DH Sorensen

Research Statement

By way of proof assistants, the notions of *proposition* and *proof* are relevant to computer scientists and mathematicians alike. Computer scientists use proof assistants, such as Coq or Lean, to formally verify code whose security is paramount by stating and proving theorems [3, 9]. Mathematicians have used proof assistants to check the proofs of highly technical lemmas [2] and formalize both classical and modern mathematics [1, 8].

However, proposition and proof for mathematicians and computer scientists do not much resemble each other, from how proofs are engineered to the kinds of propositions formalized. Computer scientists typically formalize properties from a prose specification into a set of propositions and prove code correct with regards to that specification by proving those propositions true. Importantly, the statements of the propositions comprising the specification are informed by experience and best practices, but typically not any formal, underlying theory. In contrast, the propositions posed by mathematicians draw completely on the theory in which new propositions are couched, and the statements of propositions are derived in a methodical and highly rigorous manner.

I argue that there should be greater resemblance between proposition and proof for mathematicians and computer scientists in a formal setting. Programs should be treated as complex and mysterious mathematical objects, and formal verification as the endeavor to understand their mathematical structure sufficiently to guarantee certain desired safety or behavioral properties. From this perspective, formal verification is a mathematical endeavor, seeking to understand and systematically classify the structure and behavior of a well-defined mathematical object. It follows that it may be fruitful to introduce tools and techniques from mathematics into formal verification, in order to correctly state and verify desirable properties of software.

I have results to support this thesis. My doctoral research showed that certain upgradeability properties of smart contracts, or blockchain-based programs which routinely manage huge quantities of money, can be rigorously articulated and verified using *morphisms*, a mathematical tool which formally encodes a structural relationship between a pair of pure functions [10]. These same properties are awkward to articulate in prose because of their abstract nature, and formally verifying a prose specification of said properties relies greatly on fallible intuition to justify that one has stated and proved the theorems that accurately capture the desired properties and behaviors. A failure to correctly capture the desired behavior in the specification can be extremely costly. My research went further and showed that general upgradeability frameworks exist in analogy to *fiber bundles*, a geometric construction which is used in mathematics and phyisics to separate the interacting structure of various components of complicated mathematical objects. These results suggest that programs formalized within a proof assistant exhibit properties commonly articulated by mathematicians—in this case, of geometric objects and topological spaces.

These results may not be entirely surprising. We see mathematical abstraction in software specification, especially when that software uses mathematical concepts—for example, in the specification of a system's topology with graph theory or operations on elliptic curves in zero-knowledge proofs [5]. But anyone experienced in ITP-based formal methods will know that the actual formation of a formal specification is a translation from informal to formal specification, and thus features very little by way of mathematical abstraction [4, 6, 7]. The formal specification of a system's topology will specify process APIs or communication protocols, but will almost certainly not formalize any graph theoretic description of the system. With ever-growing libraries of formalized mathematics, there is no reason why formal specifications need undergo such informal translations from mathematical to quasi- or non-mathematical.

The goal of my research is to trailblaze the use of formalized mathematics in formal specification, making the practice of formal verification—stating propositions and supplying proofs—more effectual by adding to its mathematical maturity. This can take many forms, from actively formalizing mathematics, to building tools that treat formalized programs as well-defined mathematical objects, and introducing techniques used widely in mathematics to state and prove propositions about programs which are difficult to state correctly in prose. Because programs are vulnerable to poor specifications as much as they are to incorrect code, doing so could make formally verified software more secure by grounding the process of formal verification deeper in mathematical theory.

References

[1] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 164–172, 2017.

[2] Johan Commelin. Liquid tensor experiment. *Mitteilungen der Deutschen Mathematiker-Vereinigung*, 30(3):166–170, 2022.

[3] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graiet. A survey on formal verification for solidity smart contracts. In *2021 Australasian Computer Science Week Multiconference*, pages 1–10, 2021.

[4] M-C Gaudel. Formal specification techniques. In *Proceedings of 16th International Conference on Software Engineering*, pages 223–227. IEEE, 1994.

[5] C. A. R. Hoare. Mathematics of programming. *Byte*, pages 135–154, August 1986.

[6] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159, 2000.

[7] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying formal specification in industry. *IEEE software*, 13(3):48–56, 1996.

[8] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

[9] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[10] D Sorensen. Formal Tools for Specifying Financial Smart Contracts. URL: https://github.com/dhsorens/FinCert, doi:10.17863/CAM.118487.