Research Statement

Ashish Mishra

Building reliable software has been a classic goal of Computer Science. Dijkstra, in his famous lecture- *"On the reliability of programs"* makes the following four major predictions about how he foresaw what the evolution (should be) of software: (1) *programs should be correct*; (2) *debugging is inadequate and we must prove the correctness of the programs*; (3) we must *tailor programs to proof requirements*; and, (4) *programming will become more and more mathematical.* Like any well-reasoned prediction, half a century later, some of these have become true while others have not. On the one hand, the limitations of debugging and the necessity of formal verification is well understood (although they may still need to be better adopted) in principle in Computer Science. Consequently, program verification techniques have made remarkable improvements, while software debugging and testing have also gained tremendously from these improvements. On the other hand, the prediction of programming moving towards becoming a more mathematical discipline has proven to be conclusively false. Instead, modern programming languages are less mathematical and more high-level (English-like), and programming as a discipline keeps moving towards being less formal and more accessible to the masses.
These goals, predictions and their rebuttals thereafter, very interestingly summarize my current research and its future directions. The most basic premise of my research being; **Can we make programs safe and reliable using formal techniques thereby achieving the long dreamt reliability goals, while making programming as a discipline more democratic and accessible to the masses?**
My current and future research is a step towards answering this question in the affirmative. More concretely, my research is at the junction of programming languages, program verification, software engineering, and program synthesis. I apply the techniques and tools of the former two to solve problems that arise in the latter two with a goal to make **programs reliable** while also making **programming more democratic** and accessible to everyone.

To concretely achieve the above-mentioned goals, my research spans two main directions; along one direction, I use the formal method tools, specifically program analyses, to strengthen the reliability of available code, particularly in dynamic languages, against the standard notions of reliability.

1 Program *analysis and transformation* for dynamic languages

Dynamic programming languages like JavaScript, Ruby, Python, etc., have truly popularized programming. This can be attributed to their looser typing and static disciplines, which allow easy interfacing of programs together. This in turn, leads to ease of writing programs and higher developer productivity by allowing code reuse and faster development. The consequence is their immense popularity and an ever-growing ecosystem of libraries and packages. For instance, the *node package manager* (npm) for JS has more than 1.7 million packages (*modules* in npm terminology). However, this ease of programming comes at a cost; installing a npm package typically installs all its transitive dependencies with all their functionality. Generally,

only a fraction of these functionalities is needed. This causes an unwanted increase in code size, and a problem of accumulating code that in practice is never invoked. This is known as code/software "bloat". Software bloat leads to multiple issues like higher-loading times in the browsers and an increased attack surface, making these programs additionally prone to security vulnerabilities.
Eliminating this unused functionality from distributions is highly desirable. The standard approaches for software-bloat removal requires a sound reachability analysis to find unused code [1]. Unfortunately, this is infeasible in JS due to JavaScript's extreme dynamicity, making standard approaches futile. I present a fully automatic technique in a tool called **Stubbifier** [2] that identifies unused code by constructing hybrid (static+dynamic) call graphs from the application's tests and replacing code deemed unreachable with either file- or function-level stubs. These stubs allow lazily loading those functionalities which are unsoundly identified as unused by the analyses and may, in fact, be reachable. They fetch and execute the original code on-demand to preserve the application's behavior. The technique also provides an optional guarded execution mode to guard applications against injection vulnerabilities in untested code that resulted from stub expansion. Stubbifier reduced application size by 56% on average while incurring only minor performance overhead. The evaluation also shows that Stubbifier's guarded execution mode is capable of preventing several known injection vulnerabilities that are manifested in stubbed-out code.

Along the other direction, I use formal verification approaches to aid programmers in writing correct-by-construction programs with verified safety guarantees. Interestingly, I take two related but distinct approaches to achieve this; the central intuition in both is to allow the programmer to convey her intent. Once this intent is specified, formal method techniques can either verify that a given piece of code agrees to the intent (the verification approach); or, even more preferably, we can translate these intents to programs (the program synthesis approach). I discuss one of my works along each of these forks.

2 Automatic safety *verification* for effectful, higher-order programs

Automated verification of programs with higher-order functions (functions that take other functions as argument and return functions as a result) and effectful operations (operations that have side-effects like reading or writing to a file) is challenging as tracking the interaction between different functions, and the effect of those interactions on program's state is difficult to reason. Given that such higher-order effectful programs are increasingly present in critical domains like aerospace systems, medical devices and cybersecurity systems etc. an efficient verification methodology for this class of programs is imperative.
A particularly interesting instance of such programs is that of "Parsers" which are *transformers* over data, formatting unstructured data to a structured one. A very popular technique for writing these transformers is using *Parser Combinators [3]* based on the idea of combining small, reusable parsers (called "combinators") to build up more complex parsers. They make it easier to build parsers for complex data formats, as the combinators can be composed in a flexible and modular way making it easier to maintain and modify the parser as the format evolves. Consequently, this style of parser construction has been adopted in many domains, a fact exemplified by their support in many widely-used languages like Haskell, Scala, OCaml, Java, etc.

