Abstract—Formal verification is an effective but extremely
work-intensive method of improving software quality. Verifying
the correctness of software systems often requires significantly
more effort than implementing them in the first place, despite
the existence of proof assistants, such as Coq, aiding the process.
Recent work has aimed to fully automate the synthesis of formal
verification proofs, but little tool support exists for practitioners.
This paper presents Proofster, a web-based tool aimed at assisting
developers with the formal verification process via proof synthesis.
Proofster inputs a Coq theorem specifying a property of a
software system and attempts to automatically synthesize a formal
proof of the correctness of that property. When it is unable to
produce a proof, Proofster outputs the proof-space search tree
its synthesis explored, which can guide the developer to provide a
hint to enable Proofster to synthesize the proof. Proofster runs
online at https://proofster.cs.umass.edu/ and a video demonstrating
Proofster is available at https://youtu.be/xQAi660RfwI/.

I. INTRODUCTION

Software bugs are so routine that the annual cost of
operational software failures is more than $1.56 trillion [15],
and software engineers spend 35–50% of their time validating
and debugging software [18]. Formal verification is a promising
method for building correct software systems. Proof assistants,
such as Coq [28] and HOL4 [25], inherently support program
verification and have had significant industrial impact. For
example, Airbus France uses the Coq-verified CompCert C
compiler [16] to ensure safety and improve performance of
its aircraft [26], Chrome, Android, and Firefox use verified
cryptographic libraries [5], [13], and Amazon Web Services
applies formal verification to detect misconfigurations that can
compromise cloud security [1].

Unfortunately, formal verification is challenging. Writing
proofs in Coq is a painstaking exercise that requires deep
expertise, as seen in the engineering processes behind several
large proof developments [12], [29]. Even with the help of
an Interactive Theorem Prover, the effort required to write
proofs is often prohibitive. The Coq proof of the C compiler
is more than three times that of the compiler code itself [16].

Meanwhile, it took 11 person-years to write the proofs required
to verify the seL4 microkernel [17], which represents a tiny
fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing
proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30].
Some formal verification can even be fully automated via
proof synthesis. For example, CoqHammer [4] uses a set of
precomputed mathematical facts to attempt to “hammer” out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23],
TacTok [7], Diva [6], and Passport [24] learn a predictive model
from a corpus of existing proofs and use that model to guide
a meta-heuristic search to synthesize a proof from scratch.

Unfortunately, relatively little tool support exists for practitioners to use these Coq proof-synthesis tools. For example, of
the above-mentioned search-based tools, all but one have neither
been integrated into IDEs nor built as stand-alone, graphical
interfaces, making adoption difficult. Only Tactician [2] has
a usable interface, by way of a plugin for Coq that can be integrated into Coq IDEs. But even then, the interface does not
expose the features that help the user understand what the tool
is doing under the hood, making debugging and explainability
difficult.

In this paper, we present Proofster, a new graphical frontend
for search-based proof-synthesis techniques that emphasizes
explainability. Conceptually, Proofster can be straightforwardly
extended to work with any proof-synthesis backend tool,
and implements special features to support explainability for
search-based backends. Here, we demonstrate Proofster with
Proverbot9001 [23] as its backend.

Proofster’s main contributions support the developer in two
ways:

1) The developer can enter a theorem describing a software
property they want proven, and Proofster uses its underlying
backend to attempt to generate a proof. If successful, Proofster displays the Coq proof script, verifying that the
property is correct. Proofster uses the Alectryon library to
render literate Coq code [20], which is interactive and easy to read, even when one does not have immediate access to a proof assistant to step through the synthesized proof. The developer can explore the context throughout the proof to better understand why the property is verifiably correct.

2) If the synthesis is unsuccessful, Proofster uses the D3.js library [3] to allow the developer to interactively explore the search tree it used in trying to synthesize a proof, and understand the relevant context. The developer can then identify the most promising search-path, augment it, and have Proofster attempt to synthesize a proof again, using that information.

A live Proofster deployment is available at https://proofster.cs.umass.edu/.

