Research Statement

Mukund Raghothaman

How do we help programmers contribute to large software projects? How do we detect bugs and vulnerabilities in their code, reassure them that their contributions are of high quality, and relieve tedium by automatically generating code fragments? How do we make software development fun? My research is at the intersection of programming languages, software engineering and automated reasoning. By drawing on techniques from machine learning and formal methods to solve problems in program synthesis, verification and static analysis, my goal is to build theoretically well-understood, rigorously evaluated and practically useful tools to help programmers create better software with less effort.

My previous research has explored this agenda in three directions. First, I have developed foundational program synthesis infrastructure including the SyGuS framework [FMCAD13], back-end solvers [TR17], user-facing synthesizers [HVC14, CAV15], and code search tools [ICSE16]. Second, I have worked on combining logical and probabilistic methods in program analysis tools which automatically learn from user feedback: the result is a system with much fewer false alarms and dramatically improved effectiveness in finding bugs [PLDI18, MLP18]. Finally, I have designed new domain-specific languages and programming abstractions for stream-processing systems [ESOP16, Ths17], characterized their expressive power [LICS13, LICS14, TR18a], and designed fast resource-efficient evaluation algorithms [POPL15, PLDI17]. I will now elaborate on each of these research contributions and outline my plans for future research.

Program Synthesis: Abstractions and Applications

Programmers traditionally produce code—explicit executable descriptions of their intent. This not only involves exciting new algorithms and novel data processing pipelines, but also routinely includes tricky corner cases and mundane low-level implementation details. The promise of program synthesis is to allow the programmer to express their intent in other ways—input-output examples, test cases, logical formulas, and even natural language descriptions—and have a synthesizer automatically convert these specifications into executable code.

Syntax-guided synthesis (SyGuS). Synthesis has classically been viewed as an instance of deductive theorem proving (“for each input x, there exists an output y such that \( \varphi(x, y) \) holds”). However, its application always involves additional restrictions on the artifacts being synthesized: machine instructions in the case of code generation, specific arithmetic operations when generating invariants, certain primitives in the case of string transformations, etc. As a result, previous solvers were deeply integrated into their respective application domains, and researchers were forced to repeatedly write synthesizers from scratch. Solver development consequently required great expertise and time, and made benchmarking progress difficult.

These difficulties motivated us to formalize the syntax-guided synthesis problem: “Generate an expression \( f(x) \) from a context-free grammar G, such that for all inputs \( x \), \( \varphi(x, f(x)) \) holds” [FMCAD13, DSSE15]. SyGuS provides a uniform structure for a large class of program synthesis problems, and decouples the tasks of application design and tool development. The annual SyGuS competition (http://www.sygus.org/) has since turned into a standard candle for measuring progress in solver technology. Simultaneously, domain experts can easily integrate synthesis technology into new applications, such as the implementation of constant-time cryptographic circuits, string transformations, and our own efforts in the design of distributed protocols, such as the VI and MSI cache coherence protocols [HVC14, CAV15].
Program synthesis with “Big Code.” The emergence of large open-source code corpora such as GitHub and BitBucket has the potential to revolutionize the state-of-the-art in developer tooling. These repositories provide data to mine statistical patterns and coding idioms, and can serve as benchmarks for synthesis, repair, and verification technology. One exciting application of these repositories is as a source of API-related trivia, for e.g., “How do I match a regular expression?”. This forms an important class of questions asked by developers when they move to new programming languages, frameworks and software projects. I have contributed to a system called SWIM (“Synthesize What I Mean”) which combines coding idioms mined from GitHub with click-through data from the Bing search engine and answers API-related natural language queries with short idiomatic snippets of C# code [ICSE16].

In ongoing work, I am using similar statistical regularities in syntact to accelerate program repair in Leon, a verification and synthesis system for Scala [TR17]. All of these techniques involve casting the program synthesis problem as an instance of combinatorial search over a space of expressions: Inspired by the success of gradient descent and related numerical optimization techniques in machine learning, I am also investigating continuous program embeddings [TR18b]. By transforming the program synthesis problem into an appropriate instance of numerical optimization, we both accelerate program synthesis and can also provide best-effort solutions when a perfect solution cannot be found.

