Research Statement
Arjun Guha—University of Massachusetts Amherst

Overview
We are living in an era in which ordinary programmers can manipulate computing infrastructure by just writing code. Instead of asking users to carefully install and configure dependencies, an application can package its entire runtime environment in a container. Instead of buying and maintaining new physical machines, a virtual machine can be deployed in minutes in a datacenter. Even in the network, black-box networking equipment is being replaced by programmable, software-defined networks (SDN). Unfortunately, the languages and abstractions that we use to program infrastructure were designed for an earlier era. This mismatch makes our computing infrastructure brittle and insecure. Many network outages are caused by bugs in low-level network policies; many system failures are caused by updates that go awry; and many security vulnerabilities are caused by software that is mis-configured.

I believe that rigorously developed programming language technology is necessary to address the security and reliability problems that affect our computing systems. Over the past few years, the primary goal of my research has been to develop language-based abstractions and verification techniques for modern, configurable computing infrastructure. I’ve worked on new programming languages and runtime systems for software-defined networking, a development that is revolutionizing networking. I also work on automated verification and interactive program repair for system configuration languages, which have become ubiquitous with the growth of datacenters and cloud computing. The techniques I develop are general-purpose and applicable to several domains. For example, I recently adapted my work on repairing system configurations to the problem of repairing robot controller configurations. My earliest work, completed during my PhD, tackled problems in web programming and web security. I have recently returned to this area with a project to make the web a better platform for compilers and IDEs.

Network Programming Languages
Computer networks do not simply connect machines together, but run many applications on network devices, such as load balancers, intrusion detection systems, and authentication portals. Historically, these applications were black-boxes running on proprietary hardware, but software-defined networking has made it possible for many more people to program network devices using open protocols. My research in this area is guided by the question, What are the right abstractions for programming networks? Over the past few years, I’ve made several contributions in this area. I worked on a core calculus for network programming that enables equational reasoning about program transformations and program correctness [1]. I developed high-performance compilers for network programming languages [42]. I designed a platform that allows applications to safely reconfigure the network [8]. I developed a certified network control platform that provably eliminates several classes of errors [15].

System Configuration Languages
Large modern datacenters and cloud computing have made system configuration an increasingly challenging problem. In the past few years, several large-scale outages at Facebook [7], Skype [44], the New York Stock Exchange [45], and other organizations have been attributed not to software bugs, but to mis-configured software. My research in this area is guided by the question What are the right abstractions for managing system configurations? System administrators cannot manage large-scale computing infrastructure manually. They need higher-level abstractions and automated tools to help them specify and verify the state of their systems. Although existing system configuration languages provide some rudimentary error-checking, my research has shown that serious errors can still occur. I developed a static analysis technique for system configurations that eliminates a large class of errors that are hard to detect with testing [40]. I developed a semi-automatic technique to repair system configuration errors, which also provides a new abstraction for maintaining changing configurations [50].
Programming on the Web  There are several popular IDEs that allow programmers to edit and run code within web browsers by compiling to JavaScript. These systems are widely used in schools [51, 5], online courses [26, 25, 18] and even in several universities [43, 38]. A key factor driving their adoption is that they do not have to be installed and can run on locked-down machines (e.g., ChromeBooks, iPads, and lab computers). Unfortunately, most web-based IDEs are brittle (e.g., they crash when given an infinite loop) and most compilers support only language features that trivially map to JavaScript. I recently developed a language-neutral approach to address these issues [2] that is currently being integrated into two widely-used web-based IDEs.

The rest of this document discusses my recent research in detail and then concludes with my plans for future work. My work is done in collaboration with colleagues and students who are identified on my C.V.

Network Programming Languages

Networks are built with a variety of inter-connected devices that implement a variety of in-network features. Routers and switches connect machines together, firewalls and intrusion detection systems filter malicious traffic, load-balancers and caches improve performance, and so on. Traditionally, each device was a proprietary, fixed-function black-box. If a new feature were desired, expensive new hardware would have to be bought. In recent years, this model has been disrupted by software-defined networking (SDN), which replaces proprietary, fixed-function devices with reprogrammable equipment. SDN protocols are akin to an assembly language for programmable switches, so researchers have argued for high-level network programming languages since the advent of software-defined networking [19]. Like any other high-level language, a network programming language should provide abstractions that hide low-level hardware details from the programmer. Moreover, programmers should be able to reason about network behavior within the language, and should not have to resort to reasoning about low-level details.

Foundations  Many network programming languages have powerful programming abstractions, but do not specify the structure of the network [10, 47, 27, 28, 15, 31, 48, 8]. However, the operation of a network is determined both by the programs running on networked devices and the network topology. To answer any questions about whole network behavior, such as “Can host A communicate with host B?”, “Does all traffic traverse an intrusion detection system?”, or “Is there a loop in the network?”, the programmer has to step outside the abstractions of the language to reason about end-to-end network behavior.

