Research Statement

Mukund Raghothaman

Software systems are pervasive: from mobile phones and medical devices to electricity grids and the internet, they control large parts of our modern infrastructure. At the same time, they are becoming increasingly complex and difficult to engineer. Development costs and timeframes are steadily expanding, and critical bugs and security vulnerabilities are being discovered on a regular basis. My goal is to build theoretically well-understood, rigorously evaluated, and practically useful tools that help programmers create better software with less effort. I draw on techniques from machine learning and formal methods and focus my attention on two areas: first, on building better programming abstractions and synthesis tools to simplify the process of software construction, and second, on creating better verification and program analysis systems so programmers can more effectively find bugs in existing code.

With mature constraint solving technology in the form of SAT and SMT solvers, there is an exciting opportunity to embrace program synthesis as part of the methodology of programming. I have worked on a range of problems in synthesis, from user-facing applications [HVC14, CAV15] to back-end solvers [FMCAD13, TR17], code search tools [ICSE16], and the SyGuS framework [FMCAD13, TR14].

After the code is written, automated program reasoning tools promise to detect various kinds of programming errors. Unfortunately, undecidability and scalability issues severely limit their accuracy in practice. The developers of Coverity, a commercial static analysis system, found that programmers start ignoring the tool when false alarm rates exceed 30% and that when they have a design choice between discovering more bugs or reporting fewer false alarms, they routinely choose the unsound option so as to produce fewer false positives. I have worked on using probabilistic reasoning techniques to develop Bingo, an interactive human-in-the-loop analysis system which is dramatically more effective at discovering bugs in code [TR18].

In a parallel thread of research, I have worked on designing new programming models for stream processing systems. There is a need for efficient high-level DSLs to perform computations over data streams arising from sources such as medical sensors, financial markets, and internet routers. Having drawn inspiration from regular expressions, I designed the formalism of quantitative regular expressions (QREs) [ESOP16], characterized their expressiveness [LICS13, LICS14], and designed fast single-pass 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

In contrast to classical approaches, contemporary program synthesis is distinguished by the paradigm of freely mixing input-output examples, declarative specifications, and executable code. My first exposure to the power of such multi-modal specifications was in a sequence of joint projects with Abhishek Udupa and Rajeev Alur on synthesizing distributed protocols [HVC14, CAV15]. While previous approaches were limited by the scalability of brute-force enumeration, we were able to exploit example scenarios, symmetry constraints, and protocol skeletons to synthesize non-trivial portions of self-stabilizing systems and the VI and MSI cache coherence protocols.

Syntax-guided synthesis (SyGuS). As part of this research, we had to solve the problem of synthesizing functions which satisfied a given logical specification. Despite a large body of previous work, existing tools were deeply integrated with their specific application domains, and we were therefore forced to write our own synthesizer from scratch. Not only did this take a significant amount of time, but we were also unable to
benchmark our solvers against previous work. These difficulties motivated us to formalize the SyGuS problem specification [FMCAD13, TR14], a uniform format for function synthesis problems with syntactic and semantic constraints on the desired solution. By providing a standard interface to a large class of tools, SyGuS simplifies the design of synthesis-enabled applications, allows new synthesis engines to immediately have impact across a range of application domains, and provides a standard benchmark to measure progress in solver technology. SyGuS has since achieved popularity as a standard synthesis framework, and the annual solver competition has benchmarks from a variety of application domains from string transformations to cryptographic circuits, and the participating solvers have demonstrated significant improvements in performance (http://www.sygus.org/).

Program synthesis with “Big Code.” In this context, the availability of large open-source code corpora such as GitHub and Bitbucket has the potential to revolutionize the state-of-the-art in developer tooling. During my Ph.D., I spent two summers at Microsoft Research Cambridge, where I worked with Yousef Hamadi and Yi Wei on natural language code search. Programming languages typically come with large libraries, and developers often ask questions about API-related trivia such as “How do I transmit data over a socket?”, or “How do I match a regular expression?” We developed a system called SWIM (“Synthesize What I Mean”) to answer these natural language questions with short idiomatic snippets of C# code [ICSE16]. On a benchmark suite of 30 common API-related queries, the first solution snippet produced by SWIM was relevant in 70% of the cases, and at least one of the top 10 responses was relevant for all benchmark queries. In ongoing work with Manos Koukoutos and Viktor Kuncak, I am working on exploiting similar statistical regularities in syntax to accelerate program repair in Leon, a verification and synthesis system for Scala [TR17].

Combining Logical and Probabilistic Methods in Program Reasoning

Program analysis tools traditionally rely on logical modes of deriving facts about the programs under analysis. However, their ability to accurately deduce program properties is limited both by classical reasons such as undecidability and practical considerations such as scalability. The analysis designer is therefore forced to make a trade-off between being conservative and producing large numbers of false alarms or being precise and potentially missing real bugs. In Bingo, a joint project with Sulekha Kulkarni, Kihong Heo, and Mayur Naik, we exploit interaction with a human user to significantly increase the effective precision of these analyses [TR18].

We rely on the well-known fact that the reports produced by program reasoning tools are correlated in their ground truth: multiple true alarms often share root causes, and multiple false alarms are often caused by the tool being unable to prove some shared intermediate fact about the program being analyzed.

