Research Statement

William T. Hallahan

My research interests span the areas of program analysis, verification and and synthesis, and in particular I am interested in developing automated approaches that allow programmers to benefit from such techniques. Ultimately, my goal is to enable programmers who are not experts in verification and automated reasoning to still benefit from verification and automated reasoning tools. In service of this, I am interested not just in fundamental formal methods techniques, but also in the application of formal methods to specific domains. I have experience working on projects applying formal methods to automatically verify and configure networks.

The goal of *verification* is to use mathematical formalisms to prove that a program matches a specification. As opposed to verification, *program synthesis* techniques take high level specifications as inputs, and automatically generate code satisfying that specification. In my work, I leverage and develop verification and synthesis techniques to aid programmers in writing, verifying, and understanding code. Although the goals of verification and synthesis are different, there is a benefit in combining both fields: under the hood, much of my work on verification also makes use of techniques drawn from the synthesis community, and vice versa.

During my PhD, I developed techniques which aim to reduce the burden of verification for functional programmers, by both *explaining* and *automatically repairing* verification errors. I have also worked on verification and synthesis tools for network administrators. As most network administrators likely do not have extensive experience with verification techniques, this represents the particular challenge of developing techniques which require a minimal amount of specialized knowledge to use in a real world setting. In practice, this requires not only classic verification or synthesis techniques, but also exploiting domain specific aspects of networks.

1 Functional Program Analysis

Non-Strict Symbolic Execution Much of my work has focused on automated analysis of non-strict functional programs. In my early work on this topic, I developed a formalism to symbolically execute such programs, as well as a practical tool, G2, which targets the programming language Haskell [4]. To the best of my knowledge, my work was the first to apply symbolic execution to Haskell, or any non-strict programming language.

Symbolic execution is commonly used for testing. It allows generating counterexamples that violate assertions or hit errors in a program. To accomplish this, symbolic execution uses symbolic variables in place of concrete values for program input data. This allows examining how arbitrary inputs result in different end states of program execution. As branching instructions such as *if-then-else* or *case* statements are evaluated, the execution state duplicates itself to continue execution on all branches.

Non-strictness is a language property which requires evaluation be performed only as as needed, which allows, for example, non-strict languages to handle infinite data structures. My work formalizes a calculus which extends a deterministic non-strict semantics with symbolic variables, as well as other primitives useful in symbolic execution, such as a nondeterministic choice operator.

Counterfactual Symbolic Execution To evaluate G2, we wanted a large source of programs with errors, which we could then attempt to diagnose via counterexamples. We used LiquidHaskell [10] code written by students in a course taught by Ranjit Jhala at the University of California, San Diego. LiquidHaskell is a modular verifier (a class of tools which also includes, for instance, Dafny and ESCJava). Modular verifiers allow programmers to specify and prove properties of their code. Given function specifications as *preconditions* and *postconditions*, a modular verifier attempts to prove that if each functions preconditions are satisfied, its postcondition is satisfied as well. During the course, the student's code was automatically saved every time they ran LiquidHaskell, resulting in a large collection of errors messages. Thus, we planned to take this existing collection of error messages and attempt to explain them via counterexamples.

We quickly discovered that many modular verification errors cannot be explained via traditional counterexamples. To see why, consider the following simple Haskell functions:

```
add2 :: x:Int -> { y:Int | y == x + 2}        incr :: x:Int -> { y:Int | y > x }
add2 x = incr (incr x)                         incr x = x + 1
```

`incr` adds one to its input. `add2` calls `incr` twice to add 2 to its input. `add2`'s specification says that the output `y` must be exactly 2 more than the input `x`. `incr`'s specification simply requires that the output `y` must be greater than the input `x`. Even though both specifications are true, LiquidHaskell (or another modular verifier) will fail to prove the specification for `add2`. This is because modular verifiers rely on composition: when verifying `add2`, the modular verifier ignores `incr`'s definition, and only use `incr`'s specification. Thus, even though `add2`'s specification is correct, verification will *spuriously fail* because `incr`'s specification is too weak. Strengthening `incr`'s specification to:

```
incr :: x:Int -> { y:Int | y == x + 1 }
```

allows verification to succeed. However, since both specifications in the original program are true there is no traditional counterexample that will explain this failure.

