arXiv:1905.09381v1 [cs.LO] 21 May 2019

Learning to Prove Theorems via Interacting with Proof Assistants

Kaiyu Yang[1] Jia Deng[1]

Abstract

Humans prove theorems by relying on substantial high-level reasoning and problem-specific insights. Proof assistants offer a formalism that resembles human mathematical reasoning, representing theorems in higher-order logic and proofs as high-level tactics. However, human experts have to construct proofs manually by entering tactics into the proof assistant. In this paper, we study the problem of using machine learning to automate the interaction with proof assistants. We construct *CoqGym*, a large-scale dataset and learning environment containing 71K human-written proofs from 123 projects developed with the Coq proof assistant. We develop *ASTactic*, a deep learning-based model that generates tactics as programs in the form of abstract syntax trees (ASTs). Experiments show that ASTactic trained on CoqGym can generate effective tactics and can be used to prove new theorems not previously provable by automated methods. Code is available at `https://github.com/princeton-vl/CoqGym`.

1. Introduction

Given the statement of a theorem, simply push a button, and the proof comes out. If this fantasy of automated theorem proving (ATP) were true, it would impact formal mathematics (McCune, 1997), software verification (Darvas et al., 2005), and hardware design (Kern & Greenstreet, 1999). In reality, however, state-of-the-art theorem provers are still far behind human experts on efficiently constructing proofs in large-scale formal systems.

Consider this theorem: $0+1+2+\cdots+n = \frac{n(n+1)}{2}$. As humans, we decide to prove by induction on n. After solving the trivial case ($n = 0$), we complete the proof by applying the induction hypothesis and simplifying the resulting formula. Simple as it is, the proof requires understanding the concepts (natural numbers, addition), mastering the proof techniques (induction), as well as problem-specific insights to drive the decisions we made.

What a typical theorem prover does, however, is to prove by resolution refutation: It converts the premises and the negation of the theorem into first-order clauses in conjunctive normal form (CNF). It then keeps generating new clauses by applying the resolution rule until an empty clause emerges, yielding a proof consisting of a long sequence of CNFs and resolutions. While this provides a universal procedure, the CNF representation of simple formulas can be long, complicated and inscrutable, making it difficult to benefit from the higher-level abstraction and manipulation that is common to human mathematical reasoning.

To work around the difficulties of ATP in practical applications, interactive theorem proving (ITP) (Harrison et al., 2014) incorporates humans in the loop. In ITP, human users define mathematical objects formally and prove theorems semi-automatically by entering a sequence of commands called tactics. The tactics capture high-level proof techniques such as induction, leaving low-level details to the software referred to as proof assistants. A successful sequence of tactics is essentially a proof written in the language of the proof assistant. It can also be viewed as a program that is executed by the proof assistant.

ITP relies on humans in the loop, which is labor-intensive and hinders its wider adoption. However, at the same time, it is a blessing in disguise, in that it opens up a route to full automation—human experts have written a large amount of ITP code, which provides an opportunity to develop machine learning systems to imitate humans for interacting with the proof assistant. Indeed, some recent efforts have attempted to learn to generate tactics from human-written proofs and have obtained promising results (Gransden et al., 2015; Gauthier et al., 2018; Bansal et al., 2019).

However, existing approaches to the "auto-ITP" problem suffer from two limitations. One is the lack of a large-scale dataset. Prior work was trained and evaluated on no more than a few thousands of theorems (Gransden et al., 2015; Gauthier et al., 2018; Huang et al., 2019), likely insufficient for data-hungry approaches such as deep learning. The other

[1] Department of Computer Science, Princeton University. Correspondence to: Kaiyu Yang <kaiyuy@cs.princeton.edu>, Jia Deng <jiadeng@cs.princeton.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

is the limited flexibility of the learned models in generating tactics. A tactic can be a sophisticated line of code with functions, arguments, and compound expressions, and the space of possible tactics is infinite. Prior work has limited flexibility because they generate tactics by copying from a fixed, predetermined set (Gransden et al., 2015; Gauthier et al., 2018; Huang et al., 2019; Bansal et al., 2019), and are thus unable to generate out-of-vocabulary tactics unseen in the training data.

In this work we address these limitations by making two contributions: a large-scale dataset and a new method for tactic generation that is more flexible and adaptive.

CoqGym: A large-scale ITP dataset and learning environment We construct *CoqGym*, a dataset and learning environment for theorem proving in proof assistants. It includes 71K human-written proofs from 123 open-source software projects in the Coq proof assistant (Barras et al., 1997), covering a broad spectrum of application domains, including mathematics, computer hardware, programming languages, etc. Our dataset is much larger and more diverse than existing datasets, which consist of only a few thousands of theorems and cover only a handful of domains such as Peano arithmetic (Dixon & Fleuriot, 2003) or the Feit–Thompson theorem (Gonthier et al., 2013). The scale and diversity of our dataset facilitate training machine learning models and the evaluating cross-domain generalization.

The learning environment of CoqGym is designed for training and evaluating auto-ITP agents. The agent starts with a set of premises and a goal (theorem) to prove; it interacts with the proof assistant by issuing a sequence of tactics. The proof assistant executes each tactic and reports the results back in terms of a set of new goals. The agent succeeds when no more goals exist.

To make the challenging task more amenable to learning, we augment CoqGym with shorter proofs. They are synthesized from the intermediate steps of the original human-written proofs and may serve as additional training data.

ASTactic: A new method for tactic generation We develop *ASTactic*, a deep learning model for tactic generation that is more flexible and adaptive than prior work. It generates tactics as programs by composing abstract syntax trees (ASTs) in a predefined grammar using the tokens available at runtime. To our knowledge, this is the first time learning-based AST generation is applied in the context of interactive theorem proving.

ASTactic takes in a goal and a set of premises expressed as terms in Coq's language. It outputs a tactic as an AST in a subset of Coq's tactic language.

Experimental results on CoqGym show that ASTactic can generate effective tactics. It can successfully prove 12.2% of the theorems in the test set, significantly outperforming the built-in automated tactics (`auto`, `intuition`, `easy`, etc.) in Coq (4.9%). More importantly, our model can be combined with state-of-art ATP systems (De Moura & Bjørner, 2008; Barrett et al., 2011; Kovács & Voronkov, 2013; Schulz, 2013) and boost the success rate further to 30.0%, which shows that our model has learned effective higher level tactics and has proved new theorems not previously provable by automatic systems.

Contributions In summary, our contributions are two-fold. First, we build CoqGym—a large-scale dataset and learning environment for theorem proving via interacting with a proof assistant. Second, we develop ASTactic, a deep learning model that learns to generate tactics as abstract syntax trees and can be used to prove new theorems beyond the reach of previous automatic provers.