Our work on NetKAT [1] allows programmers to ask and answer the kinds of questions listed above and, more generally, reason about equivalences between networks within a linguistic framework. NetKAT is a syntactic theory of network forwarding that supports equational reasoning, using a small set of axioms that are sound and complete with respect to its denotational semantics. Furthermore, it is straightforward to build higher-level abstractions and prove them correct using the equational theory. For example, our work shows how to do so for “slices”, which are a lightweight form of network virtualization.

The key insight of NetKAT is that end-to-end network behavior can be described using Kleene Algebra with Tests (KAT) [22] and an algebra of packet processing primitives, which NetKAT defines. The benefit of building NetKAT on this foundation is that it makes an existing body of theorems and reasoning techniques available to networking.

Compilers  To be practical, high-level languages need good compilers. A language like NetKAT needs a fast compiler because it is run in a tight loop in response to network events. For example, when a new host connects to a network, the network program has to be recompiled to route traffic for the host; when a link fails, the network program has to be recompiled to route around the failure; and when traffic patterns shift, the compiler may run again to balance the load. A slow compiler limits how quickly the network can react to changing conditions. It is natural to compile network program directly to physical forwarding rules [27]. Unfortunately, this approach is very slow and does not scale to even moderately sized network programs.
We developed a compiler for NetKAT that is two orders of magnitude faster than existing systems on a diverse set of benchmarks [42]. The key insight is a new intermediate representation based on a generalization of binary decision diagrams that can represent network programs compactly and supports fast, algebraic manipulation. This new intermediate representation also allowed us to build two powerful new abstractions for network programming. (1) We can compile programs that denote sets of end-to-end paths through the network by automatically inserting stateful operations (e.g., VLAN tagging) to distinguish overlapping paths from each other. This uses symbolic automata to represent the policy and our intermediate representation to encode their transition relation. (2) We can compile programs written for virtual networks to run on any physical network, with minimal “compatibility requirements” between the virtual and physical topologies. This abstraction models packets as values with two locations—virtual and physical—and synthesizes a program that moves packets along paths in a physical network to account for all possible actions in the virtual network.

Certified Systems Network programming languages such as NetKAT abstract away several details of networking hardware. Although this is a boon for the network programmer, it shifts the burden to the compiler and runtime system to provide a robust abstraction.

We developed a Coq-certified compiler and runtime system for NetCore (the intellectual predecessor of NetKAT) and proved that its abstract semantics is weakly bisimilar to a detailed operational model of the network [15]. In particular, the high-level abstract semantics defines a function that simply moves packets from one switch to another in big steps, but the low-level network model describes how packets are processed within network devices, including I/O buffers, idiosyncrasies of packet-parsing, and several sources of non-determinism. We extracted the controller to OCaml, benchmarked its performance, and found that it is competitive with other platforms. This work also revealed several subtle bugs in other compilers that our verified system avoids.

System Configuration Languages

System administrators manage both networking and computing resources. Just as SDN started a revolution in network management, several domain-specific languages, such as Puppet, Chef, and Ansible, have had a dramatic effect on system configuration management over the past few years. In earlier times, it may have been feasible to manage individual machines using the command-line or shell scripts. But, the scale of modern computing systems has made system configuration languages essential.

Universal Property Verification My research group has investigated universal property verification for Puppet, one of the most widely-used system configuration languages [6]. Puppet automates tasks such as application configuration, service orchestration, VM provisioning, and more. The heart of Puppet is a declarative domain-specific language that specifies a collection of resources (e.g., packages, user accounts, and files), their desired state (e.g., installed or not installed), and the dependencies between them. Configurations written in Puppet are stored on centralized servers and periodically applied to the machines under management. Unfortunately, there are several opportunities for Puppet programs to go wrong. Puppet configurations are often under-constrained (to allow them to be composed with each other) and under-specified (to allow them to be applied to a variety of machine states). However, if the user fails to specify all necessary constraints, the configuration becomes non-deterministic, which is disastrous in practice. For example, a configuration that works on a test machine may raise an error on a production machine or even silently mis-configure the production machine. It is also possible to write a configuration that is not idempotent, e.g., a configuration that makes a machine “flip flop” between two states.

I realized that the key problem with Puppet is that it is an imperative language, despite its best effort to appear declarative. Moreover, when a Puppet configuration is under-constrained, it is free to reorder the execution of actions, similar to a concurrent program. Our work on Rehearsal [40] exploits these observations to build a static determinacy analysis for Puppet. Rehearsal was the first formal investigation of Puppet
and includes a formal semantics that models configurations as programs in a small, concurrent, imperative language with filesystem operations. In this form, the subtle interactions between resources that cause determinacy bugs become clear. Rehearsal encodes the semantics of Puppet configurations as formulas for an SMT solver, applying partial order reduction and program slicing for scalability. We used Rehearsal to find determinacy bugs in several third-party Puppet configurations. Moreover, when Rehearsal finds a determinacy bug, it produces a counterexample and presents it as a patch to the configuration that eliminates the bug. Finally, once a configuration is verifiably deterministic, we can treat it as a function over machine states (instead of a relational concurrent program), which enables other kinds of analyses. For example, Rehearsal exploits determinism to prove that configurations define idempotent functions.

