Certifying Complexity Analysis

Clément Aubert
Augusta University
Augusta, USA
aubert@math.cnrs.fr

Thomas Rubiano
LIPN—UMR 7030 Université Sorbonne Paris Nord
Paris, France
rubiano.thomas@gmail.com

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

Thomas Seiller
LIPN—UMR 7030 Université Sorbonne Paris Nord
CNRS
Paris, France
seiller@lipn.fr

Abstract

This work drafts a strategy that leverages the field of Implicit Computational Complexity to certify resource usage in imperative programs. This original approach sidesteps some of the most common–and difficult–obstacles "traditional" complexity theory face when implemented in Coq.

CCS Concepts: • **Theory of computation → Program verification**; **Complexity theory and logic**.

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

1 Motivation

The ability to statically infer resource bounds of programs offers numerous benefits, e.g., to insure safe memory usage. Even more preferable if those guarantees are established with the rigor of formal verification, because that increases confidence in the obtained analysis result and enables integration of complexity analyses into larger formal developments.

Unfortunately, computational complexity is notoriously difficult to represent formally for several reasons. In general, deriving a complexity bound for an arbitrary program is an undecidable problem. In the area of complexity theory, "formalisations of even basic complexity-theoretic results are not available" [11, p. 114], hindering certification attempts.

For practical complexity analyses, many existing techniques present methodological challenges if they require e.g., program termination or inlining functions [6]. Therefore, a realistic pathway toward formal certification of a program's resource usage is narrow. A few encouraging early results exist, and we discuss some of those in Sect. 3. In this proposal we will sketch how a different approach, founded on Implicit Computational Complexity, could sidestep some of the usual difficulties in implementing and verifying complexity analyses in Coq.

The field of Implicit Computational Complexity (ICC) [10] drives better understanding of complexity classes, but it also guides the development of resources-aware languages and static source code analyzers. The core idea is to bound resources *while the program is being written (or type checked)* instead of measuring its resource usage afterwards on an abstract model of computation. This can be done through e.g., bounded recursion or using typing mechanisms. The goal is to find a syntactical restriction or a type system such that a program can be written or typed only if it belongs to a particular complexity class. ICC-based systems are often compositional and they offer more natural tools to write programs than theoretical models of computation used in complexity theory. We speculate these combined properties could make ICC-approaches a conceivable pathway toward certified complexity and sketch a more detailed plan below.

2 Preliminary Action Plan

We plan to formalize in Coq an ICC-based complexity analysis technique, the *mwp-flow analysis* [15][1]. We chose this method because its internal mechanics has been recently studied [1], and by our assessment, it seems suitable for formalization in Coq. As for Coq, it seems like the ideal target language because of its existing libraries and preliminary work–some of which are discussed in Sect. 3–, most notably related to compilers [16].

2.1 Overview of *mwp*-Flow Analysis

The *mwp*-flow analysis certifies polynomial bounds on the size of the values manipulated by an imperative program. While it does not ensure (or require) program termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that if it 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 nondeterminitic rules of the sound *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 inputs sizes. This technique is compositional, abstracts away e.g., iteration bounds, and operates on a memory-less

[1]Where mwp stands for *m*aximum, *w*eak polynomial and *p*olynomial, representing increasing growth rates of variables values.

imperative language, reminiscent of the `Imp` language from Software Foundations [19].

2.2 The Coq Formalization

Our goal is to certify the analysis as presented in the original paper [15]. Note that this does not mean that the bound is certified, but that *the mechanism to compute those bounds* is certified. Of course, this implies the correctness of the bounds as a by-product but constitutes a major difference w.r.t. the results discussed in Sect. 3. Preliminary explorations have led us to establish the following milestones.

The mathematical foundations Our first goal is to define the mathematical structure required to carry out the rest of the construction. This requires defining vectors, matrices and their operations, semi-rings, and honest polynomials[2] that are needed to represent the *mwp*-bounds. The Mathematical Components library [20] will lay the foundations for the linear algebra representations, but likely requires extensions to accommodate our specific analysis.

Implementing the language The analyzed language is a simple imperative language that manipulates natural numbers, held in a fixed number of program variables. Its syntax includes variables, expressions (operations + and ×), Boolean expressions, and commands (e.g., assignment, loop and decision statements, command sequences, and skip), with their usual semantics. We expect implementing it and its small-steps semantics in Coq to be relatively simple, following the examples from Software Foundations [18, 19].

Implementing the typing system Even if it can be computationally expensive to run an automatic inference [2], the typing system *in itself* is relatively simple. It contains only 10 rules, essentially one for each type of command, and except for the initial assignment of vectors to variables, is fully deterministic. We conjecture that standard methods [7, 8] to implement simple type systems will be enough, but will require some care to scale to the matrix-as-type paradigm of this analysis.