2. Related Work

Automated theorem proving Modern theorem provers (Kovács & Voronkov, 2013; Schulz, 2013) represent theorems in first-order logic and search for proofs in resolution-based proof calculi. The proof search has been significantly improved by machine learning (Irving et al., 2016; Wang et al., 2017; Urban et al., 2011; Bridge et al., 2014; Loos et al., 2017; Kaliszyk et al., 2018; Rocktäschel & Riedel, 2017). However, it remains a challenging task due to the large search space; as a result, state-of-the-art provers do not scale to large problems. In contrast to traditional first-order provers, we focus on theorem proving in the Coq proof assistant, which represents theorems and manipulates proofs at a higher level, offering the unique opportunity of learning from human proofs.

Some proof assistants allow a user to use existing ATP systems directly. For example, Sledgehammer (Paulson & Blanchette, 2010) translates theorems in the Isabelle proof assistant (Paulson, 1994) to first-order logic. It then proves the theorems using external provers and converts the proof back to Isabelle's tactics. Similar "hammers" were developed for other proof assistants as well (Kaliszyk & Urban, 2014; Urban, 2004; Czajka & Kaliszyk, 2018). The hammer-based approach essentially bypasses the proof assistant and outsources the work to external ATPs. In contrast, our model learns to prove theorems within the proof assistant using native tactics without hammers.

Learning to interact with proof assistants There have been relatively few works in learning to interact with proof assistants. SEPIA (Gransden et al., 2015) learns from existing Coq proofs a finite-state automaton, where each transition corresponds to a tactic, and each path corresponds to a sequence of tactics. During test time, it samples tactic sequences defined by this automaton. Note that SEPIA can only choose from a finite set of tactics, and it tries the same tactics regardless of the theorem. That is, it will apply the tactic "`apply x`" even when `x` is not a valid term in the current context. In contrast, our model can generate an infinite number of tactics tailored to the current context.

TacticToe (Gauthier et al., 2018) generates tactics by retrieving from the training data a small number of candidate tactics that have been used for theorems similar to the current goal. Each candidate is treated as a possible action and evaluated by a learned value function and by Monte Carlo Tree Search. Although more adaptive than SEPIA, the generated tactics are still chosen from a fixed set with predetermined arguments and may have difficulty generalizing to new domains with out-of-vocabulary terms.

FastSMT (Balunovic et al., 2018) generates tactics to interact with the Z3 SMT solver (De Moura & Bjørner, 2008), which determines the satisfiability of a certain class of logical formulas. Compared to our model, FastSMT uses substantially simpler tactics—all of them have only boolean and integer arguments, whereas tactics in Coq can have compound expressions as arguments. As a result, the approach of FastSMT does not output ASTs and is not directly applicable to our setting.

Datasets for theorem proving Our work is related to many previous theorem proving datasets (Sutcliffe, 2009; Bancerek et al., 2015; Gransden et al., 2015; Gauthier et al., 2018; Huang et al., 2019). Our work differs from prior work in that we focus on theorems in higher-order logic and proofs consisting of high-level tactics as opposed to first-order logic and low-level proofs, and we aim for larger scale and more diverse domains.

The closest prior work is GamePad (Huang et al., 2019). GamePad includes a tool for interacting with Coq as well as data collected from the proof of the Feit Thompson theorem (Gonthier et al., 2013). We have independently developed a similar tool in CoqGym, but we aim for a larger-scale dataset. Our dataset contains 71K theorems, much more than the 1,602 theorems in Feit–Thompson, and our theorems come from a broad spectrum of 123 Coq projects. Another difference is that we augment our dataset with synthetic proofs extracted from the intermediate steps of human proofs. Finally, in terms of tactic generation, we generate complete tactics that can be used to obtain full proofs, whereas they group all tactics into categories and only predict the category, not the specific tactic. Also, they do not predict the location of the arguments in the tactics. Their method for full proof generation is specific to an algebraic rewrite problem, which has only two possible tactics, each with two integer arguments. Their model predicts one tactic (out of 2) and two integers, and is not directly applicable to our setting.

HOList (Bansal et al., 2019) is a concurrent work introducing a dataset and learning environment for the HOL Light proof assistant (Harrison, 1996). HOList consists of 29K proofs solely from the formalization of the Kepler conjecture (Hales et al., 2017), a theorem in discrete geometry, whereas CoqGym covers more diverse domains including not only pure mathematics but also computer systems. Similar to ours, HOList also introduces a model for tactic generation. But unlike ours, their method does not generate tactics in the form of ASTs.

Representing and generating programs Our model builds upon prior work that uses deep learning to represent and generate programs. A key ingredient is using a deep network to embed a program into a vector, by treating the program as a sequence of tokens (Allamanis et al., 2016b) or using structured input such as ASTs (Allamanis et al., 2016a; Alon et al., 2018) or graphs (Allamanis et al., 2017). We use a TreeLSTM (Tai et al., 2015) on ASTs to embed the input goal and premises.

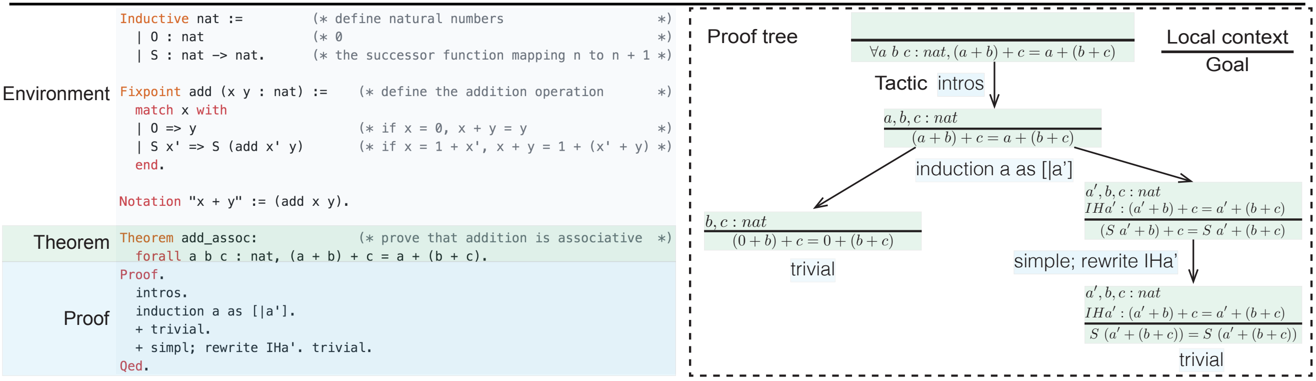
Similarly, a program can be generated by a deep network as a sequence of tokens (Hindle et al., 2012), an AST (Parisotto et al., 2016) or a graph (Brockschmidt et al., 2019). We adopt the framework of Yin & Neubig (2017) to generate tactics in the form of ASTs, conditioned on the goal and premises. However, in our specific task, we face the unique challenge of synthesizing the tactic arguments, which are subject to various semantic constraints, *e.g.*, the `H` in "`apply H`" must be a valid premise in the current context. Unlike the purely syntactic approach of Yin & Neubig (2017), our model utilizes the semantic constraints to narrow down the output space.