Unfortunately, these programs are sprinkled with both higher-order functions, as well as side effects and consequently automatically verifying these programs is challenging and not well-explored. To address this, I designed an OCaml DSL **Morpheus** [4] that allows compositional construction of data-dependent parsers using a rich set of primitive parsing combinators. The novelty of Morpheus is an expressive specification language and a refinement type system for describing and automatically verifying safety properties relevant to parsing applications.
Morpheus imposes modest constraints on the host language capabilities available to parser combinator programs; these constraints *enable* automated reasoning and verification, *without* comprising the ability to specify parsers with rich effectful, data-dependent safety properties. I justify this design through a detailed evaluation study over a range of complex real- world parser application.
Furthermore, Morpheus is perfectly positioned at a sweet spot in terms of the expressiveness and tractability of its verification. This makes it useful to write (and verify) programs from several other domains besides parsers. One such domain is of *networking* programs where researchers are independently building similar refinement types for languages like P4 [5]. Another good example is the domain of verified database applications which combine high-level language features which modify underlying database states through *transactions*. In future, I plan to extend Morpheus to these domains. Thus, Morphues conclusively takes us a step closer to the long-term goal of writing safer effectful programs.

User intent, which allows a verification toolchain in Morpheus can also be used to solve the more challenging dual problem, i.e. automatically synthesizing a program satisfying a programmer's intent. This will not only make writing programs safer by generating correct-by-construction programs but will also make programming more accessible, thus contributing towards both the major philosophical goals of my research.

3 Specification-guided *synthesis* for effectful programs

component-based program synthesis is a technique for intelligently composing a set of given high-level components or libraries to generate programs for a given user specification automatically. A key advantage of Component-based synthesis is that it allows developers to focus on the high-level design of the program rather than the low-level implementation details. This can make it easier to write and maintain complex programs. One central challenge in component-bases synthesis (and program synthesis in general) is how to search for a correct program in a combinatorial space of possible programs which can be constructed using the libraries. The problem is further exacerbated when libraries have side effects like mutable states as the search space becomes a cross-product of programs and heap states. An attempt to tackle this challenge has led to substantial research efforts towards component-based synthesis [6,7]. Unfortunately, all these efforts are either agnostic about effectful semantics of libraries [6] or have a very coarse-grained notion of effects [7]. Consequently, they either fail to scale to effectful/stateful settings and/or synthesize unsound programs when applied to component libraries with side effects.

In **Cobalt** [8] I designed a specification-guided component-based synthesis technique for effectful libraries. The key insight is that *library specifications can aid in efficient program*

search. The synthesis procedure uses Hoare-style pre- and post-conditions to express fine-grained effects of potential library component candidates and to drive a bi-directional synthesis search strategy. To further improve efficiency and scalability, Cobalt integrates a conflict-driven learning procedure into the synthesis algorithm that provides a semantic characterization of previously encountered unsuccessful search paths used to prune possible candidates' space as synthesis proceeds. I also empirically show that the synthesis approach is generic and can be applied to libraries from different domains like imperative data-structure libraries, database transactions, parsers, etc.
Cobalt provides a generic strategy for component-based synthesis over libraries manipulating states; one interesting direction for its extension is to the context of distributed systems and their safety properties. For instance, one can define the semantics of distributed database transactions in a rely-guarantee like specifications [9] and couple these with global database invariants to synthesize correct distributed database programs in a similar fashion as we do this for their non-distributed counterparts in Cobalt's evaluation.

Principled proof-search-guided component-based synthesis. In **Cobalt**, I extend the state-of-the-art for effectful program synthesis which henceforth lacked efficient enumeration/search in the space of valid program terms and thus required very precise logical specifications. The CDCL search procedure in Cobalt utilizes a specific property of program terms *viz.*, the equivalence of the programs modulo the failing terms for a given synthesis goal. Although it greatly improves the enumeration, internally, we are required to perform multiple heuristic-based optimizations in the search/enumeration procedure implicitly to dissuade it from exploring equivalent paths multiple times.
This is true for most of the program synthesis approaches which have similar heuristic-based enumeration optimizations. Unfortunately, I show that these heuristics are both ineffective and lead to incompleteness in the synthesis procedure. In one of my ongoing works [10], I solve this problem in a more principled way. The key idea is to borrow from the developments in two related fields: *proof-theory* which tackles efficient proof-search and *inductive-synthesis* which tackles efficient program enumeration in combinatorial search space.
I introduce a novel data structure called *Hyper Proof Tree Automaton*, which reifies, in a compact way, the large space of proof terms generated during the synthesis process and equivalences between these terms. Consequently, the optimizations on the proof-search process are introduced as operations on this data structure. This, coupled with a proof (equivalently a program) extraction algorithm, allows a principled approach for proof-strategy-guided synthesis procedure.

