Formally Verified Resource Bounds through Implicit Computational Complexity

Neea Rusch
Augusta University
Augusta GA, USA
nrusch@augusta.edu

Abstract

Automatic complexity analysis has not reached mainstream adoption due to outstanding challenges, such as scalability and usability, and no formally verified analyzer exists. However, the need to evaluate resource usage is crucial: even a guaranteed correct program, whose memory usage exceeds available resources, is unreliable. The field of Implicit Computational Complexity (ICC) offers a potential avenue to resolving some of these outstanding challenges by introducing unique, machine-independent, and flexible approaches to program analysis. But since ICC techniques are mostly theoretical, it is unclear how strongly these assumptions hold in practice. This project defines a 3-directional plan—focused on practical analysis, compiler-integration, and formal verification—to assess the suitability of ICC to address outstanding challenges in automatic complexity analysis.

Keywords: Implicit Computational Complexity, Automatic Complexity Analysis, Program Verification

1 Motivation

Computational complexity is a foundational pillar of computer science: a staple of university curricula and technical interviews in the industry, it also gives rise to some of the most famous unsolved problems in theoretical computer science. Given the prevalence and extensive study of the topic, one might expect tools to perform automatic and repeatable complexity analysis would be equally ubiquitous, but this is not the case. While automated resource analysis is an active area of research and several analyzers do exist, they come with challenges and lack certified correctness. None of the tools has reached industry-level adoption and consequently mainstream software is being developed without their support. This limits opportunity to measure and optimize resource usage and hinders assurance of safety and usability properties potentially compromised by excessive resource usage. It is of course impossible to analyze behavior of all programs in general, as proven by Rice's theorem [12], but if we can identify feasible techniques that cover sufficiently large classes of programs, automatic complexity analysis on realistic programs at scale advances closer to materialization.

The field of Implicit Computational Complexity (ICC) [4] offers one conceivable pathway toward advancing this goal. By embedding in programs a guarantee of resource usage, it captures implicit characterizations of complexity classes in machine-independent ways. Unlike traditional methods, ICC systems are generally expressive enough to write down actual algorithms and have introduced numerous elegant program analysis techniques that are often orthogonal to alternative approaches [1]. But ICC techniques have nonetheless remained primarily as treasures of the theorists, with only a few known practical applications [1, 2, 7, 11].

This project bridges the gap between theory and application by leveraging techniques from ICC. This will enable evaluation of their efficiency and powerfulness, and drive forward development of richer systems to cover larger classes of programs. Furthermore, it represents the first attempt to certify the correctness of a complexity analyzer; surprisingly this has not been done before[1]. Successful completion of this project would show that certifiably correct complexity analysis is achievable (3.3), and that ICC techniques can be used to obtain scalable (3.2), efficient and practical complexity analysis (3.1). These results would extend capabilities in automatic complexity analysis and take it a few steps closer to becoming a standard in modern development workflows.

2 Problem

An accurate definition the research problem requires we make this important clarification to the intended purpose: the intent of this work is *not* to generate the most elaborate complexity analyzer. Rather, after identifying challenges in the state-of-the-art techniques, the main idea is to evaluate suitability of ICC to resolve those challenges. Positive results serve to advance future versions of concrete tools, but such enhancements are outside the scope of this work.

The state-of-the-art in automatic complexity analysis is captured by tools such as AProVE, ComplexityParser, and RaML. They perform complexity analysis based on different approaches: integer term rewrite systems [5], tier-based typing [6], and cost amortization [7], respectively. They offer varying levels of maturity and feature support and target different programming languages and paradigms (multiple formalisms, `Java`, `OCaml`). The outstanding challenges, in automatic complexity analysis in general, relate to choices in methodologies (scalability, absence of compositionality [3]);

[1] Carbonneaux et al. [3] share a correctness proof for the technique, but the analyzer itself, C4B, appears not to be certified.

user interaction (level of user control, interpretability of results [3, 7]); and absence of formal verification to ensure correctness [3]. The effects of these challenges reduce tool applicability and performance, negatively impact usability, and lower trust in their results.