3. Background on Coq

Coq (Barras et al., 1997) is a proof assistant with an active community and diverse applications. It has been used to develop certified software and hardware (Leroy, 2009; Paulin-Mohring, 1995), and to prove theorems in mathematics, including the Feit–Thompson theorem (Gonthier et al., 2013). Under the hood of Coq are two pieces of machinery: a functional language for representing mathematical objects, theorems, and proofs, and a mechanism for constructing machine-checked proofs semi-automatically.

Coq allows us to define mathematical objects such as sets, functions, and relations. For example, we can define the set of natural numbers (nat) and the addition operation (add) in Fig. 1 (Left). These are examples of *terms* in Coq's language. The runtime *environment* of COq contains a set of current terms, including both user-defined terms and predefined terms from the Coq standard library. These terms are used to formalize theorems. As in Fig. 1, we state the theorem $\forall a\ b\ c : nat, (a + b) + c = a + (b + c)$ using nat and add.

Theorem proving in Coq is a backward process. The user starts with the theorem itself as the initial *goal* and repeatedly apply *tactics* to decompose the goal into a list of sub-



```
Inductive nat :=          (* define natural numbers                 *)
  | O : nat               (* 0                                      *)
  | S : nat -> nat.       (* the successor function mapping n to n + 1 *)

Fixpoint add (x y : nat) :=     (* define the addition operation       *)
  match x with
  | O => y                      (* if x = 0, x + y = y                 *)
  | S x' => S (add x' y)        (* if x = 1 + x', x + y = 1 + (x' + y) *)
  end.

Notation "x + y" := (add x y).

Theorem add_assoc:              (* prove that addition is associative  *)
  forall a b c : nat, (a + b) + c = a + (b + c).
Proof.
  intros.
  induction a as [|a'].
  + trivial.
  + simpl; rewrite IHa'. trivial.
Qed.
```

Figure 1. Left: A simple Coq proof for the associativity of the addition operation on natural numbers. *Right*: The proof tree generated by Coq when executing this proof. A Coq proof consists of a sequences of tactics. We start with the original theorem and apply tactics to decompose the current goal to sub-goals. This process generates a proof tree whose nodes are goals and whose edges are tactics.

goals (can be an empty list). The proof is complete when there are no sub-goals left. Proving is a process of trial and error; the user may try a tactic to decompose the current goal, analyze the feedback from Coq, and backtrack to the previous step to try a different tactic.

A successful Coq proof implicitly generates a *proof tree* whose root is the original theorem and whose nodes are goals (Fig. 1 *Right*). All goals share the same environment, but have a unique *local context* with premises local to each goal, such as the induction hypothesis `IHa'` in Fig. 1.

The edges of the proof tree are tactics; they can be simple strings, can have *arguments* at various positions, and can be combined into compound tactics. For example, `simpl` simplifies the current goal, "`apply H`" applies a premise `H`, and "`simpl; apply H`" performs these two operations sequentially. The space of all valid tactics is described by Ltac, Coq's tactic language (Delahaye, 2000).

From a machine learning perspective, theorem proving in Coq resembles a task-oriented dialog (Bordes et al., 2016). The agent interacts with the proof assistant for completing the proof. At each step, the agent perceives the current goals, their local context, and the global environment; it then generates an appropriate tactic, which is an expression in Ltac. Methods for task-oriented dialogues have been based on supervised learning (Bordes et al., 2016) or reinforcement learning (Liu et al., 2017). CoqGymprovides human-written proofs as supervision for training dialog agents. It also allows reinforcement learning on this task when combined with the tool for interacting with Coq.

4. Constructing CoqGym

CoqGym includes a large-scale dataset of 71K human-written proofs from 123 open-source software projects in Coq. In addition to the source files, we provide abstract syntax trees (ASTs) and rich runtime information of the proofs, including the environments, the goals, and the proof trees. Furthermore, we propose a novel mechanism for turning intermediate goals into theorems and synthesizing the corresponding proofs. These synthetic proofs may serve as additional training data. Further details are in Appendix A.

Processing Coq projects and files The source files are organized into projects, which contain a set of inter-related proofs about specific domains. The projects in CoqGym include the Coq standard library and the packages listed on the Coq Package Index[1]. Some of them may not compile because they require a specific version of Coq, or there is a missing dependency. We only include the projects that compile.

These projects are split into a training set, a validation set, and a test set. This ensures that no testing proof comes from a project that is used in training, and makes the dataset suitable for measuring how well the models generalize across various domains. There are 43,844 proofs for training, 13,875 proofs for validation and 13,137 proofs for testing.

We extract the ASTs from the internals of Coq's interpreter. They are OCaml datatypes. We serialize them into Lisp-style S-expressions (McCarthy, 1960) and provide tools for using them in Python.

Environments, goals, and proof trees Proofs are situated in environments containing Coq terms as premises. We could use the source code to represent the environments, as the code completely defines the environment. However, this is problematic for machine learning models, as it burdens them with learning the semantics of Coq code. Instead, we represent the environments as a collection of *kernel terms*, internal representations used by Coq stripped of syntactic sugar. This is achieved by executing the proofs and serializing Coq's internals.

In contrast to prior work (Huang et al., 2019), CoqGym supplies the complete environment for each proof—all the premises in the scope, including those defined in the same source file and those imported from other libraries. Having

[1] `https://coq.inria.fr/packages`

the complete environment is important because it allows the machine learning model to access all relevant information in structured forms.

We represent each proof as a proof tree, where a node is a goal along with its local context, and an edge is a tactic decomposing the goal into sub-goals. At each step in a proof, we serialize the current goals from Coq's interpreter and identify the edges in the proof tree by tracking how goals emerge and disappear during the lifetime of the proof.

