PRoofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

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

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA
shizhuo2@illinois.edu

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA
ttz2@illinois.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

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

***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/xQAi66lRfwI/.**

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 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

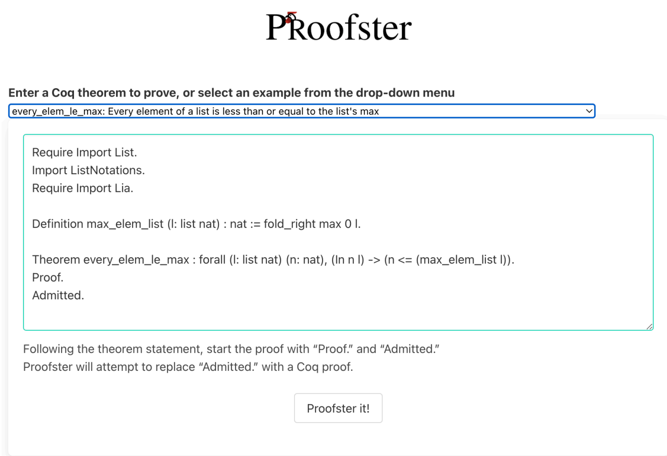

Fig. 1. A PRoofster screenshot of the developer asking to prove the theorem **every_elem_le_max** about the function **max_elem_list**.

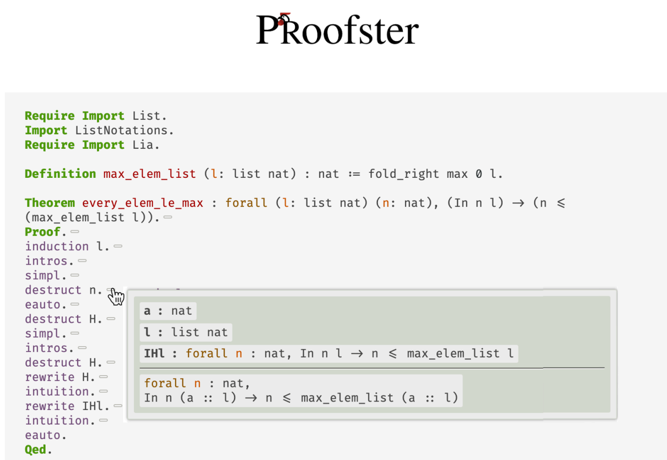

Fig. 2. When PRoofster executes the query from Figure 1, it produces a complete proof for the theorem **every_elem_le_max**. Hovering over a tactic in the proof shows the proof state at that point in the proof, which allows the developer to explore and understand how the proof verifies the property.

correctness by formally proving the property that each element of the list is less than or equal to the result of executing **max_elem_list** on that list.

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

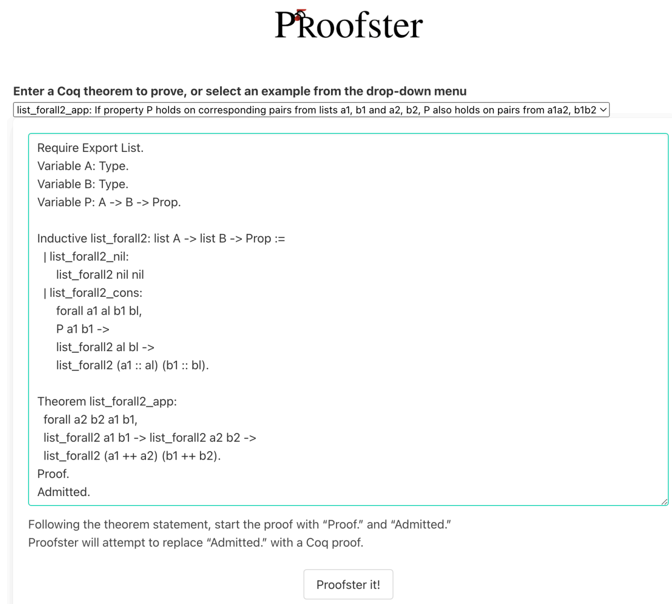

Fig. 3. A PRoofster screenshot of the developer asking to prove the theorem **list_forall2**.

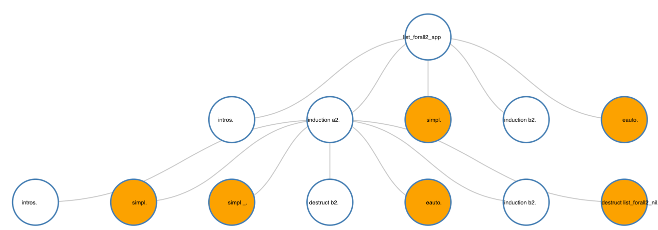

Fig. 4. When PRoofster executes the query from Figure 3, it is not able to generate a complete proof, but displays its search tree, instead. (Image has been rotated for space.)

property. Given two lists, let proposition **P** be a proposition on two elements, and let theorem **list_forall2** 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 **list_forall2_app**, which states that for all lists **a1, a2, b1, b2**, if **list_forall2** holds for **a1, b1** and for **a1, b1**, then it also holds for the pair of lists formed by appending **a1** and **a2**, and appending **b1** and **b2**.

