement. Figure 14 demonstrates this improvement for each project.

Figure 13 shows the by-project breakdown of the success rates for each tool, while Figure 14 details the by-project value added of TacTok compared to CoqHammer and ASTactic.

For 22 of the 26 projects, TacTok has added value over the combination of CoqHammer and ASTactic. The greatest increase in the number of theorems proven in a project is 19, for UnifySL; the average increase is 4.4.

added value: project	TacTok over ASTactic & CoqHammer		TacTok over ASTactic		TacTok over CoqHammer		total theorems
	theorems proven	added value	theorems proven	added value	theorems proven	added value	
weak-up-to	1 (0.7%)	2.7%	2 (1.4%)	8.6%	3 (2.2%)	10.0%	139
buchberger	4 (0.6%)	2.1%	10 (1.4%)	14.3%	27 (3.7%)	16.3%	725
jordan-curve-theorem	0 (0.0%)	0.0%	5 (0.8%)	26.3%	1 (0.2%)	0.6%	628
dblib	6 (3.3%)	9.0%	11 (6.1%)	26.8%	11 (6.1%)	20.0%	180
disel	3 (0.5%)	1.5%	24 (3.9%)	28.9%	9 (1.4%)	4.9%	634
zchinese	2 (4.7%)	15.4%	2 (4.7%)	40.0%	2 (4.7%)	16.7%	43
zfc	5 (2.1%)	7.1%	5 (2.1%)	15.2%	6 (2.5%)	9.4%	237
dep-map	0 (0.0%)	0.0%	2 (4.7%)	22.2%	2 (4.7%)	14.3%	43
chinese	2 (1.5%)	3.4%	9 (6.9%)	29.0%	4 (3.1%)	7.1%	131
UnifySL	19 (2.0%)	5.2%	37 (3.8%)	19.6%	56 (5.8%)	18.5%	968
hoare-tut	2 (11.1%)	33.3%	4 (22.2%)	400.0%	2 (11.1%)	33.3%	18
huffman	6 (1.9%)	7.4%	8 (2.5%)	32.0%	11 (3.5%)	14.9%	314
PolTac	1 (0.3%)	0.3%	4 (1.1%)	3.4%	19 (5.2%)	6.6%	363
angles	0 (0.0%)	0.0%	0 (0.0%)	0.0%	0 (0.0%)	0.0%	62
coq-procrastination	1 (12.5%)	20.0%	1 (12.5%)	20.0%	3 (37.5%)	100.0%	8
tree-automata	11 (1.3%)	3.5%	30 (3.6%)	31.3%	24 (2.9%)	8.2%	828
coquelicot	14 (1.0%)	4.7%	28 (1.9%)	29.5%	33 (2.2%)	12.1%	1,467
fermat4	1 (0.8%)	2.1%	2 (1.5%)	15.4%	1 (0.8%)	2.1%	130
demos	1 (1.5%)	1.8%	4 (5.9%)	8.0%	2 (2.9%)	3.7%	68
coqoban	0 (0.0%)	0.0%	0 (0.0%)	0.0%	0 (0.0%)	0.0%	2
goedel	9 (1.5%)	7.0%	16 (2.6%)	30.2%	17 (2.8%)	14.2%	606
verdi-raft	2 (0.1%)	0.6%	19 (0.9%)	16.2%	11 (0.5%)	3.3%	2,127
verdi	6 (1.2%)	4.7%	16 (3.1%)	43.2%	9 (1.8%)	7.4%	514
zorns-lemma	3 (2.0%)	14.3%	3 (2.0%)	30.0%	6 (4.0%)	33.3%	149
coqrel	13 (5.1%)	6.8%	17 (6.6%)	9.2%	64 (25.0%)	50.0%	256
fundamental-arithmetic	3 (2.1%)	7.3%	5 (3.5%)	45.5%	6 (4.2%)	16.2%	142
total	115 (1.1%)	3.6%	264 (2.4%)	20.0%	329 (3.1%)	11.5%	10,782

Fig. 14. TacTok added value, as evaluated on the CoqGym benchmark [Yang and Deng 2019]. TacTok proves 115 theorems that both ASTactic [Yang and Deng 2019] and CoqHammer [Czajka and Kaliszyk 2018] fail to prove, so its added value over the two tools combined is 3.6%. The added value of TacTok over ASTactic is 20.0%. While CoqHammer proves more theorems than TacTok, the added value of TacTok over CoqHammer is 11.5%.