Environments, goals, and proof trees together form a structured representation of Coq proofs. Compared to raw source code, a structured representation allow machine learning models to more easily exploit the syntactic and semantic structures. It is worth noting that this structured representation is nontrivial to extract because Coq does not provide APIs exposing its internals. In constructing CoqGym, we modify Coq and use SerAPI (Gallego Arias, 2016) to serialize the runtime information, without touching the core proof-checking module of Coq so as to not compromise the correctness of the proofs.

Synthetic proofs from intermediate goals Human-written proofs can be long and complex, making them difficult to learn from. We thus generate shorter proofs from the intermediate goals inside a long proof. We hypothesize that these intermediate goals are easier to prove and more conducive to learning. This also augments the training data with more examples.

For each intermediate goal in a human-written proof, we generate synthetic proofs of length 1, 2, 3, and 4. We detail the generation process in Appendix A.

Dataset statistics CoqGym has 70,856 human-written proofs from 123 Coq projects. On average, each proof has 8.7 intermediate goals and 9.1 steps; each step has 10,350.3 premises in the environment and 5.6 premises in the local context; each tactic has 2.0 tokens, and the height of its AST is 1.9. Among all tactics, 53% of them contain at least one argument. Note that these statistics vary significantly across different projects. For example, the average number of premises of a theorem is 13,340 in the CompCert project, but only 661 in the InfSeqExt project.

For the synthetic proofs, we have extracted 159,761 proofs of 1 step; 109,602 proofs of 2 steps; 79,967 proofs of 3 steps and 61,126 proofs of 4 steps.

5. ASTactic: generating tactics as programs

We propose a method that proves theorems by interacting with Coq. At the core of our method is ASTactic—a deep learning model that generates tactics as programs. Compared to prior work that chooses tactics from a fixed set (Huang et al., 2019; Gransden et al., 2015; Gauthier et al., 2018; Bansal et al., 2019), we generate tactics dynamically in the form of abstract syntax trees (ASTs) and synthesize arguments using the available premises during runtime. At test time, we sample multiple tactics from the model. They are treated as possible actions to take, and we search for a complete proof via depth-first search (DFS).

Space of tactics The output space of our model is specified by a context-free grammar (CFG) that is fixed during training and testing. Statistics of CoqGym show that many valid tactics (Delahaye, 2000) are seldom used in proofs. Therefore we simplify the tactic grammar to facilitate learning, at the expense of giving up on some cases. Note that these are design choices of the model, not the dataset—We train the model to generates tactics only in the simplified space, but the theorems in the testing data can require tactics outside this space.

We only generate atomic tactics while excluding compound ones such as "`tac1; tac2`". This is not a severe handicap because all proofs can be completed without compound tactics. For each type of tactic, we count its number of occurrences in CoqGym and manually include the common ones in our tactic space. When a tactic requires a Coq term as its argument, we constrain the term to be an identifier. We also exclude user-defined tactics. The complete CFG is in Appendix B.

Overall model architecture ASTactic has an encoder-decoder architecture (Fig. 2). The input and output of the model are both trees. The encoder embeds all input Coq terms: the goal and the premises expressed in ASTs. Conditioned on the embeddings, the decoder generates a program-structured tactic by sequentially growing an AST.

We follow prior works (Tai et al., 2015; Yin & Neubig, 2017) for encoding and decoding trees. The unique challenge in our task is to synthesize tactic arguments. In the decoder, we incorporate semantic constraints on the arguments to narrow down the search space.

Encoding the goal and premises ASTactic encodes the current goal and premises into vectors. We include the entire local context and up to 10 premises in the environment, excluding a large number of premises imported from libraries. A model could be more powerful if it is capable of selecting relevant premises from the entire environment, but that is left for future research.

Both the goal and the premises are Coq terms in the form of AST (Fig. 2 *Left*), and we encode them using a TreeLSTM network (Tai et al., 2015). Specifically, each node in an AST has a symbol $\mathbf{n}$ indicating its syntactical role. The network associate each node with a hidden state $\mathbf{h}$ and a memory cell $\mathbf{c}$ which are updated by its children as follows:

$$(\mathbf{c}, \mathbf{h}) = f_{update}(\mathbf{n}, \mathbf{c_1}, \cdots, \mathbf{c_K}, \sum_{i=1}^{K} \mathbf{h_i}),$$

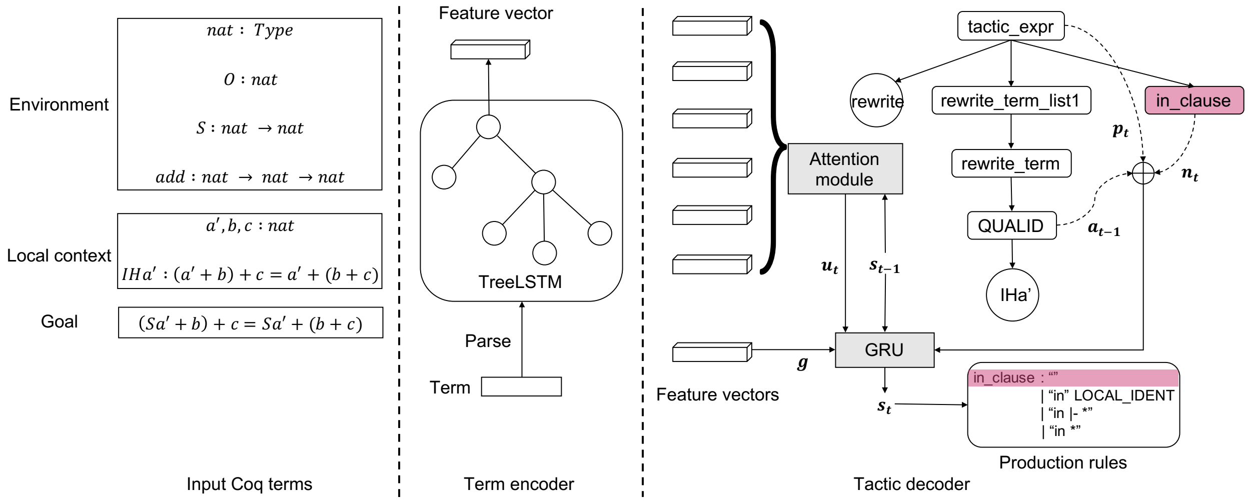


Figure 2. The architecture of ASTactic. It generates a tactic AST conditioned on the input Coq terms by sequentially expanding a partial tree. Here we illustrate a single expansion step of the non-terminal node `in_clause`. The ASTs of the input terms (*Left*) are encoded into feature vectors by a TreeLSTM network (Middle). A GRU controller then combines them with the information in the partial tree. It updates the decoder state $\mathbf{s_t}$ and uses $\mathbf{s_t}$ to predict the production rule to apply. In this example, the tactic AST is complete (`rewrite IHa'`) after expanding the current node.

where the update function f_{update} is the child-sum variant of TreeLSTM, $\mathbf{n}$ is the symbol of the node in one hot encoding, and $\mathbf{c_i}$ and $\mathbf{h_i}$ are the states of the children.

