Research Statment

Alex Sanchez-Stern

Software has become more and more pervasive in our world in recent decades, but building high-quality software quickly is still a major challenge. Recent advances have leveraged machine learning and large language models to help automate the development of software. These models train on large code repositories, and learn to predict new code based on what the user has already typed.

GitHub's Copilot has been a massive step forward in this area, developed by Microsoft using OpenAI's Codex model [1]. With it, developers can begin by writing a natural language description of code, or producing examples of how the code should behave on certain inputs. Copilot will then provide a suggested implementation of that code that the developer can use. Tools like these promise to bring the ability to swiftly build software to more organizations and developers.

However, a big concern with these approaches is correctness. As code is written with less and less human intervention, there is less scrutiny on invariants and general evidence for correctness of the code. Github's own Copilot website acknowledges this, saying "the code [Copilot] suggests may not always work, or even make sense. While we are working hard to make GitHub Copilot better, code suggested by GitHub Copilot should be carefully tested, reviewed, and vetted, like any other code." [1] More worrisome than obviously wrong code which "doesn't make sense" is code which appears to be correct, and even might pass some tests, but fails in unexpected ways during deployment. In fact, in security settings Copilot-generated code has been shown to have vulnerabilities 40% of the time [9].

But since analyzing code for correctness can be quite labor-intensive, how do we combine the benefits of machine-learned code synthesis, namely allowing code to be written more easily, with wide-scale code correctness and reliability?

Enter machine-learned *proof* synthesis. While machine-learned *code* synthesis can try to produce code which is likely correct, machine-learned *proof* synthesis can produce proofs which are **machine-checkable**. That means that the proofs of correctness that are produced can be independently checked by a small proof-checker kernel in well studied proof theories, such as Coq [3] or Isabelle/HOL [8]. When proof synthesis succeeds, this amounts to automatically and foundationally checking the correctness of the target code. And, unlike in machine-learned code synthesis, if the system fails to find a correct proof, the proof checker will always reject the proof, *never* producing false confidence in an incorrect result.

To bring machine-learned proof synthesis to the scale of tools like Copilot, we need three things: highly-effective proof search systems, techniques for specifying the many parts of a software system, and embeddings of common languages into a unified proof system.

Proof Search Systems Proof search systems are software systems that take a logical statement to be proven, and try to produce a proof of that statement. These logical statements can be statements about programs, such as "the sort function always returns a sorted permutation of its input", by using a mathematical model of program code. In simple mathematical domains, such as non-linear integer arithmetic, proof search can be done deterministically, with procedures that guarantee success for any true statement within the domain. In general though, proof search systems need to heuristically search an infinite space of possible proofs to find a proof which correctly proves the target theorem.

Proof search systems using machine learning have been explored by many in recent years, including teams at Google [2, 7], UMass Amherst [4], University of Innsbruck [5], OpenAI [6], and others. The current state of the art in proof synthesis, in terms of number of proof scripts correctly synthesized, is my dissertation work Proverbot9001, originally published at MAPL 2020 [12]. Proverbot9001 works by using a novel neural architecture to model proof contexts and proof scripts in the Coq proof assistant, one of the oldest and most commonly used proof languages for verifying software. The proof scripts generated control smaller, lower-level proof search procedures called "tactics" which manipulate the machine-checkable proof terms.

There's still much to do in order to bring these proof search systems to the threshold of wide applicability though, and here my collaborations have yielded many advantages. In my postdoc at UMass Amherst I've been working with Emily First and Yuriy Brun to understand and integrate tree structured models for understanding proof contexts, and proof-history sequence models for accurately predicting next steps. And

I've been collaborating with Talia Ringer at UIUC, bringing in her deep knowledge of proof semantics and sound transformations. With REPLICA [11], we conducted the first user-study of proof engineers, in order to understand what kinds of changes they make to proofs during development, and where their time is spent.

Specifying Components In contrast to proof search systems, less work has been done on easing the burden of creating *formal specifications* for the many parts of a software system. While proofs of a particular theorem or specification can be automatically checked for correctness, mistakes in the specification itself are much harder to discover. Indeed, as proof search systems get better, writing precise, modular specifications may become the hardest part of verifying large software systems.

Fortunately, there is one way to check the correctness of specifications for the majority of software components: correct component specifications are those that allow proving correctness theorems of the software taken as a whole. From this perspective, the challenge becomes taking a top-level correctness theorem for a piece of software, and decomposing it into specifications for various components. With a team in UCLA's programming languages group, I've already begun work in this area: turning hard-to-prove theorems into multiple easier-to-prove helper lemmas. Our work has not yet been published but is showing strong preliminary results.

Embedding Languages Finally, to bring the benefits of machine-learned proof synthesis to the code that makes up the majority of codebases, we need to be able to reason about code written in popular languages. Most verified software that exists today is written in specialized languages like Coq or Dafny that exist to be verifiable using particular tools. But moving all software development to these languages is not viable. Instead, we need to meet developers where they are by building tools which allow programs in popular languages, like Python, Javascript, or Rust, to be verified.

This presents unique challenges for the different features of each language, like duck-typing, reflection, and ownership types. Each of these features must be carefully modeled to avoid marking incorrect programs as correct, or visa-versa. However, this work can be done incrementally: first, by modeling a useful subset of a language, and checking the correctness of code that only uses that subset, and then gradually expanding the allowed features to maximize the amount of verifiable code. My own work has involved creating verifyable domain specific languages in the past [10]; in the future these techniques can be expanded to create verifiable models of these common languages.

Conclusion With highly-effective proof search, techniques for specifying software at scale, and embeddings of common languages into a unified proof system, I plan to make software correctness scale as fast as software implementation.

First, developers will write down an overall specification of a software system, much like requirement engineers do today. Then, they will write a natural language or example-based description of some software component they wish to implement. From there, software will attempt to automate the rest of the development process.

Tools like Github's Copilot will use the user-provided description to generate candidate implementations, and the code of these implementations will be embedded into a unified proof system. Then a specification of the target component will be derived from the specification of the overall software system, and a proof that the candidate implementations match that specification will be searched for using proof search systems. This search will reject incorrect implementations until a correct implementation is found, and presented to the user.

For code which is too difficult to automatically write, the system can safely return to the user without a solution, preventing any falsely-correct implementations.

References

[1] Github copilot - your ai pair programmer.

[2] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher-order theorem proving (extended version). *CoRR*, abs/1904.03241, 2019.

[3] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. The coq proof assistant - reference manual version 6.1. Technical report, 1997.

[4] Emily First, Yuriy Brun, and Arjun Guha. Tactok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[5] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Tactictoe: Learning to reason with hol4 tactics. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

[6] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *CoRR*, abs/1806.00608, 2018.

[7] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019.

[8] Lawrence C. Paulson. Natural deduction as higher-order resolution. *CoRR*, cs.LO/9301104, 1993.

[9] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *CoRR*, abs/2108.09293, 2021.

[10] John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, and Deian Stefan. Scooter & sidecar: A domain-specific approach to writing secure migrations. In *Programming Languages Design and Implementation*. ACM SIGPLAN, June 2021.

[11] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In *Machine Learning in Programming Languages*. ACM SIGPLAN, June 2020.