Research Statement

Dongjie He

July 2023
dongjieh@cse.unsw.edu.au

1 My Approach to Research

I wasn't the guy who started learning computer science as an undergraduate. In high school, I was always interested in *mathematics* and *physics*. I dreamed of being a physicist or astronomer one day (that's why I studied *astronomy* as my second bachelor's degree). Unfortunately, I ended up majoring in *educational technology* in college. It introduced how to apply modern technologies, especially computer-related ones, to education, and I got the chance to learn some basic programming skills. At that time, a question bothered me: how can a program written by a human be understandable by a machine? This question drove me to learn computer science and led me to the area of **compilers and program analysis**.

Working in compilers and program analysis is interesting, challenging, and impactful in improving real-world programs or systems. It is one of the areas in computer science where theoretical results are very often put into practice. I really enjoy working in the area, especially on fundamental problems. I am happy to see that any improvement I make instantly applies to a wide variety of programs. However, the problems in compilers and program analysis are very challenging. Most of them are NP-complete or undecidable in theory. How to approximate these problems so that they can be solved efficiently yet precisely is fascinating.

My approach to research is influenced by both engineering and theory. When I start to address a problem, I first read the papers to get a basic understanding of the problem from the theory. Then, I will often build a small prototype to test out my idea or to gain insights. Such an engineering process is very helpful for thinking about the problem deeply. Once I feel I have understood the problem sufficiently, I will re-read what others have published on the problem and think deeply about what I can do about the problem from the first principle. Once the idea is fixed, the approach is generally either inductive or deductive. I use both of them in my research depending on the kinds of problems addressed. When the approach is inductive (e.g., [6, 8, 4]), I will formalize my idea after I have a working implementation. The process of formalization often suggests new ways of simplifying or improving the implementation and exposes corner cases that I would not have considered. When the approach is deductive (e.g., [5, 3]), I will quickly implement a prototyping implementation to empirically validate whether the deductive approach works correctly or efficiently in the real world. Thus, both engineering and theory play important roles in my approach to research. This reminds me of Donald Knuth's credo "Theory and practice are not mutually exclusive; they are intimately connected. They live together and support each other."

Finally, I fully embrace the openness of research. Despite that most of my research ideas were personal inspirations, I strongly believe that the openness of research is beneficiary to all researchers. To facilitate cooperation, communication, and the advancement of research, I release all of my implementations as open source so that other researchers can build upon my work. I provide Docker images for all my research so that everyone can reproduce the evaluation results used in my paper.

2 Research Interests

I have broad interests in programming languages and compiler techniques, including compiler optimization, program analysis, program verification, and type systems. In the past few years, I focused on fundamental static program analysis techniques like **pointer analysis** and **data-flow analysis**, for large-scale real-world software applications (with millions of lines of code). These techniques are widely used in compiler optimization, software security, and many others.

Below outlines the two specific research areas on which I was working.

2.1 Pointer Analysis

As one of the most fundamental problems in program analysis and compiler optimization, pointer analysis attempts to answer the following questions, "What can a pointer point to?" and "Can pointers a and b point to the same thing?". It has applications and implications across almost all program analyses and optimizations. Understanding pointers is essential to the understanding of almost any program.

Despite its importance, pointer analysis is still one of the most vexing problems in program analysis. Take Java pointer analysis as an example, context sensitivity is regarded as the most effective technology in improving precision. At the same time, it makes pointer analysis less effective and less scalable.

Early selective techniques try to improve the efficiency of pointer analysis while maintaining its precision by only analyzing a subset of pre-selected precision-critical methods context-sensitively while others context-insensitively. Jingbo and I [12, 11, 6, 7] have made one step further. We have proposed fine-grained selective context-sensitive analysis techniques which only analyze a subset of pre-selected precision-critical variables/objects context-sensitively while others context-insensitively. In addition, we have designed Qilin [9], a new Java pointer analysis framework that supports fine-grained context sensitivity.

In 2021, I proposed a context-debloating technique for object-sensitive pointer analysis [8, 10]. The technique has been shown to be very effective in accelerating object-sensitive pointer analysis while preserving nearly all precision. This year, I have designed another context-debloating technique (which is pattern-based) that can significantly improve the precision of object-sensitive pointer analysis further with only negligible precision loss. This work has been conditionally accepted by OOPSLA 2023 [4].