This motivated us to introduce *abstract counterexamples*, and a technique to find such counterexamples called *counterfactual symbolic execution* which we implemented in G2. An abstract counterexample identifies a callee function g that has too weak a specification to verify a function f, and uses input/output pairs to show how g's specification is too weak. Returning to the `add2`/`incr` example, an abstract counterexample would be:

```
add2 0 = 3, violating add's specfication, if incr 0 = 2
```

The key point is that `incr 0` does *not* equal `2`, but (1) it is not possible to tell, just from the specification, that `incr 0` does not equal `2` and (2) *if* it did (and if `incr` otherwise had it's actual behavior) `add2`'s specification would be violated. Thus, to verify `add2`'s specification, the programmer must strengthen `incr`'s specification to rule out the abstract counterexample.

To find concrete and abstract counterexamples, our technique automatically instruments function calls with nondeterministic choices, which allow either executing the function precisely (corresponding to a potential concrete counterexample) or abstracting the result of the function as a fresh symbolic variable, constrained only by the function's postcondition (corresponding to a potential abstract counterexample.) In our evaluation, we attempted to find concrete or abstract counterexamples to 7550 LiquidHaskell errors. G2 successfully found 7379 counterexamples, accounting for 97.7% of the errors. 40.1% of found counterexamples were abstract counterexamples, and thus would not have been found by standard symbolic execution.

Automating Modular Verification While abstract counterexamples are useful as feedback to a programmer struggling with a modular verifier, they still ultimately require the programmer to debug why verification is failing, and manually find sufficient function specifications.

The process of verification works as follows. First, a programmer manually writes a specification corresponding to the code. If the modular verifier succeeds in verifying the specification, the programer is done. Otherwise, they can examine concrete or abstract counterexamples, to understand the failure. Then, they must return to the first step, and update the code or the specification based on the found counterexample.

Even with counterexamples, this can be a tedious process. My key insight was that the verification process matches a well known technique in the synthesis community: a counterexample guided inductive synthesis (CEGIS) loop. In the verification loop, the only step that is not automated is the user writing specifications. Thus, we can automate the process by using a counterexample-guided synthesizer; that is, a program which takes the counterexamples, and automatically produces specifications that block the counterexamples.

I developed a CEGIS based algorithm that enables automated modular verification of programs [3]. The process converts counterexamples into constraints, which are then passed to a synthesizer. The algorithm walks through the call graph of the program in much the same way a modular verifier does, attempting to synthesize specifications based only on those functions that are directly related. I proved the soundness and completeness of the algorithm for certain specification logics. Our practical implementation is complete for the infinite set of specifications drawn from the linear integer arithmetic logic.

In Spring 2021, my advisor received the Amazon Research Award for this work.

Functional Constraint Solving To ease solving constraints in Haskell, I developed G2Q [5]. G2Q allows users to write predicates (Boolean expressions) using Haskell functions and data types, as opposed to having to use a separate language such as SMT-LIB. Because G2Q is based on G2, a symbolic execution engine, it is able to solve problems involving recursive functions, which SMT solvers often struggle with. To evaluate G2Q in practice, I developed a variety of case studies. As an example, I implemented an interpreter for an imperative language.

In only 2 lines of code, G2Q allowed the interpreter to be run with symbolic inputs, thus obtaining a symbolic execution engine for the imperative language.

SyGuS Grammar Filtering Syntax Guided Synthesis (SyGuS) is a synthesis paradigm in which the specification is given in two pieces: a *context-free grammar* and a set of constraints. A SyGuS solver constructs a function from the grammar that satisfies the constraints. In my work on synthesizing modular specifications I initially planned to use SyGuS solvers to solve the synthesis problems, but we discovered that they were too slow for an effective approach. To improve this state of affairs, we developed the Grammar Reduction Tool [8] which acts as a preprocessor for an off-the-shelf SyGuS solver. It uses a neural network to predict, based on the constraints, which elements of the grammar are unlikely to be useful in solving a SyGuS problem. We then remove the predicted unhelpful components from the grammar before handing the problem to the solver. On our 64 problem benchmark suite, this resulted in a speedup on 32 benchmarks. The other 32 benchmarks either showed no notable change, or timed out (after one hour) both with and without the tool.

2 Network Reasoning and Repair

In addition to my work on debugging and automating modular verification of traditional programming languages, I have worked on projects applying formal methods techniques to solve problems from the networking world.