Meanwhile, for 24 of the 26 projects, TacTok has added value over CoqHammer and ASTactic, individually. TacTok proves 264 theorems that CoqHammer fails to prove, so its added value compared to CoqHammer is $\frac{329}{2,865} = 11.5\%$ (and 20.0% over ASTactic, as discussed in Section 5.2).

RA2: While CoqHammer proves more theorems, TacTok can prove 11.5% additional theorems CoqHammer fails to prove. Of these 329 newly proven theorems, 115 cannot be proven by ASTactic, so TacTok adds 3.6% value beyond the two state-of-the-art tools combined.

5.4 RQ3: Do Non-Tactic Tokens Improve Verification Efficacy?

TacTok is comprised of two underlying models, Tac and Tok (recall Section 3.2.3). The two models both encode proof state the same way, but they differ in how they encode partial proof scripts.

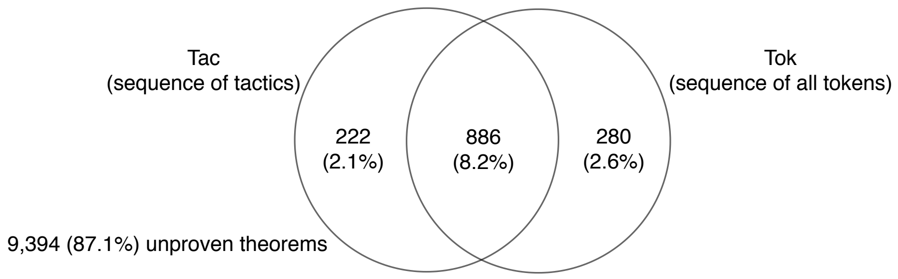


Fig. 15. Theorems proven separately by TacTok's two ways of encoding partial proof scripts. Tac encodes a sequence of tactics, whereas Tok encodes all tokens, in addition to the tactics. The two ways of encoding complement each other in proving theorems.

Tac models the partially written proof scripts by just the sequence of tactics used thus far. Tok, meanwhile, models the partially written proof scripts using all the proof tokens, including the tactics and their arguments. This research question focuses on understanding the contributions Tac and Tok make to TacTok.

Figure 15 compares the number of theorems proven by Tac and Tok. Tac proves 1,108 theorems, for a success rate of $\frac{1,108}{10,782} = 10.3\%$, while Tok has a success rate of $\frac{1,166}{10,782} = 10.8\%$. Figure 13 shows the by-project breakdown of the success rates for Tac and Tok.

Tac and Tok are clearly complementary. Tac outperforms Tok for 8 of the projects, while Tok outperforms Tac for 13 of the projects. Tac is able to prove 222 theorems that Tok fails to prove, so its value added compared to Tok is $\frac{222}{1,166} = 19.0\%$. Tok proves 280 theorems that Tac fails to prove, so its value added compared to Tac is $\frac{280}{1,108} = 25.3\%$. Their complementarity is the prime reason TacTok relies on both, improving its efficacy.

RA3: The data suggest that modeling the sequence of tactics in partial proof scripts is complementary to modeling all tokens. Each approach helps prove around a fifth of the theorems the other cannot. Some proof scripts appear to require the knowledge encoded in the tactic arguments of proof scripts, whereas others are able to be generated accurately (with the arguments) without encoding the arguments as an input to the model. Combining the two produces the highest verification efficacy.

5.5 Does TacTok Prove More Complex Theorems That Other Tools Do Not?

To understand if TacTok is able to prove more complex theorems than prior tools, we manually examined the theorems TacTok was able to prove that prior tools were unable to. We observed several examples of higher-order logic that TacTok proved, but prior tools could not.

One class of examples is theorems that have nested **`forall`** quantifiers. Figure 3 shows an example of such a theorem that TacTok is able to prove, but ASTactic and CoqHammer were not. Higher-order logic is more difficult for a programmer to reason through, and so the use of a proof script or proof synthesis tool is likely to be more helpful.

