Introduction. Much modern software manipulates concepts of a continuous nature, such as space, time, magnitude, and probability. Its broad purview is expanding to safety-critical domains such autonomous vehicles, which use software for planning and control. Construction of formal proofs has succeeded in ensuring the safety and correctness of traditional software, but there is additional difficulty in verifying most software which manipulates continuous concepts, because computations do not soundly implement the mathematical operations they represent. For instance, floating point operations are different from their analogs for real numbers. This mismatch has caused some catastrophic errors, such as the Patriot missile failure in 1991, due to rounding error, and the Ariana 5 rocket failure in 1996, due to numerical overflow.

Even when the mismatch between the code and the math doesn't seem to manifest in software testing, it makes it very difficult to formally guarantee the code's correctness or safety. Furthermore, for cyber-physical systems, safety or correctness properties often ought involve probabilistic uncertainty; for instance, due to sensor uncertainty, one might want to prove a system is safe with high probability, rather than absolutely. Foundational proofs of this sort face additional difficulties, both theoretical and practical; to date, no Coq library deals with probability over continuous spaces.[1]

The remedy I propose is to program with the continuous objects themselves! I have been working for the past 9 months to build a programming system call TOPOLOG, embedded within Coq, for continuous spaces[2]. Its most abstract interface is a programming language where types are topological spaces, values are points of a space, and functions are continuous maps. Since it eliminates the mismatch between the code and the math, formal reasoning about these programs is feasible. Since types are spaces, there is a "probability monad," which maps any space to the space of its probability distributions, making it possible to construct programs involving probability as well as to reason about them. The analogous construction is impossible in Coq, where types are not (necessarily) spaces.

The ability to *run* TOPOLOG programs is based on *formal topology*, a constructive theory of topology.[3] Although the development of formal topology was driven by goals of predicative constructivism, TOPOLOG is the first to realize its constructive content to produce computer programs for running continuous maps. The key idea is that a space is defined by a logical theory describing its observable properties ("open sets") and the relationships between them. Then there is a datatype of *covers*, which are (possibly infinite) collections of observable properties together with proofs that anywhere in the space, at least one of those properties holds. A point then consists of a computational procedure which takes covers an inputs and returns *some* observable property which holds of that particular point. For instance, for any tolerance $\varepsilon : \mathbb{Q}^+$, the real numbers $\mathbb{R}$ are covered by balls of radius ε with rational centers. Accordingly, a point in $\mathbb{R}$ can be approximated by a rational number to within ε.

[1] Philippe Audebaud and Christine Paulin-Mohring. "Proofs of randomized algorithms in Coq". In: *Science of Computer Programming* 74.8 (2009), pp. 568–589.

[2] The project is open-source and available at `http://github.com/bmsherman/topology`.

[3] Thierry Coquand et al. "Inductively generated formal topologies". In: *Annals of Pure and Applied Logic* 124.1 (2003), pp. 71–106.

Programming with continuous spaces is decidedly *different* from conventional programming; I have been exploring the principles of this new kind of programming.

For instance, consider an autonomous car which is approaching a yellow light and must decide whether or not to brake in order to stop at the intersection. For simplicity, suppose that it's safe to brake as long as the car is more than 10 meters from the light and safe to pass through the intersection as long as it's fewer than 20 meters away. Intuitively, we might want to write a function $\mathsf{brake?} : \mathbb{R} \to \mathtt{bool}$ which changes its decision somewhere between 10 and 20 meters away. But this is impossible, as comparing two real numbers is uncomputable. Because $\mathbb{R}$ is connected (a topological property), any program with the type $\mathbb{R} \to \mathtt{bool}$ must ignore its input. Leslie Lamport coined this *Buridan's prinicple*:[4] "A discrete decision based upon an input having a continuous range of values cannot be made within a bounded length of time."

A solution is to allow the decision to be non-deterministic. In TOPOLOG, one may write the program

$$\mathsf{brake?}(x : \mathbb{R}) : \mathcal{P}^+(\mathtt{bool}) \triangleq \mathsf{cases}(x) \begin{cases} \cdot > 10 & \Longrightarrow \quad \{\mathsf{true}\} \\ \cdot < 20 & \Longrightarrow \quad \{\mathsf{false}\}, \end{cases}$$

which features an *overlapping pattern match*, a language construct that I developed which is especially useful for continuous spaces. Note that, while the cases cover $\mathbb{R}$, each individual case is not even decidable (due to Buridan's principle). An input which satisfies multiple cases is allowed to follow any branch, so its result is a non-deterministic "merge" of those branches (if they don't already agree on their overlap); proof is required to ensure this merge exists (which is always the case for $\mathcal{P}^+(\mathtt{bool})$, the space of non-empty subsets of $\mathtt{bool}$).

Project proposal. I plan to continue implementing TOPOLOG. Once feasible, I'd like to use it program software which controls an autonomous vehicle, prove safety properties about that software, and run it on a real autonomous vehicle. Along the way, I anticipate further elaborating the computational content of formal topology and discovering more principles about how to program with continuous spaces.

Broader impacts. Safety of cyber-physical systems is paramount. Programming in TOPOLOG eliminates entire classes of bugs, such as overflow and rounding error, and its proof theory allows users to formally state and prove safety and correctness properties that may even involve probability. This will allow increased confidence in the safety of software in some of the most safety-critical applications.

Intellectual merit. My project bridges the fields of constructive topology and formal verification. For work in constructive topology, it is uniquely applied, extracting its computational content, and developing new constructions whose value is more practical than mathematical. For work in formal verification, the idea to have software actually *executing* continuous functions (rather than unsoundly modeling some operations as continuous) is largely unprecedented. It is fun to explore this unusual intersection of fields, and in my opinion worthwhile as well.

[4]Leslie Lamport. "Buridan's principle". In: *Foundations of Physics* 42.8 (2012), pp. 1056–1066.