Firewall Analysis and Synthesis My first networking project involved analysis and repair of firewalls, resulting in a tool named FireMason [6] [7]. Enterprise-scale firewalls contain thousands of policies, so ensuring that the policies in the firewalls meet the specifications of their administrators is an important but challenging problem. Administrators might change a policy, but make a mistake, leading to the change not having the desired affect. We noticed that such problems were often posted on StackOverflow, where administrators would provide input/output examples of the behavior they desired. Our approach allows directly providing such input/output examples, and, from them, automatically synthesizes new rules for the existing firewall.

When repairing a firewall, a major problem is that by adding new rules we could accept or reject packets that we were not supposed to. To address this issue, we developed an approach to translate firewalls into a mathematical formalism and wrote axioms that block undesired behavior. We then rely on SMT solvers to automatically reason about the formalism. This way, we not only express that the new firewall has the desired repair, but also that the changes have not had other, adverse side effects. We demonstrated the effectiveness of FireMason on real world problems posted on StackOverflow. My advisor received the Facebook Communications & Networking Award for this work.

P4 Verification During the summer of 2017, I was an intern at Barefoot Networks, a company focused on the development and production of network switches capable of running programs written in the P4 programming language. A network consisted of a *control plane* and a *data plane*. The control plane takes a global view- it is responsible for high level routing decisions. To enact these decisions, it installs rules on the data plane, a collection of switches which follow the rules locally to rout packets. Traditionally, the switches functionality was largely baked into hardware. P4 is a domain specific language for programming on network switches, greatly increasing the flexibility of the data plane for network operators.

A more flexible data plane also means an increased risk of data plane bugs. Thus, at Barefoot Networks, I worked with Nate Foster on p4v [2], a verification tool for P4 programs. The p4v tool uses standard verification techniques, along the same lines as those used in modular verifiers. However, because we expected p4v to be primarily used by network administrators, not verification experts, we wanted to prevent spurious failures. Accomplishing this while ensuring p4v scaled required domain specific insights. P4 programs are mostly straightline code- thus, the verification condition can be computed without any abstraction at function boundaries. We also noted that P4 programs have a repetitive structure: by exploiting this fact, we could rearrange the program in a way that reduced the size of the final verification condition, while still checking all the same guarantees.

The p4v tool has been successfully applied to a large number of real world P4 programs to prove a variety of properties. Further, an attempt to use p4v to verify NetPaxos, a data plane level implementation of the Paxos consensus protocol, led to the discovery of a critical bug, which in some situations could lead to consensus not being reached.

Control Plane Synthesis Ideally in a network, a centralized controller can uniformly manage a homogeneous data plane of diverse hardware. However, this is unfortunately not realistic: hardware switches have a variety of capabilities and slightly different API implementations, which can lead to subtle bugs. In one case, such subtle issues meant that it took the ONOS development team two years to validate that their software was correctly running on a new switch. To improve this situation, I worked (again, with Nate Foster) on Avenir [1], which can automatically translate control plane instructions written against an *abstract* API into instructions compatible with a separate *target* API. Both APIs specifications are provided to the tool as P4 programs. Then, at runtime, Avenir uses a CEGIS loop to synthesize mappings from the abstract to the target API. We have proven the algorithm sound and complete and demonstrated its effectiveness with a variety of case studies.

3 Future Directions

My research aim is to allow programmers with a variety of backgrounds and skill sets to benefit from verification techniques. To accomplish this, I plan to develop techniques and tools that automate much of the tedium that is often faced when using verification tools. To ensure that my proposed work will scale to real world use cases, I am also interested in the use of machine learning to guide formal methods techniques, such as symbolic execution.

Network Programming Languages As previously described, P4 greatly increases the programmability of a network data plane, relative to a traditional network setup. Taking full advantage of this requires writing or configuring two programs: the data plane program, in the domain specific P4, and the control plane program, in a traditional programming language. There are two downsides to this approach. First, it makes it difficult to experiment with moving functionality from the control plane to the data plane, or vice versa, because doing so require refactoring the functionality into an entirely different language. Second, it increases the difficulty of reasoning about the program, because developers must consider the interaction of two distinct programs.

