Certifying Implicit Computational Complexity

Neea Rusch

School of Computer & Cyber Sciences
Augusta University

Research Proposal
for Doctor of Philosophy Degree
in Computer & Cyber Sciences

December 2, 2022

Abstract

Complexity analysis offers developers better understanding of program's runtime behavior, but mechanical approaches to evaluate complexity properties are scarce and limited. This research proposal addresses this gap between computational complexity theory and its practical application. The main hypothesis is that techniques from Implicit Computational Complexity (ICC) provide new approaches to automatic program analysis and resolve certain limitations of the state-of-the-art complexity analysis techniques. This is unapparent because ICC systems have primarily been used for theoretical purposes and their practical applications are rare. The intent of this work is to evaluate the hypothesis along three directions. First to show that obtaining automatic program analysis with ICC is in fact achievable. Then, to demonstrate that ICC systems are viable candidates for achieving formally verified complexity analysis. Lastly, to confirm that ICC-based techniques find extended applications, e.g., in optimizing complexity properties during compilation. The formal verification aspect is particularly interesting because certifying the correctness of a complexity analysis technique has not been done before. Collectively these directions suggest that ICC is not just a treasure of the theorists but can move practical analyses a few steps closer to becoming a standard in modern development workflows.

Keywords: complexity, program analysis, mechanized proofs

Contents

1 Hypothesis and Specific Aims

In software verification, compile-time static analyses are used to assist developers in deriving quantitative information about program's runtime behavior. Many program properties can be analyzed, but obtaining information specifically about resource consumption is necessary when those properties form an integral part of program's correctness. For example, verifying cryptographic protocols and constant-time implementations require establishing their complexity bounds, and safety-critical systems must guarantee bounded memory consumption to avoid potential failure.

Despite this clear motivation to analyze complexity properties, solutions to perform such analysis automatically remain absent in modern software development workflows. Obtaining resource bounds for an arbitrary program is undecidable [1, p. 219]; however, by identifying techniques that cover sufficiently large classes of programs, automatic analysis becomes achievable. Prior attempts have been successful at e.g., analyzing various programming languages and obtaining tight bounds [2, 3, 4]. But the state-of-the-art techniques also have limitations, including general lack of compositionality, constrained user interaction, and high probability of implementation errors [5, 6]. The last is particularly troubling, because the correctness of the analyzer is a necessary precondition to trusting the result it computes.

The hypothesis of this project is that techniques from implicit computational complexity theory could resolve some outstanding issues in automated complexity analysis. Implicit Computational Complexity (ICC) [7] is a subfield of complexity theory, that aims at finding syntactic criteria to guarantee program's runtime behavior. By embedding in programs a guarantee of resource usage, it captures implicit characterizations of complexity classes in machine-independent ways. The systems are often compositional, and abstract away e.g., iteration bounds, thus sidestepping issues that other analysis techniques must handle. Further, their mathematical foundation makes them potentially suitable for certification. However, since ICC systems are primarily used for theoretical purposes, and their practical applications are rare [8, 9, 10, 11, 12, 13], it remains unclear whether this assumption holds.

Specific aims. This project aims to reduce the gap between complexity theory and its practical application, by leveraging techniques from ICC, to evaluate their suitability and powerfulness in automatic program analysis. This includes the plan to certify the correctness of a computational complexity technique, which has not been done before (cf. subsection 2.4). More precisely, the project has the following goals:

- Realizing ICC — demonstrate suitability of ICC in automatic program analysis. This is done by adjusting and extending a theoretical system—the *mwp*-flow analysis (cf. subsection 2.2)—to make it suitable for concrete implementation.
- Certifying complexity — obtain formally verified complexity analysis by mechanically formalizing the *mwp*-flow analysis computational technique, using an interactive theorem prover, i.e., the Coq proof assistant.

- Extended applications — assert that ICC can extend to other use cases beyond read-only analysis, and is particularly suited for compiler integration with intermediate representation. This is done by introducing applications e.g., in program optimization.