Another class of examples is theorems that require induction to prove. Figure 16 shows one example such a theorem. The theorem uses the $n^{th}\ l\ n\ a$ relation, which holds when a is the nth element of the list l. If so, the theorem states that it is also the element of a longer list that is prefixed by l. TacTok can generate the inductive proof script of this theorem, but prior tools cannot. CoqHammer does not try to prove theorems that require induction [Czajka and Kaliszyk 2018],

```
1 Lemma Nth_app :
2     forall A (l : list A) l' a n,
3       Nth l n a ->
4       Nth (l ++ l') n a.
5
6 Proof.
7 intro. intro. intro. intro. intro. clear. intro.
8 induction H. constructor. constructor. assumption.
9 Qed.
```

Fig. 16. Lines 1–4 defines the `Nth_app` theorem, and lines 6–9 is the proof script that TacTok generates for this theorem. TacTok correctly applies the **`induction`** tactic.

and so CoqHammer is automatically at a disadvantage for proving this class of theorems. TacTok's modeling of proof state and partial proof script, together, was able to capture sufficient context to properly apply induction.

6 RELATED WORK

We place our research in the context of related work in the areas of interactive theorem provers, automation for proof systems, software engineering for automated proof assistants, language models for code, machine learning in formal verification, and metaheuristic search.

Interactive Theorem Provers (ITPs). ITPs, such as Coq, Isabelle [Nipkow et al. 2002], Mizar [Trybulec and Blair 1985], HOL4 [Slind and Norrish 2008], and HOL Light [Harrison 1996] are semi-automated systems for formally proving theorems. We focus on Coq, but our approach is applicable to other ITPs.

Automation for Proof Systems. Heuristic-based search can partially automate interactive theorem provers [Andrews and Brown 2006; Blanchette et al. 2011; Bundy 1998; Bundy et al. 1990]. *Hammers* are a class of tools that use external ATPs to automatically find proofs for ITPs [Czajka and Kaliszyk 2018]. Classical search algorithms, such as A*, can also search for proofs in HOL4 [Gauthier et al. 2017]. Our models differ from the hammer approach by proving theorems within the ITP framework and using native tactics and by modeling existing proof scripts to attempt to write new ones.

Software Engineering for Interactive Proof Assistants. Pumpkin Patch generates proof patches when proof scripts and specifications need to evolve [Ringer et al. 2018] by searching through a project's history for similar specification changes. Its approach can be expensive due to the size of the search space. A learning-based approach, such as ours, could improve the performance of patch search by prioritizing the search results based on model predictions. Unlike Pumpkin Patch, TacTok does not require a nearly-working proof script, but rather generates proof scripts from scratch.

iCoq [Celik et al. 2017] finds failing proof scripts in evolving projects by only checking proof scripts that are affected by a revision, speeding up the process. iCoq tracks fine-grained dependencies between Coq definitions, propositions, and proof scripts, to narrow down the affected proof scripts. TacTok does not require a failing proof script, but ideas we presented here for modeling existing proof scripts can perhaps be used to repair proof scripts.

QuickChick [Lampropoulos et al. 2017] is a random testing tool for Coq that searches for counterexamples to executable theorems. It helps a programmer gain confidence that a theorem is correct, before they expend effort to formally prove the theorem.

Language Models for Code. Research on language modeling in natural language processing literature has focused on n-gram models [Katz 1987; Kneser and Ney 1995] and neural networks [Bengio et al. 2003; Mikolov et al. 2010; Sundermeyer et al. 2012], (recall Section 3.1). Recent work has applied language modeling to source code for bug detection and test-case generation [Abou-Assaleh et al. 2004; Ernst 2017; Ray et al. 2016]. Language models (n-grams) can be used to model code and perform code completion as the programmer types [Hindle et al. 2016, 2012]. Modified n-grams can be used as a cache to capture local dependencies in code [Tu et al. 2014]. The cache model makes test suggestions through a combination of two n-gram models, built separately on overall code and code that is local to the specific test case. These systems have been applied to mainstream programming languages, such as Java and C. By contrast, our approach focuses on interactive proof assistants and on combining models of proof state and code.

Applying language models to Coq and HOL4 proof scripts showed that n-gram models outperform recurrent neural networks [Hellendoorn et al. 2018], which is rarely true in natural language applications. Unlike TacTok, this approach did not consider the proof state and did not attempt to synthesize whole proof scripts, instead measuring the predictive power of such models on tokens. Further, the approach encoded several kinds of low-signal tokens, including punctuation (e.g., the periods after each tactic) and tokens that lie outside proof scripts. While the application domain differs from TacTok's, the underlying concept of modeling partially written proof scripts to predict the next proof step is similar.

Machine Learning in Formal Verification. Machine learning techniques can make formal verification easier: ML4PG helps Coq users construct proof scripts by showing them similar proof scripts of similar theorems, using a clustering algorithm [Heras and Komendantskaya 2014; Komendantskaya et al. 2012]. Machine learning can similarly help with premise selection, the task of selecting lemmas that are relevant to a given theorem [Alama et al. 2014; Irving et al. 2016; Wang et al. 2017].