4 Future research directions

The works discussed above make a significant contribution towards the goals of **reliability** *and* **ease of programming** *for modern software systems. However, with ever-evolving domains of software, like the modern machine learning and artificial intelligence (ML/AI) systems and modern control system software as in Robotics, these works have only scratched the surface. A long-term vision for my research is to build on these initial developments to tackle the challenges these evolutions raise while utilizing the opportunities and newer avenues they create. A few concrete examples of some of these open challenges are; Program synthesis is still in its nascent state and requires* **fundamental improvements** *to make it applicable to real-world*

problems. Further, the techniques (both verification and synthesis) I have designed can also be applied to answer important reliability questions in (AI/ML) software; at the same time, the developments in (AI/ML) world can help **improve both program verification and synthesis**. *Finally, program syntheses can have a significant impact when applied to* **novel domains** *like robotics and cyber-physical systems. Following, I list some of these imminent future directions.*

Type-guided synthesis for learning abstractions/libraries: The user intent (the logical specifications in Cobalt and Morpheus) is a part of the input to a program synthesizer; typically, a synthesis procedure also requires either a program-sketch, a domain-specific language or, as in component-based synthesis, a library of components. This is because the enumeration in program synthesis approaches faces a significant challenge: How to explore the humongous program space efficiently? A standard approach is to restrict this space; one way this space is restricted is by providing a suitable *Structure Hypothesis*, which defines some structure of the allowed programs. This hypothesis is commonly provided as a *program sketch*, a Domain Specific Language(DSL) or a set of libraries (as in Cobalt and other component-based syntheses). Unfortunately, providing these hypotheses is non-trivial for the programmer, and this hinders a broader adoption of these program synthesis approaches.

This leads to an interesting research question; *Can we learn these Structure Hypothesis automatically given a corpus of programs?* This is commonly called *Learning Abstractions* or *Learning Libraries*. Interestingly, the techniques and solutions used in Program Synthesis can also help learn these structure hypotheses given a corpus of code. The core challenge in these problems is to find common abstraction patterns in the large corpus in an efficient way. Fortunately, just like we use specifications (e.g. types) to guide the synthesis process by picking programs of correct shape and discarding incorrect ones in Cobalt, these specifications can also guide the picking of similar shapes across the corpus.
Further, types (both *basic* and richer types like *Refinement Types*) can filter many programs with overlapping structures but inconsistent types, thus more efficiently guiding the matching process. I plan to apply these type-guided search techniques to the problem of library/abstraction learning in one of my future works.

Neurosymbolic deductive program synthesis: In each of the works I hitherto discussed, I use logical formulae to capture the programmer's intent which is then deductively refined to programs. This flavor of program synthesis is called *deductive program synthesis*. In practice, the user intent can also be captured in other less formal and less precise ways, e.g. input/output examples. This flavor of program synthesis requires inductive generalisation from these input/output examples and is thus called *inductive synthesis*.
Since *program induction* is considered one of the fundamental problems in ML/AI, recent years have seen a growth in many neural architectures (and other ML approaches) applied to the program synthesis problem, particulaly *inductive* program synthesis.
For instance, there are approaches [11] that train a machine learning model (typically a deep neural network) to directly predict a complete program from the given specifications (typically i/o examples). However, these purely neural (or ML) based systems have some key limitations: for instance, often program synthesis tasks are hard for deep networks which have no symbolic component. Further, it is challenging to provide any guarantees for the synthesized programs due to the black-box nature of these neural networks.

These observations have led to recent attempts at combining the symbolic program synthesis with machine learning-based approaches leading to the development of the new popular domain of *Neurosymbolic Program Synthesis* or *Neurosymbolic Programming*. The idea is that together these approaches will be able to tackle challenges which are infeasible for either alone. For instance, Deepcoder [12] uses deep learning to guide an enumerative search in inductive-synthesis by training a model to predict the probability of each component in the source code. Similarly, a recent paper [13] uses the RL system to guide the search process based on the reward function defined using the failing terms.
Combining neural techniques with deductive program synthesis is yet an open and challenging problem; particularly challenging is how to use machine learning approaches to guide the synthesis of programs with side effects, as in Cobalt. Given that such synthesis problems are equivalent to the proof-searches, I believe advancements in the applications of ML techniques in the proof-search community can aid in efficient exploration in the search space in such synthesis problems. In future, I plan to work closely with ML researchers to tackle such problems.