The details of each goal are discussed in section 3. The collective unifying implication of these goals is that they evaluate the robustness of ICC for practical applications, and address open challenges in the area of automatic complexity analysis. Note that the intent of this work is *not* to produce the most elaborate static analyzers; rather the main idea is to evaluate suitability and advantages of ICC to address outstanding challenges in program analysis. Positive results serve to advance future versions of concrete tools, but such enhancements are outside the scope of this work.

2 Background and Significance

This section gives an overview and sufficient background of the relevant topics impacting the proposed research. It introduces Implicit Computational Complexity and *mwp*-flow analysis; discusses the results in automatic complexity analysis and certifying complexity, and comments on the significance of this proposal. If the general notions of these topics are familiar, a deep reading of this section is not required to follow the remaining sections.

2.1 Implicit Computational Complexity

Implicit Computational Complexity [7] is a subfield in computational complexity theory. Unlike the traditional approach, is not concerned with a specific computational model; rather it considers *implicit* characterizations of complexity classes by placing various syntactic criteria on a program, which in turn guarantees some semantic properties about it, usually resource usage bounds. Numerous distinct program analysis systems exist, mainly revolving around type checking, data flow, or performing checks on computed values [14]. An interesting general description of ICC has been given [15] as follows: let L be a programming language, C a complexity class, and $[\![p]\!]$ the function computed by program p. Then the task is to find a restriction $R \subseteq L$, such that the following equality holds: $\{[\![p]\!] \mid p \in R\} = C$. The variables L, C, and R are the parameters that vary greatly between different ICC systems.

The ICC flavor of program analysis offers several benefits. It drives better understanding of complexity classes, introduces new and often orthogonal analysis techniques, and provides a prospective avenue for realizing complexity analyses. By embedding a restriction in a program, by which a complexity bound will be guaranteed, the effort of satisfying that restriction is raised upstream to the programmer. If the restriction is maintained, then the bound will be maintained. This implicit character leads to a natural approach to complexity analysis: developer can focus on writing a program and concurrently obtain guarantees of its behavior. This works assuming that the restriction is not excessively limiting in expressing algorithms in practice [14].

2.2 Overview of *mwp*-Flow Analysis

The *mwp*-flow analysis [16] is one example of an ICC-based system. It is a sound computational method that certifies polynomial bounds on the size of the values manipulated by an imperative program. It provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space and, if a program terminates, it will do so in polynomial time in the size of its inputs. The analysis computes, for each program variable, a vector tracking how it depends on other variables. The vector values are determined by applying the nondeterministic rules of the *mwp*-calculus to the commands of the program. Those vectors are collected in a matrix. A program is assigned a matrix only if all the values in it are bounded by a polynomial in the input sizes. This technique is compositional, abstracts away e.g., iteration bounds, and operates on a simple, imperative memory-less language.

2.3 Automatic Complexity Analysis

The most advanced automatic complexity analyzers are represented by tools such as AProVE [2], ComplexityParser [3], and RaML [4]. They perform complexity analysis based on different approaches: integer term rewrite systems, tier-based typing, and cost amortization, respectively. They offer varying levels of maturity and feature support and target different programming languages and paradigms. The annual Termination and Complexity Competition (TermCOMP), that focuses on automated termination and complexity analysis for various kinds of programming paradigms [17], offers interesting insight of the performance of the various tools that choose to participate[1]. Although current solutions are rich in functionality, challenges remain relating to their choices in methodologies (lack scalability, absence of compositionality [5]); user interaction (level of user control, interpretability of results [4, 5]); and absence of formal verification to ensure correctness of their implementations [5]. These impact the analyzer applicability and performance and reduce their usability.

