Elizabeth Dinella: Research Statement

My research interests are in the fields of **Software Engineering** and **Machine Learning** with a focus on improving software correctness through the integration of symbolic program analysis and neural techniques.

Since the early days of computing, productively writing correct code has posed a significant challenge. Nevertheless, research in this area is of upmost and timely importance as Large Language Models (LLMs) have shown remarkable results in code generation tasks, but struggle to provide interpretable, faithful, trustworthy, and ultimately, correct responses. With an emphasis on recent advancements in machine learning, my research addresses fundamental roadblocks in program reasoning. Creating effective program analysis techniques is a challenging agenda, yielding decades of active research and fruitful results based on rules and formal logic. Despite their successes, these symbolic approaches have noteworthy limitations in accuracy, flexibility, scalability, and ease of use. Ultimately, my research objective is to create effective program analysis techniques through **Cooperative Program Reasoning and Neural Modeling**. Through integrating these fields, I seek to address the shortcomings of both LLMs and symbolic program reasoning approaches by establishing a synergistic relationship to capitalize on the strengths of both paradigms. My contributions in this space have been **published in top-tier software engineering and machine learning conferences receiving Spotlight and Distinguished Paper awards (ICLR and ICSE), over 280 citations, patented, and deployed in industrial systems**.

Neural Inference of Specifications

In my PhD research, I showed that inferring specifications of correctness can overcome fundamental roadblocks in program reasoning. My research contributions are a fundamental shift from the traditional symbolic program reasoning paradigm. A statistical paradigm is desirable as symbolic program analysis techniques suffer across multiple dimensions due to their rigid rule based nature. Since program analysis is fundamentally an undecidable problem, symbolic approaches must carefully balance tradeoffs to achieve scalability and high performance. Often times, such approaches leverage heuristics that can weaken performance. As an alternate approach, many analysis techniques provide an option for including a human-in-the-loop to achieve improved performance. However, requiring human interaction hinders ease of use and full automation. In general, symbolic approaches must balance a tradeoff between performance and including a human-in-the-loop.

My research asks the question: **Can we build program reasoning techniques without a human-in-the-loop while maintaining high performance?** In my PhD work, I have shown that human interaction can be reduced through **neural inference of specifications** while sustaining or even improving performance. I have created effective program reasoning techniques without a human-in-the-loop for a diverse set of problems including static bug finding and repair, program merge, and automated test generation.

In the **static bug finding and repair** domain, I advanced the state-of-the-art for JavaScript programs resulting in a first-author publication as a Spotlight paper: Hoppity [1]. Effective approaches in static bug finding and repair are challenging due to 1) a lack of program specific correctness properties and 2) complex real-world programming constructs with potentially unavailable source code (e.g., API / framework calls). Through an end-to-end neural approach, Hoppity addresses both challenges. Firstly, a correctness checking technique can only find bugs it has been specifically engineered to find. For an arbitrary program, there is no clear cut definition of what constitutes an error. Symbolic techniques typically default to a set of handwritten universal correctness rules which should be true across all programs: (e.g., `NullPointerException` should never occur). These heuristics are not ideal since developers often want to check for program specific rules of correctness: (e.g.,*"You must be logged in before posting to a blog application"*). In contrast, Hoppity learns latent correctness properties through a graph neural network trained on corpora of bug fixing code commits. Such a statistical method does not rely on universal correctness rules and is capable of finding program specific functional bugs. In regards to the second challenge, the neural method underlying Hoppity is inherently flexible to complex programming constructs and can scale to projects where the entire source is not available. In an evaluation on real-world bugs, Hoppity was able to detect and repair more faults than the leading symbolic method, TAJS.

In the **program merge** domain, my publication DeepMerge [3], contributed the first formulation of merge conflict resolution as a neural modeling problem, and resulted in a 10x improvement over the state-of-the-art. Techniques for program merging lack correctness specifications, suffering the same challenge as static bug finding techniques. The most widely used approach for program merge is the 40 year old `diff3` algorithm underlying `Git Merge`. When a conflict occurs, this technique has no notion of correctness for an arbitrary program, and requires developers to manually construct a resolution. Manual merge conflict resolution is a significant barrier to software development in teams and stalls pipelines to production. Our work aims to minimize developer labor by automatically performing a resolution when a conflict is detected. To learn the notion of a correct merge resolution, DeepMerge leverages a neural encoder-decoder framework trained over a dataset of real-world merge conflicts and resolutions. As a key innovation to improve performance, I developed a novel input representation unique to program merge. Our follow up work MergeBert [5] improves upon DeepMerge's accuracy by leveraging a pretrained LLM. My work has resulted in a patented tech transfer within Microsoft and will be released as an external tool in the coming year.

