The last four decades have seen computer scientists struggle with the *software crisis* [6]: an ever increasing reliance of our society on computing resulting in the growing power and complexity of these systems. Moreover, this problem is further exacerbated by the difficulty of writing correct and useful programs economically. In fact, the Consortium for IT Software Quality estimated that in 2020, the cost of poor software quality due to bugs and other defects during software development, deployment, and maintenance amounted to $2.04 trillion in the US alone[1]. Thus, research and development of tools that help programmers write software *quickly* and *correctly* is of paramount importance.

I work on program synthesis to address this problem of *automatically* generating *correct* programs from their specifications, making the task of programming easier. Program synthesis has seen significant success in program optimization, compilation to different architectures, and data cleaning tasks in spreadsheets. However, its successful application to general software development tasks remains limited, primarily because of the sheer complexity of the languages, libraries, data security, and external systems like networks and databases. My work addresses this problem by introducing new techniques to synthesize programs in production-grade languages which form components of a larger software application. It automates writing the tedious and error-prone parts of the code, enabling the programmer to focus on higher-level application logic. For example, I have built tools to synthesize database accessor methods for web apps, and to synthesize security wrappers for queries over sensitive user data to maintain privacy. My tools fit in a developer's existing workflow while giving them the control to use the synthesized code at intended places as needed in the larger application.

Synthesis is only as good as the specifications it depends upon, but programmers are often reluctant to use sophisticated logics and formal specification languages. A key feature of my research is that the techniques I have developed automatically generate code using specifications *programmers already write*: test cases and lightweight formal program properties like types. These two different kinds of specifications have strengths that complement each other. The lightweight formal properties define over-approximate semantics of programs, which I have shown how to use to guide the search for candidate programs efficiently. Notably, my work shows that the program search can be guided if any property is specified as an *abstract interpretation*, a topic that has been studied for decades. I also push the state-of-the-art on guiding program search from tests by using tests as more than just a correctness check. I show that tests can be used to synthesize branches in programs, and they can even guide the program search by inferring side effects information from test failures. These contributions build the foundation to deliver the benefits of program synthesis to end-user programmers. My work to date marks the first step towards my long-term vision of *practical synthesis tools* for programmers. Moving forward, I would like to build synthesis tools that support domain experts with formal properties, such as probabilistic properties like fairness; and general approaches to scale program search using test suites—the primary correctness check used by programmers in practice. These together lay the groundwork to realizing the potential of developing *correct* software *quickly* by removing the manual laborious parts of programming.

1 Synthesis of Effectful Programs

A key task in modern software development is writing code that composes calls to existing APIs, such as from a library or framework. Component-based synthesis aims to carry out this task automatically, and researchers have shown how to perform component-based synthesis using types or special properties of the synthesis domain, which is critical to achieving good performance. However, prior work does not explicitly consider side effects, which are pervasive in many domains. For example, consider web apps, a centerpiece of daily online interactions—from searching information to handling billions of dollars in payments. Under the hood, such apps act as the liaison between the database, network, and global state of the program itself. However, these apps are not amenable to formal specifications, often relying on type systems and testing to enforce correctness, making it hard to use existing synthesis methods.

We address this issue by introducing RbSyn [9], a new tool for synthesizing Ruby methods. In RbSyn, the user specifies the desired method by its type signature and a series of test cases it must pass. RbSyn then searches for a solution by enumerating candidates and checking them against the tests. The key novelty of RbSyn is that the search is both *type- and effect-guided*. Specifically, the search begins with a *typed hole*, *i.e.,* a placeholder program, tagged with the

[1] https://www.it-cisq.org/press-releases/cost-of-poor-software-quality-in-the-us/

method's return type. Each step either replaces a typed hole with an expression of that type, possibly introducing more typed holes; inserts an effect hole, annotated with a write effect that may be needed to satisfy a test assertion; or replaces an effect hole with an expression with the given write effect, possibly inserting another effect hole. Once this process finds a set of method bodies that cumulatively pass all tests, RbSyn uses a novel merging strategy to construct a complete solution: it creates a method whose body branches among the conditions, executing the corresponding (passing) code, thus yielding a single method that passes all tests.