Program synthesis in novel domains like *Robotics*: Hitherto, I have discussed general programming domains for program synthesis. Unfortunately, solving the general program synthesis problem is hard, consequently, many program synthesis approaches focus on a much smaller domain of programs. For instance, FlashFill and related works [14] focus on spreadsheet programs where they use domain knowledge to improve the effectiveness of the synthesis procedure. I am interested in applying program synthesis to many new unexplored domains.
In certain domains, however, like Robotics, programming traditionally involves both designing the robot structure (the parts of the robot) as well as designing the best control policy (sequence of actions) for the design and a given user objective. Consequently, the space of designs (the structure + the robot control policy) is intractably large. Automating correct robot design given real-world objectives is thus both a challenging and important research problem.
Currently, there are limited tools for automatically exploring this search space. One common approach is graph-heuristic based, using Graph Neural Networks [15], which accelerates the exploration by prioritizing certain designs over others. However, these approaches have no way of guiding the design synthesis against given real-world logical specifications.
Combining these neural-guided approaches with logical specification-guided deductive synthesis (like in Cobalt) can be beneficial in tackling these limitations. *Again using the user intent*, one possible approach is to extend Cobalt to accept temporal logic formulae (like LTL/CTL formulae) as robot specifications (along with the grammar for its structure) and use these specifications to learn a set of allowed trajectories for the robot given user specifications using the advances in robot path planning. Following this, we can use these trajectories to aid the synthesis process by defining the synthesis as a reinforcement learning problem penalizing those designs which are less likely to follow these trajectories while boosting others.
I plan to work closely with researchers in the robotics domain in the future to apply these ideas in program synthesis to tackle some of these problems.

References

[1] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for java," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, p. 625–666, nov 2002. [Online]. Available: https://doi.org/10.1145/586088.586090

[2] A. Turcotte, E. Arteca, **A. Mishra**, S. Alimadadi, and F. Tip, "Stubbifier: Debloating dynamic server-side javascript applications," *Empirical Softw. Engg.*, vol. 27, no. 7, sep 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10195-6

[3] G. Hutton and E. Meijer, "Monadic parser combinators," 1996.

[4] **A. Mishra** and S. Jagannathan, "Morpheus: Automated safety verification of data-dependent parser combinator program," in *Under Submission in ECOOP*, 2023.

[5] M. Eichholz, E. H. Campbell, M. Krebs, N. Foster, and M. Mezini, "Dependently-typed data plane programming," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498701

[6] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 599–612. [Online]. Available: https://doi.org/10.1145/3009837.3009851

[7] S. N. Guria, J. S. Foster, and D. Van Horn, "Rbsyn: Type- and effect-guided program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 344–358. [Online]. Available: https://doi.org/10.1145/3453483.3454048

[8] **A. Mishra** and S. Jagannathan, "Specification-guided component-based synthesis from effectful libraries," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563310

[9] G. Kaki, K. Nagar, M. Najafzadeh, and S. Jagannathan, "Alone together: Compositional reasoning and inference for weak isolation," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, dec 2017. [Online]. Available: https://doi.org/10.1145/3158115

[10] **A. Mishra** and S. Jagannathan, "Prudent: A principled proof-serach-guided component-based synthesis," in *Under Preparation*, 2023.

[11] A. Neelakantan, Q. V. Le, M. Abadi, A. McCallum, and D. Amodei, "Learning a natural language interface with neural programmer," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=ry2YOrcge

[12] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *Proceedings of ICLR'17*, March 2017. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/

[13] Y. Chen, C. Wang, O. Bastani, I. Dillig, and Y. Feng, "Program synthesis using deduction-guided reinforcement learning," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 587–610. [Online]. Available: https://doi.org/10.1007/978-3-030-53291-8_30

[14] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 317–330. [Online]. Available: https://doi.org/10.1145/1926385.1926423

[15] A. Zhao, J. Xu, M. Konaković-Luković, J. Hughes, A. Spielberg, D. Rus, and W. Matusik, "Robogrammar: Graph grammar for terrain-optimized robot design," *ACM Trans. Graph.*, vol. 39, no. 6, nov 2020. [Online]. Available: https://doi.org/10.1145/3414685.3417831