Looking forward, I have great interest in further advancing the state-of-the-art pointer analysis and finding new applications for analysis results. First, I keep working on improving pointer analysis along the thread of research in context debloating as there is still room for improvement based on my empirical experiment. Second, I am currently working on applying the divide-conquer idea in pointer analysis. Due to the nature of the cubic bottleneck, any improvement technique with constant coefficients cannot fundamentally solve the scalability problem of pointer analysis. Why not divides the program into multiple modules and analyze each part independently? To achieve this, one must know how to divide the programs, the data dependencies between each module, and the data interaction between modules. These issues are challenging but very attractive.

2.2 Data-flow Analysis

Data-flow analysis, which analyzes how data propagates through a program, is another fundamental problem in program analysis and compiler optimization. It aims to gain insights into how data values change throughout the execution of a program, allowing developers to optimize the program, detect potential errors, and perform various program transformations.

In 1995, Reps et al proposed a special kind of data-flow analysis called IFDS [13], which is now widely used in many applications such as taint analysis, and type-state analysis. They smartly transformed the data-flow analysis problem into a graph reachability problem and proposed the IFDS algorithm which runs in $O(E \cdot D^3)$ time and $O(E \cdot D^2)$ space. The algorithm could be very time- and memory-intensive when used in analyzing large programs with millions of lines of code.

In 2019, I proposed a new technique called sparse IFDS [5], which directly propagates data-flow facts to their next use points rather than the next program points along the traditional control flow. The technique enables IFDS-based taint analysis to run significantly faster while consuming fewer memory resources and achieving the same precision. The work won me an *ACM SIGSOFT Distinguished Paper Award*.

To reduce the memory consumption of IFDS, I have also proposed a fine-grained fact-level garbage collection algorithm named Fpc for IFDS-based data flow analysis [3]. Fpc can collect a substantial amount of path edges of IFDS during the runtime without hurting the correctness and termination property of IFDS. Furthermore, it could completely avoid redundant computation problem that exists in an earlier approach [1]. Consequently, Fpc could significantly reduce the memory consumption of IFDS while preserving nice properties. This work won me an *ACM SIGSOFT Distinguished Artifact Award*.

In addition, Yujiang and I have addressed a special issue of IFDS-based taint analysis, where data abstractions decorated with different activation statements are propagated redundantly by the IFDS solvers

[2]. We have proposed a "merge-and-replay" policy to consolidate the logically equivalent value flows effectively. This work has been accepted by ASE 2023.

3 Future Directions

After working in the area of program analysis for years, I feel that program analysis is full of interesting research problems that have practical applications. I am very excited about the possibilities. I will keep working in this area and am looking forward to contributing to whatever I can.

Compiler optimization is a closely relevant area (to program analysis) that I would like to explore. I am super interested in how to use the program analysis techniques I have learned to improve compilation optimization techniques, especially in areas like AI compilers and high-level synthesis for circuit design.

Lastly, I would also like to explore how to use program verification techniques to verify software systems such as operating systems, compilers, smart contracts, and industrial safety-critical software in the foreseeable future.

References

[1] Steven Arzt. Sustainable solving: Reducing the memory footprint of ifds-based data flow analyses using intelligent garbage collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1098–1110, New York, NY, USA, 2021. IEEE.

[2] Yujiang Gui*, Dongjie He*, and Jingling Xue. Merge-replay: Efficient ifds-based taint analysis by consolidating equivalent value flows. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Accepted*, pages 1–11, New York, NY, USA, 2023. IEEE.

[3] Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. Reducing the memory footprint of ifds-based data-flow analyses using fine-grained garbage collection. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 101–113, New York, NY, USA, 2023. Association for Computing Machinery.

[4] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. A container-usage-pattern-based context debloating approach for object-sensitive pointer analysis. *Proceedings of the ACM on Programming Languages*, (OOPSLA (Conditional Accept)):1–26, 2023.

[5] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2019.

[6] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Selecting context-sensitivity modularly for accelerating object-sensitive pointer analysis. *IEEE Transactions on Software Engineering*, 49(2):719–742, 2023.

[8] Dongjie He, Jingbo Lu, and Jingling Xue. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–91, New York, NY, USA, 2021. IEEE.

[9] Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222, pages 30:1–30:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[10] Dongjie He, Jingbo Lu, and Jingling Xue. Ifds-based context debloating for object-sensitive pointer analysis. *ACM Trans. Softw. Eng. Methodol.*, 32(4):1–45, jan 2023.

[11] Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: Cfl-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46, 2021.

[12] Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[13] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.