I observed, in practice, programmers test an effectful method by triggering a complementary side-effect in their test. For example, to check if a method *writes* to a database correctly, a corresponding test will *read* from that database location to assert expected behavior. This led to the key insight, that test executions can be monitored for errors to automatically infer the effect holes with desired effect labels. I formalized this search using types and effects, and implemented a practical tool for synthesizing programs in Ruby. It builds on RDL, a type system for Ruby, to which I have previously contributed [3, 4]. I evaluated RbSyn by synthesizing database model methods from production-grade Ruby applications like GitLab, Discourse, Diaspora, etc. using their original tests. My evaluation showed effect guidance is useful for outperformance when compared to just type-guided synthesis. Additionally, RbSyn synthesizes `if-then-else` branches using the unit tests, often used to encode data validation checks and business logic in web apps. The branches synthesized by RbSyn are at parity with code written by software engineers of those projects. Moreover, RbSyn's lightweight effects allows programmers to explore a spectrum between highly precise to very coarse grained effects–allowing flexibility to tune the synthesis performance based on specification burden vs. synthesis time budget, while still being correct because of testing.

RbSyn was published at PLDI 2021 and has received some attention from the wider community. The open-source project on GitHub[2] has received 90+ stars. It has also been featured on the Ruby Weekly Newsletter #599[3] and the Ruby Rogues Podcast[4].

2 Correct-by-construction Privacy Preserving Queries

A common task in security is to ensure that private or sensitive data cannot be accessed by unauthorized third parties, a property formalized as *non-interference*. However, in practice, non-interference is too strong to be enforced in programs. Real-world programs, however, often need to reveal information about sensitive data. For example, a password check routine needs to display the result of checking if the user-input password is correct or not. To support such use cases, programmers use *declassification* statements that can be used to weaken non-interference by allowing selective disclosure. Declassification statements, however, are typically part of an application's trusted computing base and developers are responsible for properly declassifying information. In particular, mistakes in declassification statements can easily compromise a system's security because declassified information bypasses non-interference checks.

To address this problem, I designed Anosy [10], a framework for enforcing declassification policies that regulate what information can be declassified by limiting the amount of information an attacker could learn from the declassification statements. Specifically, declassification policies are expressed as constraints over knowledge, which semantically characterizes the set of secrets an attacker considers possible given the prior declassification statements. To enforce such policies, first I develop a novel encoding of knowledge approximations using Liquid Haskell's refinement types to produce machine-checked proofs of correctness. Second, the constraints generated by these refinement types are combined with numerical optimization in Anosy to automatically synthesize functions to compute correct-by-construction knowledge approximations for queries on secret data. This is the key insight behind Anosy: a function can be generated that given any prior knowledge of an attacker, computes the posterior knowledge if the query on secrets were to be executed. As a result, such a function can be directly integrated in the application, eliminating the need to run an expensive static analysis (such as Prob [5]) for each query computation.

I evaluated Anosy using domains such as intervals or powersets of intervals on diverse set of queries from past work inspired by targeted advertising on Facebook. Anosy's synthesis being a compile time operation has a one time upfront cost, but the subsequent estimation of adversary's knowledge incurs no cost. In contrast, tools like Prob need to run an expensive static analysis each time adversary knowledge is to be computed. Moreover, Anosy's synthesis algorithm is more precise for queries containing disjunctions than Prob for higher precision domains like powersets. Anosy ships with a monad, `AnosyT` that allows safe declassifications to guarantee end-to-end policy compliance in

[2] https://github.com/ngsankha/rbsyn
[3] https://rubyweekly.com/issues/599
[4] https://go.umd.edu/rbsyn-podcast

applications. I show that such declassifications support sequence of multiple queries to be answered securely in an end-to-end application, while the precision and time taken for synthesis can be tuned using the cardinality of powersets. Anosy was published at PLDI 2022.

