# Research Statement | Xiaokang Qiu

✉ xkqiu@purdue.edu • engineering.purdue.edu/~xqiu/

## Overview

My research mission is to make computer programming easier, more reliable, and more productive. Computer software has revolutionized our daily lives, but software developers' lives have not advanced commensurately. Programming remains "one of our most demanding intellectual activities" as E.W. Dijkstra said three decades ago [2]. For example, as of 2018, it would cost approximately $14.7 billion USD to re-develop the Linux kernel [29], giving a cost per line equal to $733. The main thrust of my research is to develop algorithmic techniques to advance **program synthesis** — the process of automatically generating programs that meet the user's intent. As one of the most central research problems in computer science and AI, program synthesis is notoriously challenging. Nonetheless, in the last two decades, largely driven by the flowering of constraint solving and machine learning techniques, this traditional research area has experienced a renaissance and witnessed emerging programming paradigms (e.g., program sketching [20]), automated tools (e.g., the plethora of syntax-guided synthesis solvers [1]), and prominent applications (e.g., the FlashFill feature of Microsoft Excel [4]).

Despite the promising progress, program synthesis still has a long way to go. On one hand, to replace human written code with machine generated code in any production software, the synthesizer has to formally guarantee that the produced program is functionally correct, performant, preferable etc. These topics intersect with program verification, a well known challenging research area which has been studied for more than five decades [5]. On the other hand, Today's program synthesizers still can only be mastered by experts who can articulate their intent using formal specification or informative examples, determine preferable programs by writing ranking functions, and guide the synthesizer with domain knowledge. Moreover, while each of the two problems alone is already challenging enough, they are unfortunately tangled and have to be reconciled in a synthesizer. To this end, my general research program is to *develop **accessible** synthesis tools that help regular programmers produce high-quality, **trustworthy** programs*.

In the rest of this statement, I will highlight some of my main contributions to program synthesis, while mentioning along the way how the accessibility and trustworthiness are reconciled in these results. I will then conclude with my very recent or ongoing work and future directions.

## Provably-Correct Heap-Manipulating Programs

Heap-manipulating programs are one of the least understood classes of programs. Manipulations of dynamically allocated memory are pervasive in low-level computer systems: garbage collectors, OS kernels, device drivers, mobile browsers, etc. Moreover, the functional correctness of these programs is highly desirable, as they should provide a secure and trustworthy platform for higher-level applications. Unfortunately, building programs in this class eludes existing automatic techniques and poses one of the greatest challenges in software verification and synthesis. I have developed novel logics, decision procedures and methodologies for verifying heap-manipulating programs.

### Decidable Logics and Automated Reasoning

Programming tools are usually backed by efficient logic solvers for various theories such as integers, reals, bitvectors, arrays and strings. An important missing piece is decidable logics for tree data-structures,

which holds promise to automate many verification and synthesis problems not possible before. To this end, I have developed $Strand$ ("STRucture ANd Data") [14, 15], an expressive logic that combines graph-logic and data-logic, resulting in the first decision procedures that can prove properties of tree-like data-structures.

I have also developed $Dryad_{dec}$, another decidable logic that allows reasoning about tree data-structures with measurements [24]. The crux of the decidability proof is a small model property which allows us to reduce the satisfiability of $Dryad_{dec}$ to quantifier-free linear arithmetic theory which can be solved efficiently using SMT solvers. This decision procedure has been the first one that can prove full correctness of programs manipulating AVL trees and red-black trees.

Many synthesis tasks are essentially optimization of program, and the correctness of the optimizations boils down to reasoning about transformations between tree traversals (over the abstract syntax trees representing the programs). Recently, my group has developed $Retreet$ [27], a framework that reasons about transformations between tree-traversal programs. Our idea is a stack-based representation for iterations in tree traversals and an encoding to Monadic Second-Order logic over trees. $Retreet$ can automatically discover and verify optimizations for complex traversals on real-world data structures, such as cascading style sheets (CSS) and cycletrees, which are not possible before.