Combining Logical and Probabilistic Methods in Program Reasoning

The ideal program analysis tool—inerrant, omniscient and arbitrarily scalable—has the potential to greatly improve the experience of programming. Unfortunately, most program analysis tasks are also classically undecidable, and the scale of software projects (from a few thousand to a few million lines of code) imposes strict complexity requirements on potential analysis algorithms. The job of the analysis designer then consists principally of devising trade-offs between scalability, frequency of false positives, and the possibility of missed bugs. In several recent projects, I have designed techniques to improve analysis accuracy by incorporating user feedback [PLDI18, TR18c], through data-driven analysis design [TR18b], and by synthesizing correctness proofs using reinforcement learning and graph neural networks [NIPS18].

Automatically generalizing from user-feedback. Most existing analysis tools run in a so-called “batch mode”, where they analyse the program and output a set of alarms. These alarms share portions of their derivation trees, so that multiple bugs often have the same root cause, and multiple false alarms are caused by the tool being unable to prove some shared intermediate fact about the program. Motivated by these correlations, we developed a tool called Bingo [PLDI18], which instead engages in an interaction loop with the user: as the user triages each alarm and reports its ground truth, Bingo responds by suppressing or prioritizing the remaining alarms in the program.

The central idea is to quantify the incompleteness of each analysis rule as the probability of its generating a false conclusion about the program. The derivation trees at fixpoint naturally induce a Bayesian network where the random variables indicate the event that the respective conclusion is true. This probabilistic model then allows us to rank alarms according to their probabilities, and respond to user interaction by computing conditional probabilities. In experiments on sophisticated analyses and real-world Java programs, Bingo reduced the alarm inspection burden by approximately 62%, and by up to 98% on certain benchmarks. In ongoing work, we are extending these ideas to the setting of continuous integration, where the user repeatedly runs the analysis on different versions of the same program [TR18c]. In preliminary experiments, this system, ∆-Bingo, reduces the alarm inspection burden by an additional 50% over Bingo in its traditional interaction mode.

Synthesizing loop invariants by reinforcement learning. One of the hardest parts of reasoning about imperative programs is the construction of loop invariants. Most previous techniques either look for invariants of a specific syntactic form, or use hand-crafted features in searching for the correct invariant. On the other hand, programs can be naturally represented as graphs, such as abstract syntax trees or control flow graphs.
Inspired by the success of graph neural networks in combinatorial optimization, we have recently developed Code2Inv [NIPS18], a reinforcement learning system which operates over the AST of the program: it iteratively concretizes the syntactic structure of the invariant, and learns from feedback provided by an automatic theorem prover. On a suite of benchmark problems from the previous literature, Code2Inv learns invariants for more problem instances than the previous state-of-the-art, makes fewer queries to the expensive theorem prover, and successfully transfers knowledge from one program to the next.

**Programming Abstractions for Stream Processing**

As part of my Ph.D. thesis, I worked on the problem of processing data streams [Ths17]. Streaming data arises from a variety of sources, including event streams from sensors and medical devices, logs produced by long-running programs, feeds from financial markets, and packet sequences passing through internet routers. We were concerned with computing quantitative functions over these streams, such as counting the number of occurrences of a pattern or the mean time between occurrences of an event.

There has traditionally been limited language support to express these computations, and programmers are forced to write low-level code which manually maintains state and updates it on seeing each new input element. We proposed the model of quantitative regular expressions (QREs) to simplify this process [ESOP16]. Starting with a small set of basic function expressions (for example, “a → 3” maps sequences consisting of a single element “a” to the constant output value “3”), QREs provide a collection of hierarchically composable combinators, analogous to the operations of regular expressions, so that larger, more complicated function expressions can be constructed by combining smaller expressions. For example, “split(f, g, +)” splits the input sequence into two parts, applies f to the prefix, g to the suffix, and returns the sum of the results. Other combinators include try-else, combine, and iterate corresponding respectively to the familiar notions of union, intersection, and Kleene* from formal languages.