II. PROOFSTER

Proofster is a frontend tool that interfaces with Coq-based proof synthesis tools. Section II-A discusses how proof engineers interactively write proofs in Coq and how machine-learning-guided proof synthesis tools automatically generate proofs. Section II-B then describes the Proofster implementation and Section II-C illustrates, with examples, how a proof engineer can use Proofster to construct proofs.

A. Proofs and proof synthesis in Coq

When using the Coq proof assistant, a developer begins by specifying a theorem to prove. This theorem is a type definition in Coq’s internal language, Gallina. A proof of that theorem is a term of that type. However, writing that proof term directly is difficult, and so Coq provides an interactive environment for reasoning through a proof at a higher level, via a proof script.

The developer can use Coq’s Ltac language to construct a proof script, a sequence of tactics which Coq uses to guide its internal search for a Gallina-based proof term. The theorem prover is called interactive, because the developer can specify a tactic to try, have the theorem prover execute the tactic to update the proof state (the set of goals that need to be proven, and the known facts), and use that proof state to decide on the next tactic. This interactive process continues until no goals remain, meaning the theorem is proven.

The burden is on the developer to come up with the sequence of tactics. To ease this burden, recent work has created search-based, machine-learning-guided proof synthesis tools that perform automatic proof-script generation. Most of these tools train a predictive model on a corpus of human-written proof scripts. This model uses a partially written proof script and the theorem being proven to predict a ranked list of the most likely next tactics that should come in the proof script.

The tools differ in how they model the proof scripts when making predictions. For example, ASTactic considers only the current proof state (and ignores the current, partial proof script) [30]. TacTok is a collection of two models—Tac and Tok—both of which encode both the proof state and the partial proof script. Tac works at the tactic granularity, whereas Tok works at the token granularity; the two prove complementary sets of theorems [7]. These tools model abstract syntax trees using TreeLSTM [27] and proof-script sequences using bidirectional LSTM [19], whereas Proverbot9001, which also models proof state and partial proof script, uses a sequence model [23]. Passport further enhances the model by encoding identifier information for the names of theorems, datatypes, functions, type constructors, and local variables [24]. GamePad, meanwhile, uses its own RNN-based tree encoder and targets only synthetic lemmas [11]. Finally, Diva observes that the variability inherent in machine learning—small perturbations in the learning process, such as hyperparameters, the order in which the training data is seen, and the encoded richness of the training data—leads to diversity in the sets of theorems the learned models can prove. Using the theorem prover’s unique ability to serve as an oracle for correctness, Diva uses this diversity to significantly increase its proving power [6].

Armed with a predictive model, these search-based tools search through the space of possible proof scripts. They use the model to predict the likely next proof steps, and the theorem prover to compute the new proof states or errors resulting from these steps. They prune search paths unlikely to be successful or that repeat an already explored state: Proverbot9001, in particular, also prunes states that would explore a subgoal for which a solution was already found. This search through the space of proof scripts represents a set of potential partial proof scripts that aim to make progress toward the goal of proving the theorem. We call the set of explored search paths, together, the search tree.

B. The Proofster implementation

Proofster is implemented as a Flask app and uses BeautifulSoup to create the results page with the synthesized proof and the search graph. Proofster allows the developer to enter a theorem into a text box (or select one from several examples, as a demonstration). Proofster then passes the developer-specified theorem to its proof-synthesis backend and retrieves the search tree, and, if the backend is successful, the synthesized proof. Proofster then uses Alectryon to render the proof as an interactive, literate Coq object. Hovering over a tactic displays the context and goals at that stage of the proof.

Proofster uses the D3.js library display the search tree and allow the developer to interact with it. Subtrees can be collapsed and expanded to see the tactics tried by the proof synthesis model. This information can also be helpful to developers to provide hints to Proofster in the case where Proofster fails to prove the theorem initially.

Proofster is deployed on AWS and is publicly available at https://proofster.cs.umass.edu/. Proofster is open-source, and is publicly available at https://github.com/UCSD-PL/proverbot9001/tree/demowebtool.

Next, we illustrate Proofster’s two use cases using examples.

C. Using Proofster

Supposed a developer has written a function, max_elem_list, that takes a list of natural numbers and returns its largest element. The developer would like to verify this function’s
Fig. 1. A Proofster screenshot of the developer asking to prove the theorem \textit{every_elem_le_max} about the function \textit{max_elem_list}.