Certifying the analysis This will be the most demanding part of our plan. The original paper contains all the required handwritten proofs, but the authors caution that "[t]hese proofs are long, technical and occasionally highly nontrivial" [15, p. 2]. The main result of the paper is the soundness proof of the analysis [15, Theorem 5.3], i.e., the proof of the existence of a matrix typing the program implies the existence of an honest polynomial bounding the variables' growth rates. The main result follows from 15 pages of proofs presented in section 7 of the paper. This section revolves around proving the soundness properties of the calculus, and we expect the most substantial effort to be spent on formalizing these proofs. Some of them are quite intricate but with a satisfactory level of detail. The cases concerning soundness of loops are the most difficult on paper, but their inductive nature *should* (we hope!) be processed by Coq rather easily.

We leave for future work the possibility of creating a formally verified, automatic static analyzer founded on the proof of correctness of this method: as we discussed in other works [1, 2], care is required to implement a typing strategy that does not rapidly become intractable.

3 Related Work

A few prior results exist that combine formalization of complexity and Coq. 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 [4] and to obtain WCET loop-bound estimation [3]. Carbonneaux et al. [5] 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. [17] 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. [12], in the form of a Coq library, that allows for readily proving that a function is computable in polynomial time.

Coq has also been used to formalize some of the foundations of modern complexity theory. Ciaffaglione [9] proved the undecidability of the halting problem. Guéneau et al. [14] formalize the O notation. Forster et al. [11] 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 [13]. Despite these advances, formalization of complexity is in early stages and basic complexity-theoretic results e.g., time and space hierarchy theorems, remain unavailable.

Our proposed project differs from these earlier results primarily in its intent. We plan to formalize the complexity analysis mechanism itself—not its computed result, as was done previously. In their work with the Turing Machines, Forster et al. [11] were explicit in emphasizing the challenge they experienced in formalizing complexity. We hypothesize that our ICC-based approach, with e.g., its built-in abstractions, will help reduce this challenge. It is our hope that CoqPL will welcome our proposal for a certified complexity analysis in Coq, and will be keen on indicating any library, tool or resource that could help.

[2] Which are "polynomial build up from constants in $\mathbb{N}$ and variables by applying the operations + (addition) and × (multiplication)." [15, p. 5]

Acknowledgments

The authors wish to thank Delphine Demange for the interesting discussion she had with Neea, and the reviewers for their careful reading and many interesting comments. This research is supported by the Th. Jefferson Fund of the Embassy of France in the United States and the FACE Foundation, and has benefited from the research meeting 21453 "Static Analyses of Program Flows: Types and Certificate for Complexity" in Schloss Dagstuhl. Th. Rubiano and Th. Seiller are supported by the Île-de-France region through the DIM RFSI project "CoHOp". N. Rusch is supported in part by the Augusta University Provost's office, and the Translational Research Program of the Department of Medicine, Medical College of Georgia at Augusta University.

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) (Leibniz International Proceedings in Informatics, Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 26:1–26:23. https://doi.org/10.4230/LIPIcs.FSCD.2022.26

[2] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. 2022. pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs. (Oct. 2022). In preparation.

[3] Sandrine Blazy, André Maroneze, and David Pichardie. 2013. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8164)*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer, 281–303. https://doi.org/10.1007/978-3-642-54108-7_15

[4] Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 270–281. https://doi.org/10.1145/2594291.2594301

[5] Quentin Carbonneaux, Jan Hoffmann, Thomas W. Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 64–85. https://doi.org/10.1007/978-3-319-63390-9_4

[6] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). Association for Computing Machinery, 467–478. https://doi.org/10.1145/2737924.2737955

[7] Adam Chlipala. 2010. An Introduction to Programming and Proving with Dependent Types in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 1–93. https://doi.org/10.6092/issn.1972-5787/1978

[8] Adam Chlipala. 2022. *Formal Reasoning About Programs*. The MIT Press. http://adam.chlipala.net/frap/

[9] Alberto Ciaffaglione. 2016. Towards Turing computability via coinduction. *Science of Computer Programming* 126 (2016), 31–51. https://doi.org/10.1016/j.scico.2016.02.004

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

[11] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2020. Verified programming of Turing machines in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 114–128. https://doi.org/10.1145/3372885.3373816

[12] Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. 2018. Formal proof of polynomial-time complexity with quasi-interpretations. In *SIGPLAN*, June Andronick and Amy P. Felty (Eds.). Association for Computing Machinery, 146–157. https://doi.org/10.1145/3167097

[13] Lennard Gäher and Fabian Kunze. 2021. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (Leibniz International Proceedings in Informatics, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 20:1–20:18. https://doi.org/10.4230/LIPIcs.ITP.2021.20

[14] Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

[15] Neil D. Jones and Lars Kristiansen. 2009. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Transactions on Computational Logic* 10, 4 (2009), 28:1–28:41. https://doi.org/10.1145/1555746.1555752

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

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

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

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

[20] Mathematical Components team. 2022. Mathematical Components. https://math-comp.github.io