### Natural Proofs and Natural Synthesis

The decision procedures above allow a class of verifiers to boil down the whole verification task to a set of queries to the decision procedure, leading to push-button systems which are fully-automatic. Unfortunately, theoretical results suggest their complexity is high and their expressivity is limited. Moreover, they are black boxes from the perspective of the synthesizer and hence hard to be integrated into existing synthesis frameworks such as counterexample-guided inductive synthesis (CEGIS).

To this end, I proposed a radically new methodology called natural proofs [16]. The key insight behind natural proofs was that, to automate the verification process, it is unnecessary to thoroughly search for all possible proofs, which is the case for all decidability-based approaches. The idea of the natural proof methodology is: a) to identify a class of natural proof tactics that mimics the human way of proving heap-manipulating programs; and b) to build algorithms that efficiently and thoroughly search this class of proofs. Based on this methodology, I have developed VCDryad [22, 18], the first SMT-based program verifier against separation logic. VCDryad showed that the combination of separation logic, natural proofs and SMT solving is amenable to automated reasoning with structure, data and separation: it has automatically verified the full correctness of a wide variety of challenging C programs, including a large number of well-known routines from open-source libraries (GTK library, OpenBSD, etc.) and operating systems (Linux kernel and ExpressOS, a secure mobile OS).

I further integrated above verification techniques to synthesis tools. For instance, I extended Natural Proofs to the context of program synthesis and proposing Natural Synthesis [23], a novel verification-synthesis integration. In particular, he designed ImpSynt, a synthesis-enabled language that allows the user to describe only a high-level skeleton of the program, and developed the first synthesizer that can produce imperative, data-structure manipulations and their proofs in tandem, from these skeletons.

Other researchers have explored natural proofs in a different context and successfully verified the Windows Phone Universal Serial Bus (USB) driver. The natural synthesis methodology has influenced researchers from the Database community.

## General-Purpose Synthesis Algorithms and Tools

I have co-developed several influential, user-friendly program synthesis tools, including $Sketch$, $JSketch$ and $DryadSynth$.

## Scalable Sketch-Based Synthesis

*Sketch-based synthesis* lets the programmer specify a synthesis problem as a sketch or template, which is a program that contains some unknowns to be solved for and some assertions to constrain the choice of unknowns. Sketch-based synthesis was popularized by the Sketch [21] synthesis tool, which has seen many successful applications. As a developer of $Sketch$, I have co-developed several techniques that make Sketch more useful in practice. The following techniques have been implemented and integrated into the latest Sketch distribution.

Despite many successful applications, $Sketch$ still does not scale to programs that are even hundreds of lines long, and making sketch-based synthesis technology more scalable is an important research question. When a sketch involves too many unknowns which encode a too large search space, the user typically needs to concretize/fill some of the critical unknowns to reduce the size of the search space. Based on this key observation, I co-developed a novel synthesis algorithm called *Adaptive Concretization* [12, 13]. The key idea is to automate the human concretization process by estimating highly influential unknowns. Then by randomly concretizing these unknowns we can often speed up the overall synthesis algorithm, especially when we add parallelism. Experiments show that the new algorithm often outperforms vanilla $Sketch$, sometimes very significantly.

## Program Sketching for Java

I have also co-developed $JSketch$, the first Java sketching tool that delivers sketch-based synthesis to the hands of Java programmers [11]. $JSketch$'s input is a partial Java program that may include holes, expression generators, and class generators. $JSketch$ then translates the synthesis problem into a $Sketch$ problem, and synthesizes an executable Java program by interpreting the output of $Sketch$. $JSketch$ has also inspired the development of other Sketch-based synthesizers, such as EdSketch [6] and SkASP [19].

A special challenge for $JSketch$ is synthesizing programs that use standard Java libraries. The known approach was to model libraries with mock library implementations that perform the same function in a simpler way. However, mocks may still be large and complex, and must include many implementation details, both of which could limit synthesis performance. To address this problem, we extended $JSketch$ with algebraic specifications for describing library behaviors, e.g., encryption followed by decryption (with the same key) is the identity. This enhanced $JSketch$ handles algebraic specifications with a rewriting-based encoding. With this extension, $JSketch$ successfully synthesized provably correct Java programs that use data structure libraries, cryptography libraries, or the file system. Most of them were synthesized within 10 minutes [17].