We perform this computation bottom-up and represent the entire tree by $\mathbf{h_{root}}$, the hidden state of the root. Finally, we append $\mathbf{h_{root}}$ with a 3-dimensional one hot vector; it indicates whether the term is the goal, a premise in the environment, or a premise in the local context.

Tracking the decoder state Conditioned on the input embeddings, the decoder (Fig. 2 *Right*) follows the method in Yin & Neubig (2017) to generate program-structured tactics as ASTs. It begins with a single node and grows a partial tree in the depth-first order. At a non-terminal node, it expands the node by choosing a production rule in the CFG of the tactic space. At a terminal node, it emits a token corresponding to a tactic argument.

This sequential generation process is controlled by a gated recurrent unit (GRU) (Cho et al., 2014), whose hidden state is updated by the input embeddings and local information in the partially generated AST.

Formally, we have learnable embeddings for all symbols and production rules in the tactic grammar. At time step t, let $\mathbf{n_t}$ be the symbol of the current node; $\mathbf{a_{t-1}}$ is the production rule used to expand the previous node; $\mathbf{p_t}$ is the parent node's state concatenated with the production rule used to expand the parent; $\mathbf{g}$ is the goal, which is fixed during the generation process. The state $\mathbf{s_t}$ is updated by:

$$\mathbf{s_t} = f_{GRU}(\mathbf{s_{t-1}}, [\mathbf{a_{t-1}} : \mathbf{p_t} : \mathbf{n_t} : \mathbf{g} : \mathbf{u_t}]) \qquad (1)$$

where ":" denotes vector concatenation. The $\mathbf{u_t}$ above is a weighted sum of premises. We use $\mathbf{s_{t-1}}$ to compute an attention mask on the premises, which selectively attends to the relevant premises for the current generation step. The mask is then used to retrieve $\mathbf{u_t}$:

$$w_i = f_{att}(\mathbf{s_{t-1}} : \mathbf{r_i}) \qquad (2)$$

$$\mathbf{u_t} = \sum_i w_i \mathbf{r_i} \qquad (3)$$

where $\mathbf{r_i}$ is the ith premise and w_i is its weight. f_{att} is a two-layer fully-connected network.

Expanding ASTs and synthesizing arguments The state $\mathbf{s_t}$ determines how to expand the tree including which production rules to apply and which tokens to generate.

To select a production rule, we model the probabilities for the rules as:

$$\mathbf{p_t} = \mathrm{softmax}(\mathbf{W_R} \cdot f(\mathbf{s_t})), \qquad (4)$$

where f is a linear layer followed by $tanh$, and $\mathbf{W_R}$ is the embedding matrix for production rules. We expand the node using the applicable rule with the largest probability.

The tokens in the ASTs correspond to the tactic arguments. Synthesizing them is challenging because the syntactic output space is large: all valid identifiers in Coq. However, there are strong semantic constraints on the arguments. For example, the tactic "`apply H`" applies a premise `H` to the goal. The argument `H` must be a valid premise either in the environment or in the local context.

To leverage the semantic constraints in synthesizing arguments, we group arguments into categories and take different actions for each category.

- Identifiers of premises (as in "`apply H`"): We score each premise using $\mathbf{s_t}$ in the same way as computing the attention masks (Equation. 3). A softmax on the scores gives us the probability for each premise.

- Integers (as in "`constructor 2`"): Most integers in the data are in the range of [1, 4]. We use a 4-way classifier to generate them.

- Quantified variables in the goal (as in "`simple induction n`"): We randomly pick a universally quantified variable in the goal.

Training and inference We train the model on the proof steps extracted from CoqGym. When expanding a node using a production rule, we apply the cross-entropy loss to maximize the likelihood of the ground truth production rule in Equation. 4. However, when the model emits a tactic argument, there may be no corresponding argument in the ground truth; because the model might have generated a different tactic from the ground truth. For example, the model may output "`apply H`" with an argument H, while the ground truth may be `split` without any argument.

To apply a reasonable loss in this scenario, we train the model with teacher forcing (Williams & Zipser, 1989). During the sequential generation of a tactic AST, the model outputs how to expand the partial tree, but the tree grows following the ground truth, not the model's output. Then the arguments generated by the model must correspond to those in the ground truth, and we can apply losses normally.

During testing, we combine the model with depth-first search (DFS) for fully-automated theorem proving. At each step, the model samples a few tactics via beam search, which are used to search for a complete proof via DFS. We prune the search space by backtracking when a duplicate proof state is detected.

Implementation details We use 256-dimensional vectors for all embeddings in ASTactic, including Coq terms ($\mathbf{g}$, $\mathbf{r_i}$), production rules ($\mathbf{a_{t-1}}$), GRU hidden state ($\mathbf{s_t}$), and symbols in the tactic grammar ($\mathbf{n_t}$). The training data includes 190K steps from human-written proofs. We do not train ASTactic with synthetic proofs since they only contain tactics extracted from the human proofs. For a method to benefit from synthetic proofs, it should model the entire sequence of tactics rather than an individual tactic.

We train the model using RMSProp (Tieleman & Hinton, 2012) with a learning rate of 3×10^{-5} and weight decay of 10^{-6}. The training goes for 5 epochs, which takes a few days on a single GeForce GTX 1080 GPU. During testing, our system performs beam search with a beam width of 20 to generate the top 20 tactics at each proof step. And we set a depth limit of 50 during DFS.

6. Experiments

Experimental setup We evaluate ASTactic on the task of fully-automated theorem proving in Coq, using the 13,137 testing theorems in CoqGym. The agent perceives the current goals, their local context, and the environment. It interacts with Coq by executing commands, which include tactics, backtracking to the previous step (`Undo`), and any other valid Coq command. The goal for the agent is to find a complete proof using at most 300 tactics and within a wall time of 10 minutes. We run all testing experiments on machines with 16GB RAM and two Intel Xeon Silver 4114 CPU Cores. We do not use GPUs for testing since the speed bottleneck is executing tactics rather than generating tactics.

We compare the performance of our system with several baselines. Our first set of baselines are Coq's built-in automated tactics, including `trivial`, `auto`, `intuition`, and `easy`. They all try to prove the theorem via some simple procedures such as backward chaining. The second baseline is `hammer` (Czajka & Kaliszyk, 2018)—a hammer-based system that proves theorems using external ATP systems. In our particular configuration, `hammer` simultaneously invokes Z3 (De Moura & Bjørner, 2008), CVC4 (Barrett et al., 2011), Vampire (Kovács & Voronkov, 2013), and E Prover (Schulz, 2013), and returns a proof as long as one of them succeeds.