Practical applications that derive from ICC-based approaches appear primarily in two categories: standalone static analyzers and compiler pass implementations. The earliest known implementation appeared in 2008 by Avanzini et al. [8] in ICCT: a command-line tool for automatic analysis of polynomial time computable functions through term rewriting. In later works, Avanzini and Dal Lago [9] introduced a complexity analysis methodology for higher-order functional programs, and demonstrated this method in a tool called HoSa. Avanzini et al. [10] then worked on analyzing complexity of probabilistic programs, resulting in tool called ecoimp. A separate series of works by Moyen et al., push in the direction of implementing ICC techniques as compiler passes, integrating with the LLVM compiler intermediate representation. First work introduced an implementation of Non-Size Increasing (NSI) program analysis, which helps detect memory leaks by tracking memory allocation

[1]Results of 2021 competition: `https://termcomp.github.io/Y2021/`

and deallocations [12]. A second result applied ICC-inspired dependency graph analysis to detect loop quasi-invariants to optimize the complexity of generated code [13].

2.4 Formally Verified Complexity

A few prior results exist that combine formalization of complexity with the Coq proof assistant. They range from practical analyses to proofs in computational complexity theory. For practical application, Coq has been used to verify stack bounds for assembly code [18] and to obtain WCET loop-bound estimation [19]. Carbonneaux et al. [6] presented an automatic static analyzer for imperative programs, and although the analyzer itself is not verified, it generates bounds with machine-checkable certificates, to guarantee that the computed bound holds. For functional paradigm, McCarthy et al. [20] developed a Coq library, with a monad that counts abstract steps, which enabled running time analysis of programs written using the monad. An ICC-based characterization was introduced by Férée et al. [21], in the form of a Coq library, that allows for readily proving that a function is computable in polynomial time. For results in computational complexity theory, Ciaffaglione [22] proved the undecidability of the halting problem. Guéneau et al. [23] formalized the $\mathcal{O}$ notation. Forster et al. [24] implemented a multi-tape to single-tape compiler and introduced the first formalized universal Turing Machine verified w.r.t. time and space complexity for any model of computation, in any proof assistant. More recently, Gäher and Kunze formalized the Cook-Levin theorem in Coq [25]. Despite these advances, formalization of complexity is in early stages and basic complexity-theoretic results e.g., time and space hierarchy theorems remain unavailable.

2.5 Significance

This research proposal intersects computer science theory and application, with the intent of using those theoretical approaches to solve existing challenges in automatic program analysis. In 2017, Jean-Yves Moyen—a notable researcher in the ICC community, whose career spans 3 decades of advancements—remarked enthusiastically, that after twenty years and many results, ICC was ready to move into real-world and to concrete applications [14, p. 7]. And although a few early results exist, as noted earlier in this section, progress in this direction is still at early stages. With similar aspirations, this proposal hopes to move further in that direction. It is conceivable that certified complexity would be of significant interest to multiple research communities. The next section will detail the specifics of the methodology, that consists of multiple projects. Assuming the successful completion of each project, they would show that certifiably correct complexity analysis is achievable, and that ICC techniques can be used to obtain efficient and practical complexity analysis. These results would extend current capabilities in automatic complexity analysis, and move those techniques closer to becoming a standard in real-world software verification.

3 Research Design and Methods

The proposed research naturally divides into three directions along its specified aims. Each has separate goals, timeline, and expected results. This section discusses the details of those research directions, provides motivations for their choice approaches, and specifies their obtained or expected results. Also note that these directions are mutually exclusive: they can be completed in parallel, and success (or failure) in one branch does not impact the rest; but when considered collectively, they support the main hypothesis of this proposal.

3.1 Realizing ICC: Obtaining Automatic Analysis

The foundation of this work is the theoretical technique of *mwp*-flow analysis [16], also introduced in subsection 2.2. While it offers an elegant theoretical computational method for resource bounds analysis of simple imperative programs, its use for practical purposes is not straightforward. Due to its nondeterminism, it is generally not feasible to determine if a matrix corresponding to an input program exists. The pen-and-paper strategy also does not consider the difficulty in handling failure that is a critical consideration for concrete applications. Furthermore, the original authors are unsure of the richness and powerfulness of the method, i.e., whether its syntax can be extended to richer syntax, and how useful is this method in analyzing larger classes of programs, beyond the toy examples presented in the paper. As presented, this analysis offers a suitable investigation target, to measure if this ICC-inspired technique can be applied to obtain automatic analysis.

