Program verification is a powerful tool that can enhance and improve the performance and trustworthiness of all software. In a world in which we increasingly rely on software for our health, safety, and livelihoods, developing techniques to verify these programs is essential. Indeed, verification is crucial for certain programs, such as software utilized in a defibrillator. However, it is often very difficult to verify the correctness of these programs. The current state-of-the-art techniques for program verification require that one develop an entirely new framework for each programming language and each new version of the programming language. For example, if the defibrillator is written in C++11, a verifier specifically for C++11 is needed. Additionally, it is now difficult to verify all potential properties of software. Developing a program verification framework for a new language is currently a huge endeavor, taking decades of man-hours, typically Ph.D.-level. Even worse, most, if not all, of that hard and expensive work is wasted when a new programming language is targeted, sometimes even when a new version of the same language is released. It is therefore not surprising that there are very few program verifiers for real languages available, and no usable program verifiers for newer languages like JavaScript or Python. I want to change that. I want to develop a program verification framework which is language-independent. That is, a framework which takes the programming language as an input parameter and then, without any change, yields a program verifier for that language. First steps have been made in this direction, so we know it is possible.

Although some progress has been made that allow for language-independent verification, the advances to date are very limited. The K Framework (http://kframework.org) is a framework for developing executable operational semantics based on rewriting logic. Formal analysis tools can readily be built on top of the K Framework in a language-independent fashion. For example, reachability logic\(^1\), currently the best available framework for proving functional correctness of programs for our purposes, is in fact a language-independent verification framework recently developed and built on top of K. Nontrivial programs in C, Java, and JavaScript have been proved correct using a (rough) reachability logic prototype\(^2\). However, as this framework can only prove functional correctness of programs, the obvious next step is to develop a more general framework that can verify more properties of programs, such as security, liveness, and real-time properties.

With fellow Ph.D. student Brandon Moore and Professor Grigore Rosu, we distilled reachability logic to its mathematical essence, and arrived at what has become my proposed research project: program verification by coinduction, a language-independent general framework that will allow us to formally verify a wide range of properties. Like reachability logic, coinductive program verification is completely language-independent; all that is needed is an operational semantics of the language. No axiomatic semantics or verification condition generators are needed. Further, by using coinduction, we can reason about properties regarding potentially nonterminating processes. For example, this would allow us to formally verify properties of a server that is designed to run forever.

The coinductive verification framework I will develop will also have applications in compiler verification. One often wants to ensure that a program written in one language maintains the same behavior when it is compiled into a different language. This “behavioral equivalence” should preserve nonterminating behavior such as infinite loops, so standard verification techniques (based on partial correctness, that is, conditioned by program termination) may not work very well. Coinductive techniques, on the other hand, are often used...
to construct and prove properties about infinite structures. I intend to use this verification framework to generate a compiler directly from an operational semantics of a language and a program in that language. Given a fragment of code, I can use K’s powerful symbolic execution engine, as well as this verification framework, to prove the fragment is equivalent to a single rewrite rule. This compiler would be correct-by-construction, and by hooking it into a verification framework, it would have the power to greatly reduce the complexity of a program.

Moore, Rosu, and I have already begun to develop this verification framework, and the soundness of it has been mechanically proved using the Coq proof assistant. I plan on implementing this verification framework in K so any operational semantics can make use of it. Further, I plan on implementing a backend for K in Coq with this framework. Since Coq mechanically checks proofs, and fails if the proof is not valid, a proof in K with a Coq backend can be translated to a concrete proof object in Coq, which one can trust more concretely.

A Coq backend can also be very useful in conjunction with other formal analysis tools. For example, Boogie is an intermediate verification language that is used to build other program verifiers, normally through a manual translation. We can build “bridges” between the K framework and tools like Boogie to take advantage of these tools. Since K is very strong for defining operational semantics, one can define a language in K, then use a bridge to Boogie to obtain a translation automatically allowing any verification to be done using Boogie directly. With this and a suitable Coq backend, any verification done using Boogie can be mechanically checked using Coq and thus trusted. My goal is to include the building of these bridges, if time permits. If not, I intend to provide infrastructure in order to allow others to do so.

Another tool that can help unlock K’s full power is Maude. Like K, Maude is also based on rewriting logic and comes with many strong theoretical tools, including a confluence checker. A confluence checker can greatly speed up verification, by ensuring that any strategy for applying rewrite rules would yield the same result. Time permitting, I plan on implementing the bridge between Maude and K by defining and maintaining a semantics of K in Maude, thus unlocking Maude’s confluence checker, termination checker, and more for languages defined in K. The verification framework of K can then be used in conjunction with these tools to yield better performance. The Coq backend of K can verify these proofs are in fact correct.

**Intellectual Merit:** My previous research experience and my background in mathematics and computer science have drawn me to this research, and prepared me well for the challenges I face. Having had previous success in verification research, I know the challenges in developing innovative techniques for verification. I have been working closely with Professors and NSF grant recipients Grigore Rosu, José Meseguer, and Madhu Parthasarathy on my proposal. They have all given me input on goals I have, and steered me towards a set of concrete attainable goals. I plan on continuing to work closely with them. This grant, coupled with the assistance of these professors and other graduate students, will be greatly helpful in achieving these goals.

**Broader Impact:** While verification techniques are improving, they are not improving fast enough to keep up with its ever-increasing presence and the plethora of programming languages. By developing this new framework for verification and strengthening the Coq backend for K, we will be able to verify software functionality and its properties more broadly and with greater ease. Verification can become an even more powerful tool, and we can instill greater confidence in the software that is used for a myriad of reasons.