If we treat `hammer` as a black box tactic, it sets a default time limit of 20 seconds to the external ATP systems. We test `hammer` both in this setting and in a setting where we extend the time limit to 10 minutes.

All of these automated tactics either prove the goal completely or leave the goal unchanged; they do not decompose the goal into sub-goals. We combine ASTactic with them as follows: At each step, the agent first uses an automated tactic (*e.g.* `hammer`) to see if it can solve the current goal. If not, the agent executes a tactic from ASTactic to decompose the goal into sub-goals.

Table 1. Percentage of theorems successfully proved. Our method significantly outperforms Coq's built-in automated tactics. It achieves the highest success rate when combined with `hammer`. The default time limit for `hammer` is 20 seconds and the extended time limit is 10 minutes.

Method	Success rate (%)
trivial	2.4
auto	2.9
intuition	4.4
easy	4.9
hammer (default time limit)	17.8
hammer (extended time limit)	24.8
ours	12.2
ours + auto	12.8
ours + hammer	**30.0**

Success rates Table 1 shows the percentage of theorems successfully proved. Our system proves 12.2% of the theorems, while the built-in automated tactics in Coq prove less than 4.9%. While our model underperforms `hammer` (24.8% with extended time limit), its performance is nonetheless very encouraging considering that `hammer` invokes four state-of-the-art external ATP systems that took many years to engineer whereas our model is trained from scratch with a very limited amount of hand engineering. When combined with `hammer`, ASTactic can prove 30.0% of the theorems, a large improvement of 5.2% over using `hammer` alone. This demonstrates that our system can generate effective tactics and can be used to prove theorems previously not provable by automatic methods.

Table 2. The effect of the beam width on the success rate and the average runtime for proving a theorem.

Beam width	1	5	10	15	20	25
Success rate (%)	1.0	6.5	10.8	12.0	12.2	11.7
Average runtime (seconds)	0.2	1.2	2.2	2.7	3.3	3.9

Efficiency The beam width is the number of candidate tactics to explore at each proof step; it thus controls the trade-off between speed and accuracy. Table 2 shows how the beam width affects the runtime and the number of proved theorems. A large beam width increases the success rate as it enables the model to explore larger search space at the expense of longer runtime. However, when the beam width goes above 20, the success rate drops, probably due to the model being trapped in an unpromising branch for too long.

Fig. 3 (*Left*) illustrates the runtime of various methods. The built-in automated tactics are faster, but they can prove only a few hundred theorems. ASTactic combined with `hammer` takes less than 100 seconds per theorem for proving over 3,000 theorems.

Fig. 3 (*Right*) shows the number of tactics tried before successfully finding a proof. Compared to SEPIA (Gransden et al., 2015), which uses 10,000 tactics, ASTactic is more efficient in exploring the tactic space, needing only a few hundred tactics. When combined with `hammer`, it typically finds a proof within 10 tactics.

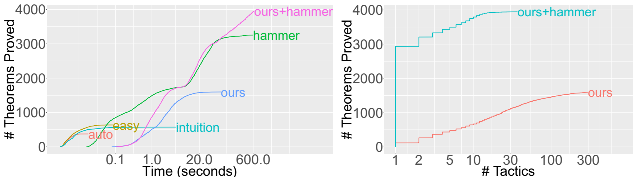


Figure 3. The number of proved theorems increases with the runtime (*Left*) and the number of allowed tactics (*Right*).

Generated proofs Fig. 4 shows the lengths of the generated proofs compared to all ground truth proofs in the testing set. As expected, most generated proofs are short (less than 10 steps). The average length of the generated proofs is 6.0 while the average of the entire testing set is 12.5, which suggests that theorems with longer proofs are much more challenging for the model.

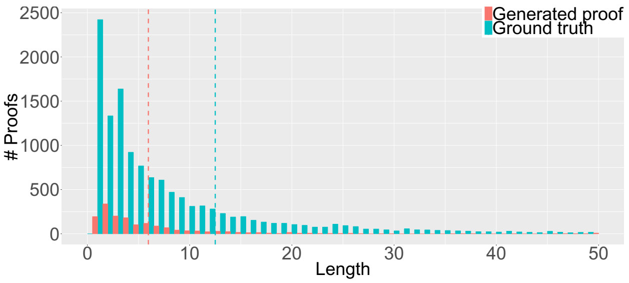


Figure 4. The lengths (the number of steps) of the generated proofs compared to all proofs in the testing set. The dash lines are the average lengths (*6.0* and *12.5* respectively).

Even for the same theorem, a generated proof can be much shorter than the ground truth. Fig. 5 is one such example. The generated proof calls a decision procedure (`ring`) to solve the goal with fewer tactics. This also reflects the fact that longer proofs are harder to find.

```
Lemma Rmult_minus_distr_r:
  forall r1 r2 r3 : R, (r1 - r2) * r3 = r1 * r3 - r2 * r3.

(* ground truth *)
Proof.
  intros.
  unfold Rminus.
  rewrite <- Ropp_mult_distr_l_reverse.
  apply Rmult_plus_distr_r.
Qed.

(* prediction *)
Proof.
  intros.
  ring.
Qed.
```

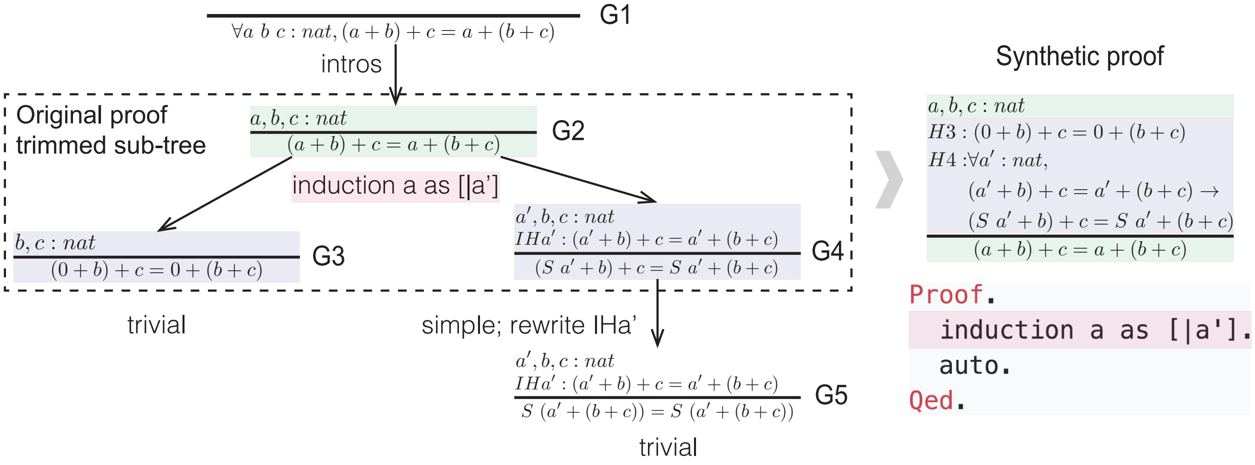
Figure 5. An example proof generated by our method. It is shorter than the ground truth thanks to the `ring` decision procedure.