Methodology. The intent of this direction of work is to answer the questions left open by the original authors, and to resolve the outstanding challenges that make this technique difficult to implement. This is done by modify the underlying mathematical framework, to make the system deterministic and capable of handling failure. This is achieved by extending the original technique and its supported syntax; redefining its derivation rules, handling failures internally through choice semi-ring, and generating efficient data representations and algorithms to process the input program and to compute the result. The system is extended with support for function calls including recursion, which are important elements to achieve compositional analysis. The enhanced system is fully automatic since input program need not be annotated to perform the analysis. A standalone static analyzer, pymwp [26], is implemented to demonstrate the enhanced and extended technique.

Results. This work has effectively been completed. The results demonstrate that the goals of this work are achievable, and signal positive feedback to strengthen the hypothesis of this research proposal. The extended theoretical method, describing appropriate system adjustments to obtain practical analysis, was published as a research paper at 7th International Conference on Formal Structures for Computation and Deduction (FSCD'22) [11]. A second publication, that demonstrates the static analyzer tool, has been submitted to the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'23), as a tool demonstration paper, for possible publication in 2023 [27].

3.2 Certifying Complexity: Provably Correct Resource Bounds

Formally verified development toolchains, with machine-assisted mathematical proofs, provide the highest assurance of the correctness of the underlying system. They are especially important in production of critical software systems to assure correctness in, e.g., transportation, energy and telecommunications. In such domains, the need to rigorously confirm that a program meets their specification justifies the high effort required by formal verification. Although results are scarce in complexity analysis (cf. subsection 2.4), encouraging results exist in related areas. The CompCert verified `C` compiler [28] established the foundation for provably correct realistic compilation, as the compiler guarantees preservation of program semantics. Subsequent work by Jourdan et al. [29], using abstract interpretation to analyze runtime errors, proved that formally verified static analysis is achievable. The motivation for this work is similar: by leveraging ICC, the goal is to obtain formally verified complexity analysis that guarantees the correctness of the obtained analysis result.

Methodology. This work reuses the *mwp*-flow analysis as a target for mechanical formalization using the Coq proof assistant. More specifically, the goal is to formalize the computational method itself, as presented in the original paper [16]. The main result of the paper is the soundness theorem, that states for a program (command) `C` and matrix M, $\vdash \texttt{C} : M$ implies $\vDash \texttt{C} : M$, which means the calculus assigns the matrix M to the program `C`, and further, a command is *derivable* if the calculus assigns at least one matrix to the command. This relation holds iff every value computed by `C` is bounded by a polynomial in the inputs. The goal of this work is to prove this soundness theorem, and its other related theorems, as presented in the original paper. Based on review of related works in this area (cf. subsection 2.4), this is the first known attempt to formalize a complexity analysis technique in Coq.

The motivation for choosing the *mwp*-flow analysis as the target of formalization is based on multiple factors. To start, its underlying mathematical structure relies on inference rules and linear algebra, which seem highly suitable for formalization. The mathematical components [30] Coq library comes with built-in tactics to prove operations on matrices. The imperative programming language under analysis also seems suitable, because it is similar to the language used in Coq literature to prove properties of programs [31, Imp]. Since the analysis technique was used in the earlier work, it is already deeply understood and familiar to the author. Lastly, assuming the formalization succeeds, it then lends itself to further formal development, such as a formally verified complexity analyzer. The choice of using Coq proof assistant is motivated by its use in the earlier related formalizations, its industry-grade development stage, and readily available literature and support. The challenge of using Coq includes determining the appropriate tools to apply and working around the foundational logical assumptions of Coq. These include e.g., constructivist logic and the *strong normalization* property, i.e., computation must always terminate [32, p. 18].