## Syntax-Guided Synthesis Algorithms

Sketch-based synthesis belongs to a more general theme called *Syntax-Guided Synthesis (SyGuS)*, which formulates the synthesis task as a computational problem of searching for a program expression that meets both syntactic and logical constraints. Inspired by other solver competitions which have catalyzed the plethora of SMT solvers and verifiers, the community has standardized the problem input format and held an annual competition called SyGuS-Comp since 2014 [1].

I have been leading the development of $DryadSynth$, the most recent winner solver of SyGuS-Comp in the conditional linear integer arithmetic (CLIA) track (2018 and 2019). A key novelty behind $DryadSynth$ is *Cooperative Synthesis* [8], a framework in which a synthesis problem is repeatedly split into subproblems and solved by deduction or enumeration separately. We have recently extended $DryadSynth$ to support the synthesis of bit-vector manipulations [3]. The key driving power is a distinct enumeration strategy with the guidance of bottom-up enumeration and large language models. $DryadSynth$ successfully solved 31 bit-vector synthesis problems for the first time, including 5 renowned Hacker's Delight problems [28].

## Lowering the Barrier for Users

While program synthesis has advanced a lot and enabled real-life code generation, modern-day synthesizers (including those I co-developed) can still only be mastered by experts. One vital problem is that these techniques have not been integrated to the traditional software development process that most regular programmers are familiar with. In recent years, I have made inroads toward lowering the barrier of algorithmic program synthesis for regular programmers – in several disparate contexts.

### Program Sketching *without* Custom Templates

A fundamental problem that significantly impacts the usability of $Sketch$ (and all sketch-based synthesis techniques) is the requirement to provide a template for every synthesis task. We showed in [9] that, at least in the context of recursive transformations on algebraic data-types (ADTs), a user does not have to write a custom template for every synthesis problem, but can instead rely on a generic template from a library. Moreover, we developed a new optimization called *inductive decomposition*, which achieves asymptotic improvements in synthesis times for large and non-trivial ADT transformations. This optimization, together with the user guidance in the form of reusable templates, allows the enhanced $Sketch$ to attack problems that are out of scope for prior synthesis techniques.

### Library-Based Synthesis *without* Hand-Crafted Models

While the algebraic library models in [17] let $JSketch$ handle libraries more efficiently, it is not too different from a user's perspective – they still have to provide carefully crafted mocks or models to $JSketch$, requiring a lot of extra manual work. Recently, I explored an approach aiming at reducing the user's burden of writing library models, and developed $Toshokan$ ("library" in Japanese) [7], a bootstrapping synthesis framework which extends traditional counterexample guided inductive synthesis (CEGIS) with an automatically built library model from logged behavior of the library. $Toshokan$ has been implemented and integrated into $JSketch$. Comparing to vanilla $JSketch$ using mocks or models, $Toshokan$ reduces up to 159 lines of code of required manual inputs, at the cost of less than 40 second of performance overheads.

Similar challenges are also faced by the users of static analyzers for Android apps, who have to provide abstract models for Java frameworks (e.g., Swing and Android). I have co-developed $Pasket$ [10], a first step toward automatically generating Java framework models. $Pasket$ takes as input the framework API, together with tutorial programs that exercise the framework, and emits framework models by instantiating design patterns to the framework API. This tool has been able to synthesize models for a subset of Swing and a subset of Android.

### Quantitative Synthesis *without* Hand-Crafted Objectives

Writing formal specification for synthesis is a major barrier for average programmers. In particular, in some quantitative synthesis scenarios (such as network design), the first challenge faced by the user is expressing their optimization targets. To address this problem, my group developed *Comparative Synthesis* [25, 26], an interactive synthesis framework that learns near-optimal programs through comparative queries, without user-specified optimization targets. We developed a voting-guided learning algorithm which provides a provable guarantee on the quality of the found program. This approach has been embodied in a system called $Net10Q$ for wide-area network design. Experiments with oracles and a pilot user study with network practitioners and researchers show $Net10Q$ is effective in finding allocations that meet diverse user policy goals in an interactive fashion.