GamePad [Huang et al. 2018] models the proof state in Coq using recurrent neural networks. TacTok uses a similar set of components to capture the proof state. However, unlike GamePad, TacTok also models the user-written proof script and combines this information with the proof state to perform proof script generation. GamePad is evaluated for a specific algebraic rewrite problem, which only has two possible tactics with each having two integer argument [Huang et al. 2018]. By contrast, TacTok interfaces with CoqGym's framework to perform general proof script generation.

We evaluate our work on the CoqGym [Yang and Deng 2019] benchmark. In addition to being the first large-scale dataset of Coq projects collected from Github, CoqGym is also a learning environment. ASTactic, a deep-learning model that generates tactics as programs in AST form was the first to evaluate on that dataset [Yang and Deng 2019].

Metaheuristic Search. Metaheuristic-search-based software engineering [Harman 2007] has been used for developing test suites [Michael et al. 2001; Walcott et al. 2006], finding safety violations [Alba and Chicano 2007], refactoring [Seng et al. 2006], project management and effort estimation [Barreto et al. 2008], and automated program repair [Afzal et al. 2020; Ke et al. 2015; Weimer et al. 2009]. Good fitness functions are critical to the success of metaheuristic-search-based software engineering, and low-quality fitness functions, or, at least, a low-quality final validation function, can lead to low-quality results, such as, for example, incorrect bug patches [Motwani et al. 2020; Smith et al. 2015]. With TacTok, the interactive theorem prover provides a strong assurance that the final produced proof script leads to a correct proof, and thus, proof script synthesis may be particularly well suited for metaheuristic-search-based methods.

7 CONTRIBUTIONS

This paper is the first to explore the value of combining modeling of the proof state and of the partially written proof script to synthesize proof scripts, from scratch. We reify this modeling in TacTok, an open-source automated proof script synthesis technique. Evaluating TacTok against CoqHammer [Czajka and Kaliszyk 2018], ASTactic [Yang and Deng 2019], and other metaheuristic search-based approaches we create, we find that TacTok is complementary to other tools: it successfully synthesize proof scripts for theorems prior tools cannot for 24 out of the 26 projects on which we evaluate. With TacTok, 11.5% more theorems can be proven automatically than by CoqHammer alone, and 20.0% than by ASTactic alone. Compared to a combination of CoqHammer and ASTactic, TacTok can prove an additional 3.6% more theorems, proving 115 theorems no tool could previously prove.

Overall, our experiments provide evidence that partial proof script and proof state semantics, together, provide useful information for proof script modeling. We create a concrete approach for modeling the two types of information together, which can serve as a basis for further progress creating automatic verification tools.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grants no. CCF-1453474, no. CCF-1564162, and no. CCF-2018393. 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.

REFERENCES

Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. 2004. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International IEEE Computer Software and Applications Conference*, Vol. 2. 41–42. https://doi.org/10.1109/CMPSAC.2004.1342667

Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2020. SOSRepair: Expressive Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering (TSE)* (2020). https://doi.org/10.1109/TSE.2019.2944914

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

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

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

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

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. https://doi.org/10.1016/j.cor.2007.01.010

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *International Conference on Computer Aided Verification (CAV)*, Vol. 6806. Springer, Snowbird, UT, USA, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

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. https://dl.acm.org/doi/10.5555/944919.944966

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 (2010), 66–75. https://doi.org/10.1145/1646353.1646374

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

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. 2016. Hammering towards QED. *Journal of Formalized Reasoning* 9, 1 (2016), 101–148. https://doi.org/10.6092/issn.1972-5787/4593

Eric Brill and Robert C Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*. Hong Kong, 286–293. https://doi.org/10.3115/1075218.1075255

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

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*. Springer, 647–648. https://doi.org/10.1007/3-540-52885-7_123

Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. ICoq: Regression proof selection for large-scale verification projects. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 171–182. https://doi.org/10.1109/ASE.2017.8115630

Adam Chlipala. 2013. *Certified Programming with Dependent Types*. MIT Press, Boston, MA, USA.