Fig. 2. When Proofster executes the query from Figure 1, it produces a complete proof for the theorem \textit{every_elem_le_max}. Hovering over a tactic in the proof shows the proof state at that point in the proof, which allows the developer to explore and understand how the proof verifies the property.

Fig. 3. A Proofster screenshot of the developer asking to prove the theorem \textit{list_forall2}.

Fig. 4. When Proofster executes the query from Figure 3, it is not able to generate a complete proof, but displays its search tree, instead. (Image has been rotated for space.)

The developer decides to use Proofster to prove the above property, in Coq. She heads over to \url{https://proofster.cs.umass.edu/} and enters some basic imports, the definition of the \textit{max_elem_list} function, and the theorem \textit{every_elem_le_max}. She does not enter the proof of the theorem, but only starts it with \texttt{Proof.} and \texttt{Admitted.} to tell Proofster to generate a proof for that theorem. (Proofster will replace \texttt{Admitted.} with the proof.)

Figure 1 shows a Proofster screenshot with the developer's inputs. Clicking “Proofster it!” tells Proofster to run its backend to attempt to generate a proof. It succeeds, and Proofster displays the full proof (partial screenshot in Figure 2).

The backend will not always be able to produce a proof fully automatically. Suppose the developer wants to verify another property. Given two lists, let proposition \( p \) be a proposition on two elements, and let theorem \textit{list_forall2_app} say that proposition \( p \) holds for every pair formed by zipping the two lists together. Suppose the developer wants to then prove another property, captured by theorem \textit{list_forall2_app}, which states that for all lists \( a_1, a_2, b_1, b_2 \), if \textit{list_forall2} holds for \( a_1, b_1 \) and for \( a_1, b_2 \), then it also holds for the pair of lists formed by appending \( a_1 \) and \( a_2 \), and appending \( b_1 \) and \( b_2 \).

Figure 3 shows the query the developer submits to Proofster to prove this theorem. However, Proofster’s backend fails to automatically synthesize a proof for this theorem. Instead of a proof, Proofster displays the search tree for the developer to investigate (Figure 4). She sees that Proofster tried a few forms of induction on the input lists and gets an idea: perhaps inducting over terms of the relation between lists \textit{list_forall2} \hspace{1mm} \( a_1 \hspace{1mm} b_1 \), rather than over the lists directly, will result in a more informative inductive hypothesis. The developer returns to the query page and suggests a hint for Proofster: \textit{induction 1}, which inducts over the first unnamed hypothesis (here, the term of type \textit{list_forall2} \( a_1 \hspace{1mm} b_1 \)), something Proofster had
failed to try. She then admits the rest and queries Proofster. Armed with this hint, Proofster synthesizes the correct proof (Figure 5).

D. Evaluation Plan

We plan to evaluate Proofster by soliciting feedback from developers, and by using it in a proof engineering graduate class. Proofster’s backends have been thoroughly evaluated on a benchmark of 68K Coq theorems from 122 open-source projects. AsTactic can fully automatically prove 12.3% of the theorems [30]. Passport 12.7% [24], TacTok 12.9 [7], Proverbot9001 [23] 19.2%, and Diva 21.7% [6]. Together with CoqHammer, these tools can prove more than 33% of the theorems.

III. RELATED WORK

The Proofster web interface provides an environment to interactively explore both the synthesized proof, and the synthesis search process. It uses the Alectryon [20] library to render literate Coq code, which is interactive and easy to read, even when one does not have immediate access to a proof assistant to step through the synthesized proof. jsCoq [8] and PeaCoq [22] also allow you to interact with formal proofs via web interfaces, but neither synthesize proofs. Tactician tactic-learning Coq plugin can be accessed through a web demonstration of two examples using jsCoq [2]. Section 7.1 of “QED at Large” [21] provides a thorough survey of user interfaces for formal proofs.

Automatically synthesizing proofs from scratch is a promising direction in easing formal verification [4], [7], [6], [23], [24], [30], jointly proving more than 33% of a large proof benchmark [6]. However, these efforts have not yet directly addressed usability and adoption, which is Proofster’s goal.

REFERENCES