## Ongoing and Future Work

I will continue my work on making program synthesis more trustworthy and user-friendly, by *developing advanced **programming systems** and making them accessible to **domain users***.

### Algorithmic Innovations

On the Systems side, I am committed to the continued development and improvement of practical, general-purpose program synthesis tools including DryadSynth and JSketch. These tools have the potential to significantly broaden the base of people who can overcome the programming-related challenges in their own fields, by delivering the power of algorithmic program synthesis to their hands. In particular, I believe that modularity—a key enabling concept in many disciplines for hiding and managing complexity of large systems—remains an essential missing piece in the program synthesis puzzle and will potentially lead to the next wave of breakthrough for program synthesis. I envision a modular synthesis framework which decouples complex verification and synthesis tasks, decomposes a large synthesis task to multiple verification or synthesis subtasks, and delegates each subtask to an appropriate, off-the-shelf solver. This line of work is currently supported by my NSF CAREER award (CCF-2046071).

Besides conventional formal methods-based approaches, I am enthusiastic about the advent of large language models (LLMs) and look forward to harnessing the power of this revolutionary thrust for my research program. Our recent research in bit-vector synthesis [3] has shown that while GPT models are incapable of producing complete solutions for most challenging synthesis problems, we can extract some critical components that significantly assist in DryadSynth's search process. I am eager to delve deeper into understanding the strengths and weaknesses of LLMs and exploring more novel interaction paradigms between LLMs and traditional enumerative and deductive synthesis techniques.

### Engaging with Domain Users

On the Domain side, I will explore novel programming techniques to the fields of Networking and Quantum Programming. In the realm of Networking, I believe what remains missing is a flexible language for data plane programming. Despite promising development in recent years, there is still a long way to go to let average network architects program data planes in a natural and efficient way. Such a language will transplant and adapt a suite of computer-aided programming techniques I developed—including sketch-based synthesis, comparative synthesis and automated reasoning—to the field of networking. This endeavor has been supported by two consecutive NSF Formal Methods in the Field (FMitF) awards (CCF-1837023 and CCF-2319425).

Furthermore, I have been collaborating with colleagues on applying synthesis techniques to the exciting domain of Quantum Programming. Our particular focus lies in the discovery of quantum circuit identities [30], i.e., finding general patterns of quantum circuit that can be transformed to a simpler and (approximately) equivalent circuits. This pursuit necessitates the development of novel synthesis and verification techniques and holds great promise for quantum program compilation, analysis, and education.

## References


[1] Alur, R., Singh, R., Fisman, D., and Solar-Lezama, A. Search-based program synthesis. *Commun. ACM 61*, 12 (Nov 2018), 84–93.

[2] Dijkstra, E. W. Visuals for bp's venture research conference. EWD963, June 1986.

[3] Ding, Y., and **Qiu**, **X**. Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. *To appear in Proc. ACM Program. Lang. 8*, POPL (jan 2024).

[4] Gulwani, S., Harris, W. R., and Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM 55*, 8 (Aug 2012), 97–105.

[5] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct 1969), 576–580.

[6] Hua, J., Zhang, Y., Zhang, Y., and Khurshid, S. Edsketch: execution-driven sketching for java. *International Journal on Software Tools for Technology Transfer 21*, 3 (Mar 2019), 249–265.

[7] Huang, K., and **Qiu**, **X**. Bootstrapping library-based synthesis. In *Static Analysis* (Cham, 2022), G. Singh and C. Urban, Eds., Springer Nature Switzerland, pp. 272–298.

[8] Huang, K., **Qiu**, **X**., Shen, P., and Wang, Y. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, pp. 1159–1174.

[9] Inala, J. P., Polikarpova, N., **Qiu**, **X**., Lerner, B. S., and Solar-Lezama, A. Synthesis of recursive adt transformations from reusable templates. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2017), A. Legay and T. Margaria, Eds., Springer Berlin Heidelberg, pp. 247–263.

[10] Jeon, J., **Qiu**, **X**., Fetter-Degges, J., Foster, J. S., and Solar-Lezama, A. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, Association for Computing Machinery, pp. 156–167.

