Research Statement
Ravi Mangal
Georgia Institute of Technology

Programs operate on data. Data produced by "real-world" processes is statistical, i.e., these processes are inherently random, so their outcome can only be described via probability distributions. Consequently, we can say that programs operate on data with statistical properties. This observation motivates my first research topic - the study of using the statistical properties of program input data to help the algorithmic search for proofs of program correctness.

A second observation is that computation involving randomness is becoming more and more common, with the power of randomness harnessed for applications such as improving algorithmic efficiency, security, privacy, and computational learning, particularly Bayesian learning. This motivates my second research topic - the study of algorithmic techniques for proving the correctness of randomized/probabilistic programs.

These research interests have been shaped by the diverse problems in program verification that I have worked on over the course of my graduate studies. My initial work focused on developing a better understanding of inter-procedural data-flow analyses for first-order imperative languages (Mangal et al., 2014) and on pushing the boundaries of counterexample-guided abstraction refinement (CEGAR) algorithms for such analyses (Zhang et al., 2014). However, challenges in efficiently applying such automated proof techniques to large software systems motivated me to adopt a more expansive perspective on program proofs. The process of formulating and trying to answer these research questions has required me to think at the intersection of programming languages theory, statistics, and computational learning theory through most of my graduate career; I eagerly look forward to continuing this journey of understanding and exploration as a postdoctoral fellow.

Program proofs using statistical properties of inputs

Though programs operate on statistical inputs, the input probability distributions are unlikely to be known. The most we can expect is to have samples from these distributions. In my past and current work, I have proposed two different ways of utilizing these samples for program proofs.

The first approach is based on the observation that program analyzers are themselves programs. The inputs to an analyzer for language $\lambda_A$ are syntactically valid $\lambda_A$ programs, and these programs are distributed per some unknown distribution. If this distribution were known, one could identify the high density regions and tailor the program analyzer to be sound yet precise and efficient in these regions at the cost of unsoundness in regions with low density. This idea is the basis of my work on "user-guided program analyses" (Mangal et al., 2015). For over-approximate analyses expressed in Datalog, we presented a systematic approach for translating these analyses into the language of Markov Logic Networks (MLNs). Informally, an MLN analysis is similar to a Datalog analysis, except that each Datalog rule is now associated with a real number, i.e., a weight, that denotes our degree of belief in the validity (precision) of the rule. These weights are learnt by observing the behavior of the Datalog analysis on a small set of sample programs. A program analyzer formulated in the language of MLNs is thus automatically adapted to make use of common program patterns. Moreover, due to the Bayesian semantics of MLNs, a user of the program analyzer can provide feedback in the form of new observations, and update the behavior of the analyzer.

The second approach for utilizing statistical inputs is an ongoing project of mine that draws inspiration from research on gradual typing (Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2006). If a program analyzer is not precise enough to prove a program property of interest, the analyzer is allowed to make assumptions such that the proof is completed, with these assumptions being checked at run-time. However, unlike gradual typing, where the programmer explicitly annotates expressions with the unknown type, we would like to impose no annotation burden on the programmer. This means that the analyzer does not a priori know the expressions on which assumptions are to be made. Since the assumptions are meant to compensate for the imprecision introduced by the analyzer, in the absence of annotations, the analyzer needs to be self-aware about where the imprecisions are introduced. The framework of monadic

---

1MLNs, proposed by Richardson and Domingos (2006), are a probabilistic generalization of first-order logic.
abstract interpreters (Sergey et al., 2013), with its well-designed interfaces, is a promising candidate for designing such self-aware analyzers, and my current work involves extending this framework. Another major challenge is computing these assumptions, particularly when we are interested in proving complex, program properties. My approach is to infer assumptions using the samples from the input distribution such that, (i) the proof is completed, (ii) the chosen assumptions are likely to be true per the statistical properties of the inputs. As part of my postdoctoral research, I intend to continue exploring the means for systematic construction of program analyzers that can exploit the statistical properties of program input data.

Verification of probabilistic programs

Existing work on verification of probabilistic programs includes a variety of expressive probabilistic program logics that require manual proofs, as well as automated verification algorithms that have been hard to scale. I am interested in the design of new logics and algorithms for algorithmically analyzing probabilistic programs in an efficient manner. In particular, I have been working on this problem in the context of verifying robustness of neural networks. I want to prove that, given some distribution over the inputs of a trained neural network, there is only a very small probability that small perturbations in the input will lead to large perturbations in the output. Since the composition of the input distribution with the neural network represents a probabilistic program, we need to apply probabilistic program verification techniques to prove this program property.

In earlier work (Mangal et al., 2019), I proposed an algorithm that combines polyhedral abstract interpretation with volume computation of convex bodies to verify probabilistic robustness of neural networks. Though correct, the algorithm is computationally expensive. In ongoing work, I am developing a new probabilistic program logic tailored to efficiently proving probable, approximate correctness (PAC) of a probabilistic program. A PAC proposition is best explained by an example - consider the proposition $\mathbb{E}_{x \sim D}[x] = c$, stating that the expected value of $x$ drawn from distribution $D$, described by a probabilistic program, is $c$. The PAC version of this proposition is of the form $\Pr_{x \sim D}(|\mathbb{E}[x] - c| \geq \delta) \leq \epsilon$, stating that the probability of the expected value of $x$ being $\delta$-close to $c$ is greater than $1 - \epsilon$. The PAC version represents a weaker guarantee, but such probabilistic assertions are amenable to being checked just by sampling the underlying distribution. Depending on the sampling budget, the parameters $\epsilon$ and $\delta$ can be adjusted. Moreover, drawing a sample is akin to a single run of the probabilistic program, so sampling can be performed cheaply. Thus, assertions in this PAC probabilistic program logic can be verified via sampling, and the proof rules of the logic inform how sampling-based proofs can be composed. While my current work focuses on proving properties of neural networks, in future work, I am very interested in considering PAC versions of other program properties while also improving the sample efficiency (i.e., the number of samples to be drawn) of such verification algorithms.
References