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 *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

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

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963 (April 2008), 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. 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). *arXiv preprint arXiv:1810.04805*, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Michael D. Ernst. 2017. Natural Language is a Programming Language: Applying Natural Language Processing to Software Development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.4

Emily First, Yuriy Brun, and Arjun Guha. 2020. Replication package for "TacTok: Semantics-aware proof synthesis". https://doi.org/10.5281/zenodo.4088897

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*, Vol. 46. 125–143.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. *Neural Computation* 12, 10 (1999), 2451–2471. https://doi.org/10.1162/089976600300015015

Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. 2017. LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems (TNNLS)* 28, 10 (2017), 2222–2232. https://doi.org/10.1109/TNNLS.2016.2582924

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

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

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

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

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. https://doi.org/10.1145/3236024.3264832

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

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (April 2016), 122âĂŞ131. https://doi.org/10.1145/2902362

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

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

Atalay İleri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2018. Proving Confidentiality in a File System Using DISKSEC. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi18/presentation/ileri

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*. Barcelona, Spain, 2235–2243. https://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection

Jonathan Jacky, Stefani Banerian, Michael D. Ernst, Calvin Loncaric, Stuart Pernsteiner, Zachary Tatlock, and Emina Torlak. 2017. Automatic formal verification for EPICS. In *International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS)*. Barcelona, Spain. https://doi.org/10.18429/JACOW-ICALEPCS2017-TUDPL02

Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2012. Establishing Browser Security Guarantees Through Formal Shim Verification. In *Proceedings of the 21st USENIX Conference on Security Symposium*. Bellevue, WA, USA. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang

Slava Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35, 3 (1987), 400–401. https://doi.org/10.1109/TASSP.1987.1165125

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

Reinhard Kneser and Hermann Ney. 1995. Improved backing-off for m-gram language modeling. In *International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1. Detroit, MI, USA, 181–184. https://doi.org/10.1109/ICASSP.1995.479394

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL; Interactive Poster and Demonstration Session*. 177–180. https://www.aclweb.org/anthology/P07-2045

Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. 2012. Machine learning in proof general: Interfacing interfaces. In *Proceedings of 10th International Workshop on User Interfaces for Theorem Provers*, Vol. 118. Bremen, Germany. https://doi.org/10.4204/EPTCS.118.2

Laura Kovács and Andrei Voronkov. 2013. First-Order Theorem Proving and Vampire. In *International Conference on Computer Aided Verification (CAV)*, Vol. 8044. Springer-Verlag, Saint Petersburg, Russia, 1–35. https://doi.org/10.1007/978-3-642-39799-8_1

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 45:1–45:30. https://doi.org/10.1145/3158133

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, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

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

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

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *The 11th Annual Conference of the International Speech Communication Association (INTERSPEECH)*. Makuhari, Chiba, Japan. https://doi.org/10.1109/IALP.2016.7875937

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

Manish Motwani, Mauricio Soto, Yuriy Brun, René Just, and Claire Le Goues. 2020. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering (TSE)* (2020). https://doi.org/10.1109/TSE.2020.2998785 DOI: 10.1109/TSE.2020.2998785.

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

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)*, Vol. 1. Association for Computational Linguistics, New Orleans, LA, USA, 2227–2237. https://doi.org/10.18653/v1/N18-1202

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

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. Los Angeles, CA, USA, 115–129. https://doi.org/10.1145/3167094

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533. https://doi.org/10.1038/323533a0

Stephan Schulz. 2013. System Description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Ken McMillan, Aart Middeldorp, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, 735–743. https://doi.org/10.1007/978-3-642-45221-5_49

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

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

Konrad Slind and Michael Norrish. 2008. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*. 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

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 *Proceedings of the 10th 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

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 *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1631–1642. https://www.aclweb.org/anthology/D13-1170

Andreas Stolcke. 2002. SRILM — An extensible language modeling toolkit. In *Proceedings of the International Conference on Spoken Language Processing*. http://www.speech.sri.com/projects/srilm/papers/icslp2002-srilm.pdf

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In *Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH)*. Portland, OR, USA.

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

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

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

Andrzej Trybulec and Howard A Blair. 1985. Computer Assisted Reasoning with MIZAR. In *Proceedings of the 9th International Joint Conferences on Artificial Intelligence (IJCAI)*, Vol. 85. 26–28. https://www.ijcai.org/Proceedings/85-1/Papers/006.pdf

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. Hong Kong, China, 269–280.

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

Martin Vechev and Eran Yahav. 2016. Programming with “Big Code”. *Foundations and Trends® in Programming Languages* 3, 4 (2016), 231–284. https://doi.org/10.1561/2500000028

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

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

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

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. https://doi.org/10.1145/2737924.2737958

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, 11. http://proceedings.mlr.press/v97/yang19a/yang19a.pdf

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