Expected results. The success of this work greatly depends on the ability to complete the Coq formalization. Subsequent directions can then be explored in formalizing a static

analyzer based on this technique. The result of the formalization itself is likely sufficiently interesting for conference publication at venues that have accepted similar results in the past (cf. subsection 2.4). A preliminary presentation of this work has been accepted and will occur in January 2023 at the CoqPL workshop [33].

3.3 Extended Applications: Transforming and Optimizing Programs

Correctly analyzing a "real world" programming language is a difficult task because of the large number of allowed constructions [14, p. 15]. Compilers represent a natural target for introducing practical and automatic analyses, because they already perform the task of mapping rich languages to more simplified intermediate representations, while still maintaining sufficient information about program structure. Compilers already include optimization techniques, that can either benefit from, or be reused by, choice program analysis techniques. Further, compilers make integrated analyses applicable for large classes of programs, since any language accepted by the compiler front-end can potentially be analyzed. These motivations provided the underlying drive for earlier work by Moyen et al. [13], showing that performing ICC-based analysis in compilers is not only achievable, but can also offer new techniques for optimizing program's resource usage during compilation. This seminal paper was the first known application of ICC in a mainstream compiler (LLVM) and serves as inspiration for the direction of work presented next. Although the early result was encouraging, subsequent pushes in this direction have not occurred. The goal of this work is to explore the extended applications of ICC, and to establish its other practical uses, particularly in connection with compilation.

Methodology. Using an ICC-inspired dependency analysis technique, a program transformation technique is developed to detect independence between loop body statements, and to split those statements into separate loops. This transformation is generally known as "loop fission", and a fissioned program can then be parallelized, to obtain performance gain on multicore architectures. Although similar program transformations are generally achievable by other methods, the technique developed here is distinct in its ability to analyze and optimize loops with unknown iteration spaces. This work outlines the details of this transformation technique and includes the development of a set of benchmarks to evaluate its anticipated gain in terms of runtime performance. Further work in this direction is open to exploration, where other related applications, or extensions of this technique, can be investigated, possibly by developing compiler passes.

Results. This direction of work, i.e., the approach related to loop optimization, has resulted in one accepted conference publication and will appear at the 24th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'23) [34].

3.4 Tentative Timeline

3.4.1 Preliminary and Investigative Work

Start	End	Description
2021 Jan	2022 Apr	Investigative work on implementing *mwp*-flow analysis
2021 Oct	2022 May	Investigative work on parallelizing loops
2022 Jul	2022 Dec	Study logical foundations in Coq
2022 Aug	2022 Sep	Benchmark and artifact for parallelization of loops
2022 Sep	2022 Oct	Prepare pymwp artifact and paper submission
2022 Sep	∼12 mo.	Formalization of *mwp*-flow analysis, including: specifying analyzed language, mathematical structures and type system; proving all lemmas and theorems
2023 Jan	∼2 mo.	Study mathematical components needed for *mwp* formalization
2023 Jan	∼4 mo.	Study PL foundations in Coq

3.4.2 Schedule of Presentations

Date	Title (At)
2022 Apr 1	Semantic-preserving optimization algorithm for automatic program parallelization (AU Graduate Research Day)
2022 Jun 20	Realizing Implicit Computational Complexity (TYPES'22)
2022 Aug 4	*mwp*-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity (FSCD'22)
2022 Dec 6	Formally Verified Resource Bounds Through Implicit Computational Complexity (SPLASH'22 Doctoral Symposium)
2023 Jan 16/7	Distributing and Parallelizing Non-canonical Loops (VMCAI'23)
2023 Jan 21	Certifying Complexity Analysis (CoqPL'23)

4 Previous Work Done in this or Related Fields

This section lists aspects demonstrating author's preparedness for completing this proposal.

- Research experience — author has worked with a research team on ICC-inspired topics for past 2 years, which has resulted in the publications listed in section 5.
- Computer science foundations — author has completed undergraduate and master's degrees in computer science. These have included wide-range exposure to different CS topics and provides ability to leverage those concepts in completion of this project.
- Mathematical foundations — author has more limited exposure in these areas but is confident all gaps can be resolved dynamically.