7. Conclusion

We address the problem of learning to prove theorems in Coq. We have constructed CoqGym—a large-scale dataset and learning environment with human-written proofs from a broad spectrum of Coq projects. We have developed ASTactic, a deep-learning based model that generates Coq tactics in the form of AST. Experimental results on CoqGym confirm the effectiveness of our model for synthesizing complete proofs automatically.

Acknowledgements

This work is partially supported by the National Science Foundation under Grant No. 1633157.



```
Proof.
  induction a as [|a'].
  auto.
Qed.
```

Figure F. Extracting a synthetic proof from the intermediate goal G2. Goals G3 and G4 are converted into premises in G2's local context. The synthetic proof corresponds to a trimmed sub-tree rooted at G2.

Appendix

A. Details on Constructing the Dataset

A.1. Building the Coq projects

We manually compile and install the Coq standard library and a few projects (such as math-comp) that are frequently required by other projects. For the rest, we try compiling them automatically using simple commands such as "`./configure && make`", and we take whatever compiles, ending up with 123 projects and 3,061 Coq files (excluding the files that do not contain any proof).

A.2. Reconstructing the Proof Tree

After applying a tactic, the current goal disappears, and a set of new goals emerge, which become the children of the current goal in the proof tree. We can identify the edges of the tree by tracking how goals emerge during the proof. For example, if the list of goals changes from [2, 7] to [8, 9, 7], we know that node 2 has two children: 8 and 9.

In certain cases, a tactic can affect more than one goal, and it is unclear who should be the parent node. This can happen when a tactic is applied to multiple goals using a language feature called goal selectors (by default, a tactic is applied only to the first goal). However, goal selectors are rarely used in practice. We discard all such proofs and lose only less than 1% of our data. For the remaining data, only one goal disappears at each step, and we can build the proof trees unambiguously.

A.3. Extracting Synthetic Proofs from Intermediate Goals

Given an intermediate goal, it is straightforward to treat it as a theorem by adding its local context to the environment. For example, in Fig. F, the goal G2 can be a theorem $(a+b)+c = a+(b+c)$ in the environment augmented by a b and c. Extracting synthetic proofs for the new theorem requires nontrivial processing. One straightforward proof would be the sequence of tactics that follows G2 in the original human-written proof: "`induction a as [|a']. trivial. simpl; rewrite IHa'. trivial.`". This proof corresponds to the sub-tree rooted at G2.

However, there are potentially shorter proofs for G2 using a trimmed sub-tree. For example, if we only apply the first tactic to generates G3 and G4, then we can treat them as premises H3 and H4, and complete the proof by "`apply H3. apply H4.`". Equivalently, we can also use `auto` to complete the proof. This technique of converting unsolved sub-goals into premises allows us to generate synthetic proofs of controllable lengths, by taking a sequence of tactics from the original proof and appending an `auto` at the end.

We need to take extra care in converting a goal into a premise. For example, it is easy to treat G3 as a premise, but G4 needs some care. G4 depends on `a'`, which is missing in G2's context. In order to convert G4 into a well-formed term in G2's context, we apply the "`generalize dependent`" tactic to push a local premise into the statement of the goal. When applied to G4, it generates H4 in Fig. F, which can be added to G2's local context.

B. The Space of Tactics for ASTactic

Below is the context-free grammar in extended Backus-Naur form for the tactic space. The start symbol is *tactic_expr*.

```
tactic_expr : intro
            | ‘apply’ term_commalist1 reduced_in_clause
            | ‘auto’ using_clause with_hint_dbs
            | ‘rewrite’ rewrite_term_list1 in_clause
            | ‘simpl’ in_clause
            | ‘unfold’ qualid_list1 in_clause
            | destruct
            | induction
            | ‘elim’ QUALID
            | ‘split’
            | ‘assumption’
            | trivial
            | ‘reflexivity’
            | ‘case’ QUALID
            | clear
            | ‘subst’ local_ident_list
            | ‘generalize’ term_list1
            | ‘exists’ LOCAL_IDENT
            | ‘red’ in_clause
            | ‘omega’
            | discriminate
            | inversion
            | simple_induction
            | constructor
            | ‘congruence’
            | ‘left’
            | ‘right’
            | ‘ring’
            | ‘symmetry’
            | ‘f_equal’
            | ‘tauto’
            | ‘revert’ local_ident_list1
            | ‘specialize’ ‘(’ LOCAL_IDENT QUALID ‘)’
            | ‘idtac’
            | ‘hnf’ in_clause
            | inversion_clear
            | contradiction
            | ‘injection’ LOCAL_IDENT
            | ‘exfalso’
            | ‘cbv’
            | ‘contradict’ LOCAL_IDENT
            | ‘lia’
            | ‘field’
            | ‘easy’
            | ‘cbn’
            | ‘exact’ QUALID
            | ‘intuition’
            | ‘eauto’ using_clause with_hint_dbs

LOCAL_IDENT : /[A–Za–z_][A–Za–z0–9_’]*/
```

```
QUANTIFIED_IDENT : /[A–Za–z_][A–Za–z0–9_']*/

INT : /1|2|3|4/

QUALID : /([A–Za–z_][A–Za–z0–9_']*\.)*[A–Za–z_][A–Za–z0–9_']*/

HINT_DB : /arith|zarith|algebra|real|sets|core|bool|datatypes|coc|set|zfc/

local_ident_list :
                 | LOCAL_IDENT local_ident_list

local_ident_list1 : LOCAL_IDENT
                  | LOCAL_IDENT local_ident_list1

qualid_list1 : QUALID
             | QUALID ',' qualid_list1

term_list1 : QUALID
           | QUALID term_list1

term_commalist1 : QUALID
                | QUALID ',' term_commalist1

hint_db_list1 : HINT_DB
              | HINT_DB hint_db_list1

reduced_in_clause :
                  | 'in' LOCAL_IDENT

in_clause :
          | 'in' LOCAL_IDENT
          | 'in' '|- *'
          | 'in' '*'

at_clause :
          | 'at' INT

using_clause :
             | 'using' qualid_list1

with_hint_dbs :
              | 'with' hint_db_list1
              | 'with' '*'

intro : 'intro'
      | 'intros'

rewrite_term : QUALID
             | '→' QUALID
             | '←' QUALID

rewrite_term_list1 : rewrite_term
                   | rewrite_term ',' rewrite_term_list1
```

```
destruct : ‘destruct’ term_commalist1

induction : ‘induction’ LOCAL_IDENT
          | ‘induction’ INT

trivial : ‘trivial’

clear : ‘clear’
      | ‘clear’ local_ident_list1

discriminate : ‘discriminate’
             | ‘discriminate’ LOCAL_IDENT

inversion : ‘inversion’ LOCAL_IDENT
          | ‘inversion’ INT

simple_induction : ‘simple induction’ QUANTIFIED_IDENT
                 |  ‘simple induction’ INT

constructor : ‘constructor’
            | ‘constructor’ INT

inversion_clear : ‘inversion_clear’ LOCAL_IDENT
                | ‘inversion_clear’ INT

contradiction : ‘contradiction’
              | ‘contradiction’ LOCAL_IDENT
```