3 Synthesizing Programs from Tests and Abstract Semantics

My observation from these projects was, we built synthesis tool for each domain that has a specialized search procedure combined with the right abstraction. This, however, meant developing a complete and accurate embedding of the source language in the logic of the underlying the synthesis tool. Often the key insight is in designing the right abstraction for the problem domain, like types and effects for Ruby programs or refinement types for Haskell programs.

To ease this process, I develop Absynthe [8], an alternative approach based on user-defined abstract semantics that aims to be both lightweight and language agnostic. The synthesis engine is parameterized by the abstract semantics and independent of the source language. In Absynthe, users define a synthesis problem via concrete test cases and an abstract specification in some user-defined abstract domain. These abstract domains, and the semantics of the target language in terms of the abstract domains, are written by the user in a domain-specific language. Moreover, the user can define multiple simple domains, each defining a partial semantics of the language, which they can combine together as a product domain automatically. Absynthe uses these abstract specifications to automatically guide the search for the program using the abstract semantics. The abstract semantics are lightweight to design, simplifying away inconsequential language details, yet effective in guiding the search for programs. Moreover, this taps into the collective knowledge the programming language community has developed over past decades in using abstract interpretation for designing static analysis tools. The key novelty of Absynthe is that it separates the search procedure from the definition of abstract domains, allowing the search to be guided by any user-defined domain that fits the synthesis task. More specifically, the program search in Absynthe begins with a hole tagged with an abstract value representing the method's expected return value. At each step, Absynthe substitutes this hole with expressions, potentially containing more holes, until it builds a concrete expression without any holes. Each concrete expression generated is finally tested in the reference interpreter to check if it passes all test cases. A program that passes all test cases is considered the solution.

For a baseline comparison against other tools, I evaluated Absynthe on the SyGuS strings benchmark—a standard program synthesis benchmark in a small functional language. Unlike tools like CVC4, that have complete background theory for strings and linear integer arithmetic, Absynthe works with more lightweight semantics like string length, string prefix, and string suffix domains. Moreover, Absynthe's guidance using the abstract semantics allows it to outperform other enumerative synthesis tools designed for SyGuS programs [1]. Moreover, Absynthe allows the programmer to mix-and-match abstract domains, while the synthesis time adapts based on the expressiveness of the domains used for specifications. To evaluate the generalizability of Absynthe to different synthesis problems, I used it to synthesize the AutoPandas benchmark [2]—a suite of Python data frame manipulation programs using the Pandas library sourced from StackOverflow questions. AutoPandas used a graph neural network models trained on dedicated hardware over a 48 hour period to solve 17 out of 26 benchmarks. In contrast, Absynthe solves the same number of benchmarks (with some overlap in the set of benchmarks solved) using just a type system and sets of data frame column labels on a 2016 Macbook Pro.

4 Future Research Directions

My long-term research vision is to make program synthesis tools accessible enough such that parts of larger software are automatically generated with correctness guarantees. I will address three increasingly broad audiences to make this a reality: practicing programmers, domain experts, and potential users in other domains.

Synthesis for practicing programmers. Building synthesis tools for software engineers means adapting the specifications to the kind engineers already write, which often are tests. However, there is an impedance mismatch between the kind of formal specifications synthesis tools traditionally accept and test cases, as it is not economically feasible to formally model the plethora of components or systems used by modern applications. RbSyn's inference of side-effects from failures, and the ability to merge programs to create branches are early promising results that search can be guided using tests. The key research question is: can a tool infer general reasoning for why a test failed, and using that to guide

the search? I believe recent advances in specification mining and random testing can be used to build under-approximate language semantics. These in turn can be used to construct proofs using incorrectness logic [7] that might help guide the search for program synthesis. The impact of generalizing search from tests are immense, as this opens the applicability of synthesis to generate correct implementations from integration test suites in larger software programs.