Figure 3 shows the query the developers 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 **list_forall2 a1 b1**, 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: **induction 1**, which inducts over the first unnamed hypothesis (here, the term of type **list_forall2 a1 b1**), something PRoofster had

Proofster

```
Require Export List.
Variable A: Type.
Variable B: Type.
Variable P: A → B → Prop.

Inductive list_forall2: list A → list B → Prop :=
  | list_forall2_nil:
      list_forall2 nil nil
  | list_forall2_cons:
      forall a1 al b1 bl,
      P a1 b1 →
      list_forall2 al bl →
      list_forall2 (a1 :: al) (b1 :: bl).

Theorem list_forall2_app:
  forall a2 b2 a1 b1,
  list_forall2 a1 b1 → list_forall2 a2 b2 →
  list_forall2 (a1 ++ a2) (b1 ++ b2).
Proof.
induction 1.
simpl.
intros.
eauto.
intros.
econstructor.
eauto.
eauto.
Qed.
```

Fig. 5. The succesful result of running the query in Figure 3, modified by adding **induction 1** before **Admitted**.

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

[1] AWS Provable Security. https://aws.amazon.com/security/provable-security.

[2] L. Blaauwbroek, J. Urban, and H. Geuvers. Tactic learning and proving for the Coq proof assistant. In E. Albert and L. Kovacs, editors, *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 73, pages 138–150, 2020.

[3] M. Bostock. D3.js — Data-driven documents, 2012.

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

[5] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic — with proofs, without compromises. In *IEEE S&P*, pages 1202–1219, 2019.

[6] E. First and Y. Brun. Diversity-driven automated formal verification. In *ICSE*, 2022.

[7] E. First, Y. Brun, and A. Guha. TacTok: Semantics-aware proof synthesis. *PACMPL, OOPSLA issue*, 4:231:1–231:31, November 2020.

[8] E. J. Gallego Arias, B. Pin, and P. Jouvelot. jsCoq: Towards hybrid theorem proving interfaces. In *Workshop on User Interfaces for Theorem Provers*, pages 15–27, 2017.

[9] V. J. Hellendoorn, P. T. Devanbu, and M. A. Alipour. On the naturalness of proofs. In *ESEC/FSE NEIR*, pages 724–728, 2018.

[10] J. Heras and E. Komendantskaya. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science*, 8(1):99–116, 2014.

[11] D. Huang, P. Dhariwal, D. Song, and I. Sutskever. GamePad: A learning environment for theorem proving. In *International Conference on Learning Representations (ICLR)*, 2019.

[12] J. Jacky, S. Banerian, M. D. Ernst, C. Loncaric, S. Pernsteiner, Z. Tatlock, and E. Torlak. Automatic formal verification for EPICS. In *ICALEPCS*, 2017.

[13] K. Jacobs and B. Beurdouche. Performance improvements via formally-verified cryptography in Firefox. blog.mozilla.org/security/2020/07/06/performance-improvements-via-formally-verified-cryptography-in-firefox/, 2020.

[14] E. Komendantskaya, J. Heras, and G. Grov. Machine learning in proof general: Interfacing interfaces. In *International Workshop on User Interfaces for Theorem Provers (UITP)*, volume 118, 2012.

[15] H. Krasner. The cost of poor software quality in the US: A 2020 report. https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf, 2020.

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

[17] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *IEEE S&P*, pages 415–429, 2013.

[18] D. H. O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, Feb. 2017.

[19] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. In *NAACL-HLT*, volume 1, pages 2227–2237, 2018.

[20] C. Pit-Claudel. Untangling mechanized proofs. In *International Conference on Software Language Engineering*, pages 155–174, 2020.

[21] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends®in Programming Languages*, 5(2-3):102–281, 2019.

[22] V. Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis, UC San Diego, 2018.

[23] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *Machine Learning in Programming Languages (MAPL)*, 2020.

[24] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer. Passport: Improving automated formal verification using identifiers. *CoRR*, abs/2204.10370, 2022. https://arxiv.org/abs/2204.10370.

[25] K. Slind and M. Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, 2008.

[26] J. Souyris. Industrial use of CompCert on a safety-critical software product. projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf, 2014.

[27] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. In *ACL*, volume 1, pages 1556–1566, 2015.

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

[29] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

[30] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. In *ICML*, 2019.