Our main idea is to fundamentally extend the analysis with probabilistic modes of reasoning. We quantify the incompleteness of each deduction rule with a probability, which represents our belief that the rule produces invalid conclusions despite having valid hypotheses. Therefore, instead of just a set of alarm reports, Bingo also produces confidence scores measuring our belief that the alarm represents a real bug. Once the user inspects an alarm report—typically the one with the highest confidence value—Bayesian inference allows us to incorporate her ground truth label into updated confidence values for the remaining alarms.

We implemented Bingo in the context of analyses expressed in Datalog, a logic programming language which is being increasingly used to declaratively specify complex program analyses. Drawing inspiration from the literature on probabilistic databases, Bingo relies on a technique to eliminate cycles from Datalog derivation graphs and convert them into Bayesian networks. We then perform marginal inference to determine the posterior probabilities of individual alarms conditioned on user feedback. By embedding the belief computation directly into the derivation graph, our technique is able to exploit correlations between alarms at a much finer granularity than previous approaches.

We evaluated Bingo on two state-of-the-art static analyses: a datarace analysis on a suite of Java programs ranging from 95–616 KLOC, and an information-flow analysis on a suite of Android apps ranging from 40–98 KLOC. The analyses produce an average of 538 alarms per benchmark program, of which only 66 alarms are true on average. On these benchmarks, the user needs to inspect an average of just 200 reports to discover all real bugs, i.e., approximately 62% fewer alarms than with no assistance tools. On some benchmark programs,
the savings from user interaction are much more dramatic. For example, on LUIndex, a Lucene-based text search engine from the popular DaCapo benchmark suite, the datarace analysis produces 940 alarms, of which only 2 are real dataraces. Bingo presented both true alarms to the user within just 14 rounds of interaction.

**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 iter, 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 \( O(\text{poly}(|f|)) \) time, independent of the length of the stream seen so far [POPL15, PLDI17].

**Future Research: AI-Based Programming Systems**

The primary thrust of my research will be in applying ideas from statistical AI to build the next generation of programming systems. By pursuing applications in code completion, program synthesis, static analyses and type systems, I intend to help create IDEs and programming environments which cause demonstrable increases in programmer productivity. In the other direction, there is a growing realization of the social impact of algorithmic decision-making, and of the “right to explanation.”¹ I plan to explore the use of programming language technology to convert machine learning models into representations that are comprehensible to humans.

goal is to improve the accuracy of the analysis, present a more effective interaction model with the user, and gracefully handle problems such as unknown environment models and proprietary binary blobs which are opaque to analysis.

One of the major causes for false alarms is the inability of analyses to recognize software patterns and coding conventions such as double-checked locks. I will investigate the use of inductive logic programming to automatically mine these extra-analytic features from open-source code corpora, and thereby improve analysis precision.

Working on Bingo has made me appreciate the central role of the human user who deploys software engineering tools, and the importance of focussing on usability. While assigning confidence scores to alarms and updating them based on interaction forms one part of the tool’s user experience, another aspect involves propagating user feedback across routine updates, bug-fixes, and changes to the source code. I plan to extend Bingo to support the incremental evaluation of analyses expressed in declarative formalisms such as Datalog. Another important direction is exploring the disconnect between static analysis tools, which traditionally prove the absence of certain classes of bugs, and the practice of programming, which involves experimenting with poorly documented libraries and adapting example code found on the internet. I believe that static tools such as type systems and SMT solvers can be extended with first-class support for beliefs, uncertainty, and black-boxes, and potentially even synergistically combined with dynamic analyses, and thereby provide valuable assistance to working programmers during the software construction process.

Foundations of program synthesis. Counterexample-guided inductive search (CEGIS) has emerged as the dominant solution strategy in the literature—it is both easy to describe and performs well on some common classes of problems—but not much is known about its theoretical or empirical properties. For example, most implementations of CEGIS have difficulty in dealing with the simultaneous synthesis of multiple correlated functions (for example, “Synthesize functions f, g such that for all x, y, f(x, y) + g(y) = x + y and g(y) < y”) which often arise in applications in robotics and distributed systems where multiple independent components cooperate to achieve a common goal. I am interested in studying the properties of CEGIS and also plan to pursue alternative solution strategies, based, for example, on Skolem functions and variable elimination, or based on smooth interpretation and gradient descent, with the ultimate goal of building more robust and scalable synthesis engines.

Synthesis problems are also often associated with quantitative objectives. For example, when applied to program repair, we might want to minimize the edit distance from the original program or maximize the grammatical probability of the proposed solution. In problem instances drawn from cyber-physical systems, natural objectives include minimizing energy consumption or maximizing numerical accuracy. There is a diversity among these objective functions: some metrics, such as the edit distance, are properties of the syntactic structure of the solution, while other objectives, such as energy consumption, are complex semantic properties which may only be approximated by simulation. Furthermore, users might be interested in exploring the space of trade-offs between multiple competing objective functions, such as performance, accuracy, and efficiency. By formalizing quantitative extensions of SyGuS and creating high-quality solvers, I hope to make program synthesis more widely applicable in areas such as approximate computing and energy-aware programming.

References