Synthesis for domain experts. Program synthesis needs new abstractions to express formal guarantees about programs to aid domain experts write programs with desired guarantees. For example, data scientists care about probabilistic properties (like fairness), or security researchers care about protecting secrets while still running computation that use those secrets. All these properties correspond to a formal property, that can be verified against semantics of a given program. I would like to design new abstractions for domain experts specifically to aid synthesis, *i.e.*, lightweight specifications combined with supporting for automatic program analysis. My previous work on Absynthe forms a vehicle for delivering such tools for domain experts. Along the way, I will make foundational contributions to improve scalability of abstract-interpretation guided synthesis—better abstraction inference methods and machine learning-based approaches to bias the search towards *likely and sound* candidates.

Synthesis for all. Increasing the scope of automation for mundane tasks has the promise to utilize human labor for more productive work. While programmers are well equipped to design automations, it still remains a highly specialized skill. I will continue to drive the improvement of program synthesis technologies through better specification input modes and encapsulation of programming language details from non-expert users to realize this potential.

Nearer-term goals. As a stepping stone towards my larger vision, I want to focus on a few key projects next:

- *Real-time synthesis support for code editors.* Technology developed in prior research still have not made it a programmers workflow. Getting there requires work on technical challenges such as handling partial programs, handling invalid programs as the user is typing, and making program search incremental to make synthesis quick. The success of such editor support for end-user programming needs to be validated through user studies.
- *Resource-safety through synthesis.* Active research on capabilities and coeffects can be adapted to a synthesis context to support polymorphic effect specifications, permissions inference from a program context to allow more expressive specifications beyond static database or memory labels explored in RbSyn. The goal will be to synthesize common programs with an expressive capabilities system that enforces resource safety, such as file handles or network sockets should be closed after I/O.
- *Program logic for under-approximate reasoning.* As a milestone towards generalizing search guidance from tests, I will develop theoretical foundations for proving program realizability. The principles of under-approximation can be structured in Hoare logic style to prove a program exists in a grammar provided the tests check concrete program executions. It will form the basis for future work on designing practical tools that are guided by tests and under-approximate semantics.

References

[1] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*. Vol. 10205. Lecture Notes in Computer Science. 2017. DOI: 10.1007/978-3-662-54577-5_18.

[2] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. "AutoPandas: neural-backed generators for program synthesis". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019). DOI: 10.1145/3360594.

[3] Jeffrey Foster, Brianna Ren, Stephen Strickland, Alexander Yu, Milod Kazerounian, and **Sankha Narayan Guria**. *RDL: Types, type checking, and contracts for Ruby*. Version 2.1.0. 2020. URL: https://github.com/tupl-tufts/rdl.

[4] Milod Kazerounian, **Sankha Narayan Guria**, Niki Vazou, Jeffrey S. Foster, and David Van Horn. "Type-level computations for Ruby libraries". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 2019. DOI: 10.1145/3314221.3314630.

[5] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation". In: *J. Comput. Secur.* 21.4 (2013). DOI: 10.3233/JCS-130469.

[6] Peter Naur and Brian Randell. “Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968”. In: (1969).

[7] Peter W. O’Hearn. “Incorrectness logic”. In: *Proc. ACM Program. Lang.* 4.POPL (2020). DOI: 10.1145/3371078.

[8] **Sankha Narayan Guria**, Jeffrey S. Foster, and David Van Horn. “Absynthe: Abstract Interpretation-Guided Synthesis”. In Submission. 2022. URL: https://www.cs.umd.edu/~sankha/drafts/absynthe.pdf.

[9] **Sankha Narayan Guria**, Jeffrey S. Foster, and David Van Horn. “RbSyn: Type- and Effect-Guided Program Synthesis”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021. DOI: 10.1145/3453483.3454048.

[10] **Sankha Narayan Guria**, Niki Vazou, Marco Guarnieri, and James Parker. “ANOSY: Approximated Knowledge Synthesis with Refinement Types for Declassification”. In: *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022. DOI: 10.1145/3519939.3523725.