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
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 & Björner, 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...
Learning to Prove Theorems via Interacting with Proof Assistants

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 & Björner, 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 tactics, 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 ∀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-
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 \( \text{IH}a' \) 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 \( \text{H} \)” applies a premise \( \text{H} \), and “\( \text{simp} \) apply \( \text{H} \)” 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). CoqGym provides 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 the supplementary material.

**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 Index¹. 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 (McC, 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

---

¹[https://coq.inria.fr/packages](https://coq.inria.fr/packages)
We propose a method that proves theorems by interacting with Coq. 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 the supplementary material.

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 n indicating its syntactical role. The network associates each node with a hidden state h and a memory
Learning to Prove Theorems via Interacting with Proof Assistants

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 \( s_t \) and uses \( s_{t-1} \) to predict the production rule to apply. In this example, the tactic AST is complete (\`rewrite \\{H'\}) after expanding the current node.

cell \( c \) which are updated by its children as follows:

\[ (c, h) = f_{\text{update}}(n, c_1, \ldots, c_K, \sum_{i=1}^{K} h_i) \]

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

We perform this computation bottom-up and represent the entire tree by \( h_{\text{root}} \), the hidden state of the root. Finally, we append \( h_{\text{root}} \) 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 \( n_t \) be the symbol of the current node; \( a_{t-1} \) is the production rule used to expand the previous node; \( p_t \) is the parent node’s state concatenated with the production rule used to expand the parent; \( g \) is the goal, which is fixed during the generation process. The state \( s_t \) is updated by:

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

where \( \cdot \) denotes vector concatenation. The \( u_t \) above is a weighted sum of premises. We use \( s_{t-1} \) 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 \( u_t \):

\[ w_i = f_{\text{att}}(s_{t-1} : r_i) \]

\[ u_t = \sum_i w_i r_i \]

where \( r_i \) is the \( i \)th premise and \( w_i \) is its weight. \( f_{\text{att}} \) is a two-layer fully-connected network.

**Expanding ASTs and synthesizing arguments** The state \( s_t \) 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:

\[ p_t = \text{softmax}(W_R : f(s_t)) \]

where \( f \) is a linear layer followed by \( \tanh \), and \( W_R \) 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.
We train the model using RMSProp (Tieleman & Hinton, 2012) with a learning rate of \( 3 \times 10^{-5} \) 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.

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
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.

<table>
<thead>
<tr>
<th>Method</th>
<th>Success rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>trivial</td>
<td>2.4</td>
</tr>
<tr>
<td>auto</td>
<td>2.9</td>
</tr>
<tr>
<td>intuition</td>
<td>4.4</td>
</tr>
<tr>
<td>easy</td>
<td>4.9</td>
</tr>
<tr>
<td>hammer (default time limit)</td>
<td>17.8</td>
</tr>
<tr>
<td>hammer (extended time limit)</td>
<td>24.8</td>
</tr>
<tr>
<td>ours</td>
<td>12.2</td>
</tr>
<tr>
<td>ours + auto</td>
<td>12.8</td>
</tr>
<tr>
<td>ours + hammer</td>
<td>30.0</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Beam width</th>
<th>Success rate (%)</th>
<th>Average runtime (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0.2</td>
</tr>
<tr>
<td>5</td>
<td>6.5</td>
<td>1.2</td>
</tr>
<tr>
<td>10</td>
<td>10.8</td>
<td>2.2</td>
</tr>
<tr>
<td>15</td>
<td>12.0</td>
<td>2.7</td>
</tr>
<tr>
<td>20</td>
<td>12.2</td>
<td>3.3</td>
</tr>
<tr>
<td>25</td>
<td>11.7</td>
<td>3.9</td>
</tr>
</tbody>
</table>

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.

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.

References


Learning to Prove Theorems via Interacting with Proof Assistants


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.