It is a reasonable conjecture that ICC-based techniques could resolve some of these issues, particularly those related to methodology and formal verification. ICC systems commonly feature simplified syntax, are machine-independent, language-agnostic, and abstracted to avoid challenges related to termination and iteration bounds. These properties suggest they offer scalable and widely applicable alternatives, especially if introduced at compile-time[2]. Within ICC, we can also find naturally compositional systems [8]. Compositionality facilitates fast analysis by creating opportunity for parallelization, and because unmodified sections of source code do not require re-analysis on repeated executions. Furthermore, ICC systems seem suitable to formal verification based on their rigorous mathematical foundation. However, the factuality of these assumptions remains mostly unestablished, and it is the purpose of this work to evaluate them.

3 Approach

To approach the identified problem, we construct the following 3 hypotheses of equal importance, as related to ICC systems and complexity analysis:

1. ICC systems can be applied in realistic settings (3.1).
2. Scalability can be achieved through compilers (3.2).
3. Formal verification is achievable (3.3).

We then define a matching subproject to address each hypothesis: the first will evaluate the suitability of ICC as a technique in practical analysis; the second will implement ICC-based analysis as a compiler pass; and the third will strive to produce a formally verified static analyzer for resource usage. These 3 subprojects are mutually exclusive: they can be completed in parallel, and success (or failure) in one branch does not impact the rest. Collective success across all subprojects would strongly enforce the argument for suitability of ICC to practical applications, and bring us closer to achieving formally verified, efficient and scalable automatic complexity analysis in compilers. A brief overview of the methods and evaluation strategies follows next.

3.1 Realizing ICC

The foundation of this work is a theoretical method called *mwp*-analysis [8], explained shortly below. The question of interest then becomes, given this method, does it provide an efficient and feasible technique that can be applied in practice? If yes, it then affirms the assumption that ICC-based techniques are also relevant in practical scenarios.

[2]Compilers are an optimal target because they offer multi-language support and inherent ability to translate between different representations.

Methodology. *mwp*-analysis is a computational method that allows formal derivations of true statements about programs. Founded on the *mwp*-calculus, it uses matrices that record data flows between variables as commands are executed. For a program C and matrix M, the analysis computes a polynomial bound—if it exists—on the sizes of variables of C, captured in M. The main theorem is the soundness property, that guarantees iff there exists a derivation in the calculus, every value computed by C is bounded by a polynomial in inputs [8].

Practical application of this analysis is a non-trivial task, because the original derivability problem is NP-complete, the inference rules of the calculus are nondeterministic, and the syntax is not rich enough to support realistic analysis. Our practical and extended method resolves these challenges by extending the original method and its syntax, redefining its derivation rules, and generating efficient data representations and algorithms to process the data and to compute the result. Our method is implemented as a standalone static analyzer pymwp, to demonstrate and measure the efficiency of the improved and extended technique. The analyzer takes as input programs written in a subset of the `C` language, and returns a multi-variate growth bound, if one can be derived. Since the result provides individual bounds of each variable, it is useful to programmers for obtaining fine-grained and actionable feedback.

Evaluation. Work on this subproject has reached the stage of publication [1]. We benchmarked pymwp using 42 custom benchmarks, on which it produced correct results efficiently (< 5 s). This demonstrates a positive result for the first hypothesis. However, further work remains. We need to extend the syntactical support of `C` language to enable evaluation on standard benchmark suites. While the multivariate result is useful, it is also problematic when attempting to compare results between alternative tools that produce single-variate bounds. Addressing those issues will enable more extensive measurements. Nonetheless, the result is promising. We are confident this technique can be extended and will continue to pursue this direction.

3.2 ICC Meets Compilers

Earlier work by Moyen et al. [11] proved that performing ICC-based analysis in compilers is not only achievable but can also offer new techniques for optimizing resource usage during compilation. This seminal paper was the first known application of ICC in a mainstream compiler (LLVM) and serves as inspiration here. The assertion to evaluate is whether ICC can be applicable to a large number of programs and source programming languages, if the techniques are implemented as compiler passes. While the earlier work suggests this assertion holds, it left open questions about measured scalability and applications of alternative ICC techniques, and therefore further work remains.

Methodology. This subproject involves the LLVM compiler, and the goal is to implement an analysis method (possibly paired with optimization) on the intermediate representation (IR) in the compiler middle-end. Future decisions must be made regarding the exact analysis technique, the input syntax and expected program structure. The challenge with this approach is handling memory operations, which are often omitted in the definitions of ICC systems. This subproject is currently in active planning stage, and we hope to clarify these details shortly.