In the **automated testing** domain, my first-author work on TOGA [4] received a **distinguished paper award** at ICSE 2022 for my contributions in neural test oracle generation. Automated testing also suffers from the common program analysis challenge: a lack of program specific correctness specifications in the form of test oracles. A *test oracle* is a description of the expected output on a given input. Without effective test oracles, automated testing often gives inaccurate results in the form of both false positives and false negatives. By framing test oracle generation as a neurosymbolic problem, leveraging both coverage guided testing techniques and pretrained LLMs, TOGA is able to detect a variety of program specific bugs. On a benchmark of real-world faults, I significantly advanced the state-of-the-art in testing by finding 57 bugs using our inferred specifications in contrast to 20 bugs by the next best approach. By inferring function specific oracles of correctness, our work is capable of finding bugs that are not captured by universal heuristics.

Future Work

In my future research, I am excited about broad synergies of program analysis techniques and language models that hold immense potential for enhancing software correctness and programmer productivity. My goals can be categorized as follows: *Deep learning to assist program analysis techniques* and *Symbolic techniques to assist neural models*.

Deep learning to assist program analysis techniques: In my PhD work, I have explored this direction, but exciting challenges still remain. The fundamental challenge of balancing human intervention and performance is prevalent in nearly every program reasoning domain. Regardless of domain, neurosymbolic methods for program reasoning have room for improvement. Can we develop general purpose techniques rather than customized approaches for each reasoning domain? Can we infer interpretable specifications of correctness? My early efforts toward interpretable specifications of correctness have shown promising results [2]. In the near term, I am excited about working toward a general purpose neural specification inference library to address the challenges which program analysis techniques face. My explorations in this area indicate that such an inference approach will require training data with qualities of naturalness and readability for model understanding.

Symbolic techniques to assist neural models: Neural models, particularly LLMs, have achieved remarkable successes in programming tasks. However, they struggle to understand program semantics, are not robust to semantic preserving transformations, often hallucinate, and tend to provide incorrect yet confident responses. My future research asks, can we exploit symbolic techniques to develop approaches with interpretable outputs? Can we provide formal guarantees on a model's outputs? This nascent domain will require rigorous experiments to evaluate current approaches on fair benchmarks, observing its failures and successes. I am eager to explore this quickly developing research direction. My early efforts in this direction show promise in exploiting symbolic reasoning to address fallbacks in neural modeling for code reasoning tasks.

There is growing interest in these future research directions from both industry and federal funding programs, acknowledging the importance of improving both program reasoning and neural methods. The NSF Software and Hardware Foundations (SHF) program supports software engineering research and encourages joint synergies in areas such as machine learning. DARPA also provides funding for such research directions. In particular, the Intelligent Generation of Tools for Security (INGOTS) program supports techniques driven by program analysis and artificial intelligence. Lastly, industry programs such as Amazon's automated reasoning award provide funding for research at the intersection of software engineering and machine learning. My prior research and knowledge in these domains, as well as my experience writing grant proposals and securing funding makes me uniquely positioned to tackle the difficult challenges in this agenda.

In general, success in my research agenda will improve software development efficiency, quality, and correctness. Grounded in my prior research, I believe there are huge opportunities in integrating program reasoning techniques and large language models. Effective work in these directions will lead to more trustworthy, scalable, and accurate neurosymbolic approaches. This fusion of advanced technologies is timely, important, and will have far-reaching implications for diverse industries and domains.

Publications

E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2020.

E. Dinella, S. Lahiri, and M. Naik. Program structure aware precondition generation. *Under Review*, 2023.

E. Dinella, T. Mytkowicz, A. Svyatkovskiy, C. Bird, M. Naik, and S. Lahiri. Deepmerge: Learning to merge programs. *IEEE Transactions on Software Engineering*, 49(04):1599–1614, apr 2023.

E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2130–2141, New York, NY, USA, 2022. Association for Computing Machinery.

A. Svyatkovskiy, S. Fakhoury, N. Ghorbani, T. Mytkowicz, E. Dinella, C. Bird, J. Jang, N. Sundaresan, and S. K. Lahiri. Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 822–833, New York, NY, USA, 2022. Association for Computing Machinery.