Interactive System Configuration Repair

Universal properties such as determinism and idempotence are necessary but not sufficient to ensure that a system configuration is correct. For example, a configuration may have a deterministic bug, e.g., it always fails to start a web server, and even a bug-free configuration needs to change to respond to changing requirements and evolving threats.

Unfortunately, languages like Puppet make changes harder to apply. A small change, such as creating a new account or tweaking a firewall rule is easy to perform with the command-line shell, which is familiar to system administrators. Another advantage of the shell is that it allows the user to explore the state of the system and typically provides immediate feedback when changes are made. In contrast, when a large Puppet configuration uses high-level abstractions and third-party code, it can be difficult to find where and how an update should be made. Moreover, once a change is made, it has to be redeployed, which can take several minutes, unlike a simple fix from the shell. Finally, a configuration update can have unintended effects, especially if it is in a module that is used in multiple contexts or deployed to multiple machines. The obvious solution—write the initial system configuration in a high-level language and then use the shell to apply small updates and fix errors—does not work. First, the system state becomes irreproducible, since the configuration no longer describes the actual state of the system (i.e., configuration drift). Second, when the configuration is reapplied, it may undo changes made from the shell.

Our work on Tortoise [50] allows system administrators to use system configuration languages and the shell in harmony. Tortoise allows a system administrator to update a system using the shell and propagates edits made from the shell back to the original system configuration. Tortoise guarantees that its configuration edits preserve changes made from the shell. Moreover, when multiple possible repairs exist, Tortoise presents them in ranked order, favoring repairs that minimize the number of changes made to the system configuration. Tortoise uses SMT-based program synthesis to search for repairs and system call tracing to record interactions with the shell (thus it supports any shell). To relate high-level Puppet constructs to low-level file system operations, Tortoise uses the Puppet semantics we developed for Rehearsal. Finally, Tortoise’s repairs are meaningful only when the configuration is a function over machine states. Although this property is not universally true, it is exactly what our work on Rehearsal guarantees.

Interactive Configuration Repair in Other Domains

Tortoise is effective because system configurations are a domain where test cases are scarce and formal specifications are difficult to write. However, several other domains share these characteristics. In a recent collaboration [20], we adapted the Tortoise approach to the problem of configuring robot control software, which tends to have configurations that need to be carefully adjusted whenever the software is deployed on a new robot or the robot is deployed in a new environment.

Instead of requiring the roboticist to manually rewrite the configuration, our system automatically generates corrections from a handful of instances of incorrect behavior that the roboticist identifies in the execution log. For example, consider a soccer-playing robot that triggers its kicking mechanism too late. It is hard for a human to determine the exact change to make to the kicking threshold, but is straightforward for a human to examine a recorded game and identify when then ball was close enough to kick. Our system takes these high-level corrections and translates them into low-level corrections of the configuration values.

However, there is a tradeoff between making large adjustments to the configuration (that may have unintended effects) and satisfying all the corrections provided by the user. Our approach is to formulate repair
as a MaxSMT problem that maximizes the number of corrections satisfied and minimizes the magnitude of each correction, with a hyperparameter determining the tradeoff between these two objectives. We have performed experiments on a physical robot (RoboCup Small Size League) and in simulations and found that this approach is fast, effective, and produces results that can outperform a robot that is hand-tuned by an expert.

Web Programming

I have recently started to tackle problems that affect compilers that target JavaScript and web-based IDEs. There are several popular IDEs that allow programmers to edit and run code within web browsers by compiling to JavaScript. These systems are widely used in schools [51, 5], online courses [26, 25, 18] and even in several universities [43, 38]. A key factor driving their adoption is that they do not require software installation and can run on ChromeBooks (which account for nearly 60% of new devices sold to schools [41]), iPads, and locked-down lab computers.

Unfortunately, browsers lack the abstractions necessary to build robust IDEs and serve as a complete compilation target for high-level languages. Most web-based IDEs are very brittle. For instance, they do not have a “stop” button and simply crash the browser tab if given an infinite loop. This is particularly problematic in instructional contexts where such errors frequently occur. Similarly, most compilers that produce JavaScript do not support features such as blocking I/O, threads, and control operators because JavaScript does not support them. The few systems that address some of these issues [46, 38, 51] rely on language-specific engineering efforts that cannot be reused by other systems. Therefore, it is natural to ask if we can significantly reduce the effort needed to compile to the Web and build robust programming environments that run on the Web. A natural way to share infrastructure, make more IDEs robust, and make more compilers complete is to develop a JavaScript-to-JavaScript compiler that augments the language with the primitives that IDEs and other languages need. However, a robust implementation of a powerful new primitive (e.g., \texttt{call/cc}) would be very costly, because JavaScript is a complex language.