References

Allamanis, M., Chanthirasegaran, P., Kohli, P., and Sutton, C. Learning continuous semantic representations of symbolic expressions. *arXiv preprint arXiv:1611.01423*, 2016a.

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pp. 2091–2100, 2016b.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *arXiv preprint arXiv:1803.09473*, 2018.

Balunovic, M., Bielik, P., and Vechev, M. Learning to solve smt formulas. In *Advances in Neural Information Processing Systems*, pp. 10317–10328, 2018.

Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pak, K., and Urban, J. Mizar: State-of-the-art and beyond. In *Conferences on Intelligent Computer Mathematics*, pp. 261–279. Springer, 2015.

Bansal, K., Loos, S. M., Rabe, M. N., Szegedy, C., and Wilcox, S. Holist: An environment for machine learning of higher-order theorem proving (extended version). *arXiv preprint arXiv:1904.03241*, 2019.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.-C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., and Tinelli, C. CVC4. In Gopalakrishnan, G. and Qadeer, S. (eds.), *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pp. 171–177. Springer, July 2011. URL http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf. Snowbird, Utah.

Bordes, A., Boureau, Y.-L., and Weston, J. Learning end-to-end goal-oriented dialog. *arXiv preprint arXiv:1605.07683*, 2016.

Bridge, J. P., Holden, S. B., and Paulson, L. C. Machine learning for first-order theorem proving. *Journal of automated reasoning*, 53(2):141–172, 2014.

Brockschmidt, M., Allamanis, M., Gaunt, A. L., and Polozov, O. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Czajka, Ł. and Kaliszyk, C. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018.

Darvas, Á., Hähnle, R., and Sands, D. A theorem proving approach to analysis of secure information flow. In *International Conference on Security in Pervasive Computing*, pp. 193–209. Springer, 2005.

De Moura, L. and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Delahaye, D. A tactic language for the system coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 85–95. Springer, 2000.

Dixon, L. and Fleuriot, J. Isaplanner: A prototype proof planner in isabelle. In *International Conference on Automated Deduction*, pp. 279–283. Springer, 2003.

Gallego Arias, E. J. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, October 2016. URL https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408.

Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., and Norrish, M. Learning to prove with tactics. *arXiv preprint arXiv:1804.00596*, 2018.

Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., OConnor, R., Biha, S. O., et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pp. 163–179. Springer, 2013.

Gransden, T., Walkinshaw, N., and Raman, R. Sepia: search for proofs using inferred automata. In *International Conference on Automated Deduction*, pp. 246–255. Springer, 2015.

Hales, T., Adams, M., Bauer, G., Dang, T. D., Harrison, J., Le Truong, H., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T. T., et al. A formal proof of the kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

Harrison, J. Hol light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design*, pp. 265–269. Springer, 1996.

Harrison, J., Urban, J., and Wiedijk, F. History of interactive theorem proving. In *Computational Logic*, volume 9, pp. 135–214, 2014.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 837–847. IEEE, 2012.

Huang, D., Dhariwal, P., Song, D., and Sutskever, I. Gamepad: A learning environment for theorem proving. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=r1xwKoR9Y7.

Irving, G., Szegedy, C., Alemi, A. A., Eén, N., Chollet, F., and Urban, J. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pp. 2235–2243, 2016.

Kaliszyk, C. and Urban, J. Learning-assisted automated reasoning with flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

Kaliszyk, C., Urban, J., Michalewski, H., and Olšák, M. Reinforcement learning of theorem proving. *arXiv preprint arXiv:1805.07563*, 2018.

Kern, C. and Greenstreet, M. R. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

Kovács, L. and Voronkov, A. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pp. 1–35. Springer, 2013.

Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Liu, B., Tur, G., Hakkani-Tur, D., Shah, P., and Heck, L. End-to-end optimization of task-oriented dialogue model with deep reinforcement learning. *arXiv preprint arXiv:1711.10712*, 2017.

Loos, S., Irving, G., Szegedy, C., and Kaliszyk, C. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

McCarthy, J. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

McCune, W. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

Parisotto, E., Mohamed, A.-r., Singh, R., Li, L., Zhou, D., and Kohli, P. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

Paulin-Mohring, C. Circuits as streams in coq: Verification of a sequential multiplier. In *International Workshop on Types for Proofs and Programs*, pp. 216–230. Springer, 1995.

Paulson, L. C. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.

Paulson, L. C. and Blanchette, J. C. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *PAAR@ IJCAR*, pp. 1–10, 2010.

Rocktäschel, T. and Riedel, S. End-to-end differentiable proving. In *Advances in Neural Information Processing Systems*, pp. 3788–3800, 2017.

Schulz, S. System description: E 1.8. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 735–743. Springer, 2013.

Sutcliffe, G. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Tieleman, T. and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Urban, J. Mptp–motivation, implementation, first experiments. *Journal of Automated Reasoning*, 33(3-4):319–339, 2004.

Urban, J., Vyskočil, J., and Štěpánek, P. Malecop machine learning connection prover. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 263–277. Springer, 2011.

Wang, M., Tang, Y., Wang, J., and Deng, J. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pp. 2786–2796, 2017.

Williams, R. J. and Zipser, D. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

Yin, P. and Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.