- Programming experience — author is highly technically proficient, with e.g., several years of industry experience and contributions to open source community. These skills will facilitate timely completion of all programming aspects of the project.
- Proof assistants — author has independently completed, or is completing, training in use of Coq proof assistant. This is done by studying the Software Foundations-series of books, volumes "Logical Foundations" [31] and "Programming Language Foundations" [35]. These provide the necessary skills for proving properties of programs.

5 Personal Publications

As commonly done in theoretical computer science, authors are listed in alphabetical order.

Published and Submitted Papers

1. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity". In: *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022*. Vol. 228. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 26:1–26:23. DOI: 10.4230/LIPIcs.FSCD.2022.26.
2. Neea Rusch. "Formally Verified Resource Bounds Through Implicit Computational Complexity". In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. Association for Computing Machinery, 2022. DOI: 10.1145/3563768.3565545.
3. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Distributing and Parallelizing Non-canonical Loops". To appear in Verification, Model Checking, and Abstract Interpretation (VMCAI). 2023.
4. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs". Submitted to 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2023.

Peer-Reviewed Papers Without Published Proceedings

5. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Realizing Implicit Computational Complexity". At the 28th International Conference on Types for Proofs and Programs (TYPES). 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_14.pdf.
6. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Certifying Complexity Analysis". At the Ninth International Workshop on Coq for Programming Languages (CoqPL). 2023.

Artifacts

7. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. *Distributing and Parallelizing Non-canonical Loops – Artifact*. Version 1.0. Sept. 2022. DOI: 10.5281/zenodo.7080145. URL: https://github.com/statycc/loop-fission.
8. Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. *pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs*. Version 1.0. Oct. 2022. DOI: 10.5281/zenodo.7159134. URL: https://github.com/statycc/pymwp.

6 References

[1] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012. ISBN: 9781133187790.

[2] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. "Analyzing program termination and complexity automatically with AProVE". In: *Journal of Automated Reasoning* 58.1 (Jan. 2017), pp. 3–31. DOI: 10.1007/s10817-016-9388-y.

[3] Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. "Complexity-parser: An automatic tool for certifying poly-time complexity of java programs". In: *Theoretical Aspects of Computing – ICTAC 2021*. Vol. 12819. LNCS. Springer, 2021, pp. 357–365. DOI: 10.1007/978-3-030-85315-0_20.

[4] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Resource Aware ML". In: *Computer Aided Verification - 24th International Conference, CAV 2012*. Vol. 7358. LNCS. Springer, 2012, pp. 781–786. DOI: 10.1007/978-3-642-31424-7_64.

[5] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. "Compositional Certified Resource Bounds". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Association for Computing Machinery, 2015, pp. 467–478. DOI: 10.1145/2737924.2737955.

[6] Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. "Automated Resource Analysis with Coq Proof Objects". In: *Computer Aided Verification*. Springer, 2017, pp. 64–85. DOI: 10.1007/978-3-319-63390-9_4.

[7] Ugo Dal Lago. "A Short Introduction to Implicit Computational Complexity". In: *ESSLLI*. Vol. 7388. LNCS. Springer, 2011, pp. 89–109. DOI: 10.1007/978-3-642-31485-8_3.

[8] Martin Avanzini, Georg Moser, and Andreas Schnabl. "Automated Implicit Computational Complexity Analysis (System Description)". In: *Proceedings of the 4th International Joint Conference on Automated Reasoning*. IJCAR '08. Springer-Verlag, 2008, pp. 132–138. DOI: 10.1007/978-3-540-71070-7_10.

[9] Martin Avanzini and Ugo Dal Lago. "Automating Sized-Type Inference for Complexity Analysis". In: *Proceedings of the ACM on Programming Languages* 1.ICFP (Sept. 2017), 43:1–43:29. DOI: 10.1145/3110287.

[10] Martin Avanzini, Georg Moser, and Michael Schaper. "A Modular Cost Analysis for Probabilistic Programs". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020), 172:1–172:30. DOI: 10.1145/3428240.