[11] Jeon, J., **Qiu**, **X**., Foster, J. S., and Solar-Lezama, A. Jsketch: Sketching for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, Association for Computing Machinery, pp. 934–937.

[12] Jeon, J., **Qiu**, **X**., Solar-Lezama, A., and Foster, J. S. Adaptive concretization for parallel program synthesis. In *Computer Aided Verification* (Cham, 2015), D. Kroening and C. S. Păsăreanu, Eds., Springer International Publishing, pp. 377–394.

[13] Jeon, J., **Qiu**, **X**., Solar-Lezama, A., and Foster, J. S. An Empirical Study of Adaptive Concretization for Parallel Program Synthesis. *Formal Methods in System Design (FMSD) 50*, 1 (March 2017), 75–95.

[14] Madhusudan, P., Parlato, G., and **Qiu**, **X**. Decidable logics combining heap structures and data. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, Association for Computing Machinery, pp. 611–622.

[15] Madhusudan, P., and **Qiu**, **X**. Efficient decision procedures for heaps using strand. In *Static Analysis* (Berlin, Heidelberg, 2011), E. Yahav, Ed., Springer Berlin Heidelberg, pp. 43–59.

[16] Madhusudan, P., **Qiu**, **X**., and Stefanescu, A. Recursive proofs for inductive tree data-structures. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2012), POPL '12, Association for Computing Machinery, pp. 123–136.

[17] Mariano, B., Reese, J., Xu, S., Nguyen, T., **Qiu**, **X**., Foster, J. S., and Solar-Lezama, A. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct 2019), 132:1–132:25.

[18] Pek, E., **Qiu**, **X**., and Madhusudan, P. Natural proofs for data structure manipulation in c using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, Association for Computing Machinery, pp. 440–451.

[19] Singh, R., Singh, R., Xu, Z., Krosnick, R., and Solar-Lezama, A. Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2014), K. L. McMillan and X. Rival, Eds., Springer Berlin Heidelberg, pp. 395–414.

[20] Solar-Lezama, A. Program sketching. *International Journal on Software Tools for Technology Transfer 15*, 5 (Oct 2013), 475–495.

[21] Solar-Lezama, A. *The Sketch Programmers Manual*, 2020. https://people.csail.mit.edu/asolar/manual.pdf.

[22] **Qiu**, **X**., Garg, P., Ștefănescu, A., and Madhusudan, P. Natural proofs for structure, data, and separation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2013), PLDI '13, Association for Computing Machinery, pp. 231–242.

[23] **Qiu**, **X**., and Solar-Lezama, A. Natural synthesis of provably-correct data-structure manipulations. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct 2017), 65:1–65:28.

[24] **Qiu**, **X**., and Wang, Y. A decidable logic for tree data-structures with measurements. In *Verification, Model Checking, and Abstract Interpretation* (Cham, 2019), C. Enea and R. Piskac, Eds., Springer International Publishing, pp. 318–341.

[25] Wang, Y., Jiang, C., **Qiu**, **X**., and Rao, S. G. Learning network design objectives using a program synthesis approach. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2019), HotNets '19, Association for Computing Machinery, pp. 69–76.

[26] Wang, Y., Li, Z., Jiang, C., **Qiu**, **X**., and Rao, S. Comparative synthesis: Learning near-optimal network designs by query. *Proc. ACM Program. Lang. 7*, POPL (jan 2023), 91–120.

[27] Wang, Y., Liu, J., Zhang, D., and **Qiu**, **X**. Reasoning about recursive tree traversals. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2021), PPoPP '21, Association for Computing Machinery, pp. 47–61.

[28] Warren, H. S. J. *Hacker's Delight*, 2 ed. Addison-Wesley Professional, 2012.

[29] Wikipedia. https://en.wikipedia.org/wiki/linux_kernel#estimated_cost_to_redevelop, 2022.

[30] Xu, M., Li, Z., Padon, O., Lin, S., Pointing, J., Hirth, A., Ma, H., Palsberg, J., Aiken, A., Acar, U. A., and Jia, Z. Quartz: Superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2022), PLDI 2022, Association for Computing Machinery, p. 625–640.