In our recent work on Stopify [2], we observe that existing compilers do not generate code that relies on the full semantics of JavaScript. For example, a Scheme compiler does not produce code that requires prototype inheritance and a OCaml compiler does not produce code that relies on implicit type coercions. Therefore, Stopify defines sub-languages of JavaScript that a compiler author can identify to dramatically improve performance. Fundamentally, Stopify’s abstractions rely on an implementation of first-class continuations and it has several browser-specific optimizations that can be shared across compilers. We evaluated Stopify on 10 compilers that were virtually unchanged to show its effectiveness and generality.

In a way, my work on Stopify is a return to my older interests. During my PhD, I worked on type systems [36, 17], static analysis [14], and operational semantics [16] for JavaScript. At the time, I applied these techniques to problems in web-sandbox security [35, 13, 37] and Stopify’s abstractions could be used to build better web sandboxes too.

Impact

Several of my research papers are highly cited and have had broader impact. My earliest work on a tested, operational semantics for JavaScript led to several follow-up efforts [12, 33, 34, 3] and was leveraged by several other researchers [4, 11, 24, 39, 21, 23]. The Flow Type-Checker for JavaScript, developed at Facebook, integrates type-checking and flow analysis in a manner that is similar to my work [17]. Moreover, a similar feature appeared in TypeScript shortly after Flow was released. I am a primary contributor to the Frenetic SDN platform [9], which incorporates my research on certified NetCore [15], NetKAT [1], and high-performance compilation [42]. Several others’ systems build directly on this implementation [31, 30, 32, 29, 49] and it serves as the new backend for the Pyretic SDN controller [28]. My most recent work, Stopify [2], which makes JavaScript a better platform for compilers and IDEs, is presently being integrated into a commercial educational product (bool.com) and the Pyret IDE (pyret.org).
Future Directions

There are three broad research areas that I plan to pursue in the following years. I plan to continue working on programming language technology for system configuration and network configuration, and I am particularly interested in synthesizing the two areas. Right now, it is very hard to reason about end-to-end properties that involve both network configuration and the software stack. Toward this goal, my research group has started to investigate serverless computing (e.g., AWS Lambda). The serverless abstraction is attractive because it completely manages low-level configuration, resource allocation, and load balancing, which frees the programmer to focus entirely on writing application code. We have started developing a formal semantics that models the essence of serverless computing, including the unusual requirements that it imposes on programs that are hard to get right. Our immediate plan is to use this semantics as the foundation of a verification tool for serverless programs. We also plan to develop a serverless programming language with the goal of making serverless computing more efficient and more expressive. We also plan to investigate problems that arise when serverless functions are dynamically updated to a new version. Dynamic updates are an old problem, but we believe that the serverless abstraction, which requires all persistent state to be externalized, will be conducive to elegant solutions.

I am exciting to continue my collaboration with the robotics group at UMass Amherst. For our published work, we developed a semi-interactive procedure that helps a roboticist adjust a robot control program for a new environment [20]. We have started to apply similar techniques to transfer policies learned in simulation, using reinforcement learning, to physical robots. Another topic that we plan to pursue is to infer the hidden, physical relationships between parameters in a robot control program. Our goal is to develop a system that can can tell the roboticist, “If you adjust parameter X, you will probably need to also adjust parameter Y or Z. Moreover, run the following experiment to determine which one to adjust.”

We plan to use a combination of causal inference and dynamic program analysis to accomplish this. More broadly, robot control systems are event-driven, have hard deadlines, and involve subtle, environment-specific configuration. Therefore, the techniques that I’ve used to study web programming, networking, and system configurations can be adapted to make robot control software easier to write and more robust.

I am eager to continue working on problems in web programming and web security. A clear next step for our work on Stopify [2] is to build similar abstractions for WebAssembly. I am also interesting in the security problems that may arise WebAssembly code. Although WebAssembly is isolated from the browser, it may be possible for a maliciously-crafted input to make a WebAssembly program behave in unexpected ways. Finally, JavaScript remains significant and its reach now extends far beyond client-side web programming, e.g., it is widely-used for servers, IoT, serverless computing, and edge computing. My past work applied static analysis and type systems to address security issues in client-side JavaScript [14, 35]. I am keen to investigate the unique security problems that arise in these new platforms using language-based techniques.

References


[27] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012.


[51] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. WeScheme: The browser is your programming environment. In Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2011.