In addition to being modular, QREs are also agnostic of the output domain, i.e., they are parameterized by the operations permitted between values of the output type (+, −, min, and max for stream-to-integer functions, and concatenation for string-to-string transformations). We showed that functions expressible using QREs coincides with the class of regular cost functions [LICS14, ESOP16], a robust class of functions which can be equivalently represented using the operational model of cost register automata and as stream-to-term functions in monadic second-order logic, and is closed under various transformations such as input reversal and regular look-ahead [LICS13]. As a result of these connections, equivalence checking, computing pre- and post-conditions, and several other analysis problems are mechanically decidable for the case of string-to-string transformations expressible in our framework. Finally, the most important part of the QRE framework is the presence of fast evaluation algorithms: we designed an algorithm which accepts a function expression f and an input stream w, and computes f(w) in a single left-to-right pass over w, with optimal memory consumption, and which processes each element of w in O(poly(|f|)) time, independent of the length of the stream seen so far [POPL15, PLDI17].

**Future Directions: Intelligent Programming Systems**

The software developer of the future will contribute to large unfamiliar projects with great ease: in collaboration with program comprehension systems, they will be able to readily understand the structure of the codebase and explore the network of assumptions and guarantees that ensures its proper functioning; synthesis systems will automatically suggest non-trivial fragments of code based on multi-modal expressions of intent; and sophisticated analysis techniques will concurrently verify that their contributions are of high quality, free of bugs, and conform to various project-specific policies and coding idioms. I anticipate the following technical challenges in realizing this vision:

**Idiomatic, interactive, and real-time program synthesis.** Contemporary program synthesizers are typically run in a one-shot setting, accepting a set of specifications—in the form of logical formulas, test cases or input-
output examples—and outputting a piece of code which satisfies the spec. There are several problems with this approach: First, specifications are often difficult to provide and it is unclear whether they completely capture the programmer’s unspoken intent. Furthermore, since the proposed code is often cryptic and unintuitive, the programmer does not know whether it will generalize to situations beyond those in their immediate consideration. Finally, the process of programming is often iterative, with the programmer repeatedly refining their requirements on the one hand, and concretizing their implementation on the other: the programmer may not be able to articulate their intent until the code is already written down.

To be truly embraced by working programmers, synthesizers must enable simultaneous exploration of the space both of specifications and of solutions. They must be responsive, and produce intuitive code with simple explanations of their operation. To generate natural code, I will develop new synthesis algorithms based on ideas introduced in SWIM [ICSE16] and Leon [TR17], and incorporate statistical biases into synthesis procedures. Producing auditable certificates for program synthesis is closely connected to the challenge of explainable AI, and I will explore a range of solution techniques including generalized program representations [TR18b] and new human-readable descriptions of proofs and UNSAT-cores.

Automating program comprehension. Building effective program reasoning tools is a challenging endeavor: besides scalability-accuracy trade-offs, there is often uncertainty regarding environment assumptions, and programs differ in their coding idioms and quality requirements. In this context, the combination of deductive reasoning techniques with probabilistic and inductive methods has the potential to significantly advance the state-of-the-art in software engineering tools. I view Bingo [PLDI18] as the first step in such an over-arching program [MLP18].

While the immediate consequence of this research will be greatly improved analysis accuracy, I will also investigate models of alarm severity and relevance, and better ways of interacting with the programmer, such as actively soliciting their feedback on difficult sections of the program. Further out, I envision unifying testing and verification into a single framework, which automatically abduces interface guarantees, exhaustively verifies some components with respect to these properties, and subjects the remaining components to comprehensive testing to ensure conformance. Another aspect of this research program will involve developing self-tuning search algorithms such as Code2Inv [NIPS18], and data-driven verification techniques, such as those in Difflog [TR18b].

References