To address both these problems, I am interested in developing a domain specific language that allows a single program to specify the desired behavior of the network. One possible design of such a language would be a main function which accepts a packet as input, and as output provides a series of desired actions, which may, for instance, modify or forward the packet. The function would also have access to and the ability to modify stateful variables, so that the processing of past packets could influence the processing of the current packet. The language would then be automatically converted into both the controller program (in a traditional programming language) and the P4 data plane program. Optional annotations could enforce that certain variables should exist and certain decisions should be made either on the control plane or the data plane. Thus, adjusting the program to move functionality between the control and data plane would require simply changing annotations, rather than rewriting code. Since the entire network would be represented as a single program, not only could network administrators more easily reason about the overall behavior, but automated program analysis or verification tools that consider both the control and data plane behavior could be more easily developed. In contrast, existing data plane verification tools, such as my own work on p4v, often rely on the user to write unchecked assumptions about the control plane behavior.

I expect my experience with synthesis will prove useful in designing the conversion from the single network program into separate controller and data plane programs. Given the high-level nature of the language, it seems likely that many programs without explicit annotations could be naively converted into programs which send every packet to the controller for routing or that flood every packet through the whole data plane, but of course neither of these behaviors is desirable in practice. Thus, a synthesizer with smart domain specific heuristics is desirable to ensure reasonable results even without extensive annotations.

Automated Type Enhancement Modular verification is not the only technique to prove code properties. Advanced typing features, such as GADTs and type families, allow programmers to specify and prove properties of their code. For example, such features allow checking at compile time that matrix operations are applied to matrices of appropriate dimensions and that abstract syntax trees in an interpreter or compiler represent only correctly typed programs. Unfortunately, adding additional safety properties via types is not a trivial process, especially if the existing code base is large. Further, writing *new* code that passes the typechecker can be tricky- in a recent mailing list discussion, this concern was cited by Simon Peyton Jones as a reason to not use stronger types in the internals of a widely used Haskell compiler, GHC [9]. Thus, the development of techniques to automatically- or at least, semiautomatically- update programs with stronger types would aid both new and experienced programmers.

Like my work on modular verification, this research would target automatically proving properties. My work "fills in the gaps" to ensure that the proof succeeds. However, unlike modular verification, which typically relies on a conversion to first-order logic, and checking via an SMT solver, type checkers are based on (comparatively) restricted typing rules. Thus, as opposed to the CEGIS driven approach that I took with modular verifiers, I believe other techniques are likely to be more effective and efficient instrumenting programs with stronger types.

In particular, I suspect that deductive synthesis approaches based on applying rules and constraints to the syntax of an existing program are likely to be most effective.

Formal Methods via Machine Learning Through my work on the Grammar Reduction Tool and SyGuS, I have already done some exploration of how machine learning techniques could be helpful to formal methods tools. Many formal methods problem are fundamentally about searching large spaces. In synthesis, one is searching over a large space of programs for a particular program that satisifies a specification. In symbolic execution, one is searching over a large number of possible states, for a particular state that corresponds to a counterexample. Using machine learning techniques to help direct such searches is a natural direction to explore. Even with a somewhat crude application of machine learning techniques, this produced a positive result in the Grammar Reduction Tool. I'd be especially interested in collaborating with machine learning researchers to look for deeper results and insights into what techniques are likely to be most effective for particular formal methods problems, and why.

My Work

[1] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *NSDI*, pages 133–153, 2021.

[2] Calin Cascaval, Nate Foster, William Hallahan, Lee Jeongkeun, Jed Liu, Nick McKeown, Cole Schlesinger, Milad Sharif, Robert Soul, and Han Wang. p4v: Practical verification for programmable data planes. *SIGOCMM 2018*, 2018.

[3] William T Hallahan, Ranjit Jhala, and Ruzica Piskac. Counterexample-guided inference of modular specifications, Under submission.

[4] William T Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–424, 2019.

[5] William T Hallahan, Anton Xue, and Ruzica Piskac. G2q: Haskell constraint solving. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 44–57, 2019.

[6] William T Hallahan, Ennan Zhai, and Ruzica Piskac. Automated repair by example for firewalls. *Formal Methods in Computer-Aided Design FMCAD 2017*, page 220, 2017.

[7] William T Hallahan, Ennan Zhai, and Ruzica Piskac. Automated repair by example for firewalls. *Formal Methods Syst. Des.*, 56(1):127–153, 2020.

[8] Kairo Morton, William Hallahan, Elven Shum, Ruzica Piskac, and Mark Santolucito. Grammar filtering for syntax-guided synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1611–1618, 2020.

References

[9] Simon Peyton Jones. More type safety in core?, 2021.

[10] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, 2014.