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

[12] Jean-Yves Moyen and Thomas Rubiano. "Detection of Non-Size Increasing Programs in Compilers". At Developments in Implicit Computational Complexity (DICE 2016). 2016. URL: https://lipn.univ-paris13.fr/DICE2016/Abstracts/paper_2.pdf.

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

[14] Jean-Yves Moyen. *Implicit Complexity in Theory and Practice*. Habilitation à Diriger des Recherches (HDR). 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

[15] Romain Péchoux. *Complexité implicite : bilan et perspectives*. Habilitation à Diriger des Recherches (HDR). 2020. URL: https://hal.univ-lorraine.fr/tel-02978986.

[16] Neil D. Jones and Lars Kristiansen. "A flow calculus of *mwp*-bounds for complexity analysis". In: *ACM Transactions on Computational Logic* 10.4 (Aug. 2009), 28:1–28:41. DOI: 10.1145/1555746.1555752.

[17] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. "The termination and complexity competition". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[18] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. "End-to-End Verification of Stack-Space Bounds for C Programs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Association for Computing Machinery, 2014, pp. 270–281. DOI: 10.1145/2594291.2594301.

[19] Sandrine Blazy, André Maroneze, and David Pichardie. "Formal Verification of Loop Bound Estimation for WCET Analysis". In: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013*. Vol. 8164. LNCS. Springer, 2013, pp. 281–303. DOI: 10.1007/978-3-642-54108-7_15.

[20] Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. "A Coq library for internal verification of running-times". In: *Science of Computer Programming* 164 (Oct. 2018), pp. 49–65. DOI: 10.1016/j.scico.2017.05.001.

[21] Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. "Formal proof of polynomial-time complexity with quasi-interpretations". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Association for Computing Machinery, 2018, pp. 146–157. DOI: 10.1145/3167097.

[22] Alberto Ciaffaglione. "Towards Turing computability via coinduction". In: *Science of Computer Programming* 126 (Sept. 2016), pp. 31–51. DOI: 10.1016/j.scico.2016.02.004.

[23] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018*. Vol. 10801. LNCS. Springer, 2018, pp. 533–560. DOI: 10.1007/978-3-319-89884-1_19.

[24] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. "Verified programming of Turing machines in Coq". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. Association for Computing Machinery, 2020, pp. 114–128. DOI: 10.1145/3372885.3373816.

[25] Lennard Gäher and Fabian Kunze. "Mechanising Complexity Theory: The Cook-Levin Theorem in Coq". In: *12th International Conference on Interactive Theorem Proving, ITP 2021*. Vol. 193. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021, 20:1–20:18. DOI: 10.4230/LIPIcs.ITP.2021.20.

[26] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. *pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs*. Version 1.0. Oct. 2022. DOI: 10.5281/zenodo.7159134. URL: https://github.com/statycc/pymwp.

[27] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs". Submitted to 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2023.

[28] Xavier Leroy. "Formal verification of a realistic compiler". In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[29] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A formally-verified C static analyzer". In: *ACM SIGPLAN Notices* 50.1 (Jan. 2015), pp. 247–259. DOI: 10.1145/2775051.2676966.

[30] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2022. DOI: 10.5281/zenodo.7118596.

[31] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. "Logical Foundations". In: *Software Foundations*. Ed. by Benjamin C. Pierce. Version 6.2. Vol. 1. 2022. URL: http://softwarefoundations.cis.upenn.edu.

[32] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2004. ISBN: 9783540208549.

[33] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Certifying Complexity Analysis". At the Ninth International Workshop on Coq for Programming Languages (CoqPL). 2023.

[34] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Distributing and Parallelizing Non-canonical Loops". To appear in Verification, Model Checking, and Abstract Interpretation (VMCAI). 2023.

[35] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. "Programming Language Foundations". In: *Software Foundations*. Ed. by Benjamin C. Pierce. Version 6.2. Vol. 2. 2022. URL: http://softwarefoundations.cis.upenn.edu.

7 Human Subjects/Vertebrate Animals

Not applicable.