Evaluation. Several quantifiable measures for evaluating the success of this subproject exist: 1) the performance of the implemented pass and how much overhead it adds to compilation time, and 2) the proportion of programs that can be analyzed or optimized, and if the technique involves program transformation, 3) the execution time of the generated assembly can be measured before and after (clock time, LOC) on standard benchmark suites. A positive result, in combination with the LLVM compiler, would suggest scalability of these techniques, since the compiler supports multiple source and target languages.

3.3 Formally Verified Complexity

Formally verified toolchains, using machine-assisted mathematical proofs, are in high demand and especially vital for ensuring the correctness of critical software. While results are lacking in complexity analysis, realistic systems exist in related domains. The CompCert verified C compiler [10] establishes the foundation for provably correct realistic compilation and guarantees preservation of program semantics. Subsequent work by Jourdan et al. [9], in their case using abstract interpretation to analyze runtime errors, proved that formally verified static analysis is possible. The goal for this work is similar but using ICC to analyze program complexity.

Methodology. Reusing the *mwp*-analysis [8], the plan is to mechanically prove its correctness using the Coq proof assistant. In more detail, it requires defining the language and operational semantics, and inference rules of the *mwp*-calculus, and the matrix-based analysis; and proving correct the main theorem, the soundness property, $\vdash C : M$ implies $\models C : M$. The Coq proof assistant was selected for implementation because it was used in the earlier works, offers maturity and community support, and appears suitable for this project. The current plan is to implement and prove the system completely in Coq, rather than *a posteriori* validation. The written proofs from the original paper and pymwp version of the analysis will further help guide the Coq implementation.

This work is currently in early stages but seems feasible because we have firm understanding of the method and how to implement it. If a formally verified analyzer can be completed, a subsequent question of interest then becomes whether the analyzer can be integrated with a formally verified compiler, such as CompCert, to ensure compiler-preserving treatment of the analysis result.

Evaluation. The success of this work greatly depends on the ability to complete the Coq proofs and implementation. Evaluation metrics can be obtained relating to the efficiency of the analysis and its scalability, measured using suitable benchmarks, depending on the analysis source language.

4 Conclusion and Future Work

This project will assess the capabilities of ICC in resolving some of the outstanding challenges in automatic complexity analysis. The problem is further divided into 3 subprojects, each with different focus and independent results: efficient and practical analysis, scalability through compilers, and correctness through formal verification. Work in the latter two areas is still preliminary, but encouraging results exist in the first case to enforce the proposed assumptions. The emphasis of future work will be on LLVM compiler integration and formal verification using the Coq proof assistant. If all work can be completed successfully, it will be a step toward achieving formally verified, efficient and scalable automatic complexity analysis in mainstream compilers.

References

[1] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. 2022. mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity. In *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022 (LIPIcs, Vol. 228)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 26:1–26:23. https://doi.org/10.4230/LIPIcs.FSCD.2022.26

[2] Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.* 1, ICFP (2017), 43:1–43:29. https://doi.org/10.1145/3110287

[3] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 467–478. https://doi.org/10.1145/2737924.2737955

[4] Ugo Dal Lago. 2011. A Short Introduction to Implicit Computational Complexity. In *ESSLLI (LNCS, Vol. 7388)*, Nick Bezhanishvili and Valentin Goranko (Eds.). Springer, 89–109. https://doi.org/10.1007/978-3-642-31485-8_3

[5] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. 2017. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning* 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[6] Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. 2021. Complexityparser: An automatic tool for certifying polytime complexity of java programs. In *Theoretical Aspects of Computing - ICTAC 2021 (Lecture Notes in Computer Science, Vol. 12819)*. Springer, 357–365. https://doi.org/10.1007/978-3-030-85315-0_20

[7] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012 (LNCS, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

[8] Neil D. Jones and Lars Kristiansen. 2009. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.* 10, 4 (2009),

28:1–28:41. https://doi.org/10.1145/1555746.1555752

[9] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified C static analyzer. *ACM SIGPLAN Notices* 50, 1 (Jan. 2015), 247–259. https://doi.org/10.1145/2775051.2676966

[10] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[11] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. 2017. Loop Quasi-Invariant Chunk Detection. In *International Symposium on Automated Technology for Verification and Analysis*. 91–108. https://doi.org/10.1007/978-3-319-68167-2_7

[12] M. Sipser. 2012. *Introduction to the Theory of Computation*. Cengage Learning.