Realizing Implicit Computational Complexity*

Clément Aubert[1], Thomas Rubiano[2], Neea Rusch[1], and Thomas Seiller[2,3]

[1] School of Computer and Cyber Sciences, Augusta University, USA
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord, France
[3] CNRS, France

Originalities in Implicit Computational Complexity. Automatic performance analysis and optimization is a critical for systems with resource constraints. The field of Implicit Computational Complexity (ICC) [12] pioneers in embedding in the program itself a guarantee of its resource usage, using e.g. bounded recursion [10, 18] or type systems [8, 17]. It drives better understanding of complexity classes, but also introduces original methods to develop resources-aware languages, static source code analyzers and optimizations techniques, often relying on informative and subtle type systems. Among the methods developed, the *mwp-flow analysis* [16] certifies polynomial bounds on the size of the values manipulated by an imperative program, obtained by bounding the transitions between states instead of focusing on states in isolation, and is not concerned with termination or tight bounds on values. It introduces a new way of tracking dependencies between "chunks" of code by typing each statement with a matrix listing the way variables relate to each others.

Having introduced such novel analysis techniques, and, as opposed to traditional complexity, by utilizing models that are generally expressive enough to write down actual algorithms [20, p. 11], ICC provides a conceivable pathway to automatable complexity analysis and optimization. However, the approaches have rarely materialized into concrete programming languages or program analyzers extending beyond toy languages, with a few exceptions [7, 15]. Absence of realized results reduces ability to test the true power of these techniques, limits their application in general, and understanding their capabilities and potential expressivity remains underexplored.

We present an ongoing effort to address this deficiency by applying the mwp-flow analysis, that tracks dependencies between variables, in three different ways, at different stages of maturation, in their temporal order. The first and third projects bend this typing discipline to gain finer-grained view on statements independence, to optimize loops by hoisting invariant [21] and by splitting loops "horizontally" to parallelize them [5]. The second project refines, extends and implements the original analysis to obtain a fast, modular static analyzer [6]. All three projects aim at pushing the original type system to its limits, to assess how ICC can in practice lead to original, sometimes orthogonal, approaches. We also discuss our intent and motivations behind formalizing this analysis using Coq proof assistant [22], in a spearheading endeavour toward formalizing complexity analysis.

1. Loop Quasi-Invariant Chunk Detection. Loop peeling for hoisting (quasi-)invariants can be used to optimize generated code [1, p. 641], and is implemented e.g. in LLVM as the `licm` pass. This work [21] leverages an ICC-inspired dependency analysis to provide a transformation method to compilers. It enables detection of quasi-invariants of arbitrary degree in composed statements called "chunks". It reuses the mwp's matricial system and typed data flows to generate dependency graphs, to compute an invariance degree for each statement or chunks of

*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".

statements. It then finds the maximum (worst) dependency graph for loops, and recognizes whether an entire block is quasi-invariant. If this block is an inner loop, it can be hoisted, and the computational complexity of the overall program can be decreased. A prototype analysis pass [3] has been designed, proven correct and successfully implemented using a toy `C` parser, and as a prototype pass for the LLVM. This is the first known application of introducing ICC techniques in mainstream compilers.

2. Improved and Implemented mwp-Analysis. In an ongoing development, we improved and implemented the mwp-bounds analysis [16], which certifies that the values computed by an imperative program are bounded by polynomials in the program's input, represented in a matrix of typed flows, characterizing controls from one variable to another. While this flow analysis is elegant and sound, it is also computationally costly–it manipulates non-deterministically a potentially exponential number of matrices in the size of the program [6, 2.3]—and missed an opportunity to leverage its built-in compositionality. We addressed both issues by expanding the original flow calculus, and adjusting its internal machinery to enable tractable analysis [6], and further extended the theory with analysis of function definitions and calls—including recursive ones, a feature not widely supported [14, p. 359]. Our effort and theoretical development is realized in an open-source tool `pymwp` [4], capable of automatically analyzing complexity of programs written in a subset of the `C` programming language.

3. Splitting Loops Horizontally to Improve Their Parallel Treatment. Our most recent effort is directed toward program optimization through loop parallelization. Using an ICC-inspired data flow-based variable dependency analysis, we can reproduce the *tour de force* of detecting opportunities for loop fission that have been missed by other standard analyses [21]. In particular, the dependency analysis allows optimizing loops by splitting them "horizontally", e.g. from `for (int i = 1; i < 10; i++){a[i] = a[i-1] + i; b[i] = b[i-1] + i;}` to:

```
for (int i = 1; i < 10; i++){a[i] = a[i-1] + i;}
for (int i = 1; i < 10; i++){b[i] = b[i-1] + i;}
```

Our approach can process loop-carried dependencies [11, 3.5.2]–such as the one illustrated above–and optimizes `while` loops [5, Sect. 5].—and, more generally, loops whose trip-count cannot be known at compilation time—that are completely ignored by OpenMP [11, 3.2.2], and generally present great difficulty and often prevent optimization [5, Sect. A]. Combined with OpenMP `pragma` directives, this approach gives a speed-up "as good as" `AutoPar-Clava`—which "compare[s] favorably with closely related auto-parallelization compilers" [2, p. 1]—when both are applicable that can be integrated in automatic parallelization pipelines [5, Sect. B]. Our benchmark, shared at https://github.com/statycc/icc-fission, substantiate experimentally those claims and provides further evidence.

...and Pushing Even Further. From there, many other directions can be explored. Since ICC techniques tend to be designed for simpler program syntax, compiler intermediate representations present an ideal location and point of integration for performing analyses. Implementing the analysis in certified tools such as the CompCert compiler [19] (or, more precisely, its static single assignment version [9]) naturally necessitates certifying the complexity analysis, and we plan to pursue this effort using the Coq proof assistant [22]. The plasticity of both compilers and of the implemented analysis should facilitate porting our results to support further programming languages in addition to `C`. As complexity analysis is difficult in Coq [13], we believe a push would be welcome, and that ICC provides the necessary tools for it.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Hamid Arabnejad, João Bispo, João M. P. Cardoso, and Jorge G. Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of C applications. *J. Supercomput.*, 76(9):6753–6785, Sep 2020. doi:10.1007/s11227-019-03109-9.

[3] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Lqicm on c toy parser. URL: https://github.com/statycc/LQICM_On_C_Toy_Parser.

[4] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. pymwp: MWP analysis in Python. URL: https://github.com/statycc/pymwp/.

[5] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. A Novel Loop Fission Technique Inspired by Implicit Computational Complexity. Submitted to ATVA 2022, May 2022. URL: https://hal.archives-ouvertes.fr/hal-03669387.

[6] Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. In Amy Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD)*, LIPIcs. Schloss Dagstuhl, March 2022. To appear. URL: https://hal.archives-ouvertes.fr/hal-03596285.

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

[8] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. doi:10.1109/LICS.2004.1319621.

[9] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, 2014. doi:10.1145/2579080.

[10] Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. doi:10.1145/129712.129740.

[11] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, Oxford, England, October 2000.

[12] Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. doi:10.1007/978-3-642-31485-8_3.

[13] Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL: https://tel.archives-ouvertes.fr/tel-02437532.

[14] Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *LNCS*, pages 357–365. Springer, 2021. doi:10.1007/978-3-030-85315-0_20.

[15] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012. doi:10.1007/978-3-642-31424-7_64.

[16] Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. doi:10.1145/1555746.1555752.

[17] Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1):163–180, 2004. doi:10.1016/j.tcs.2003.10.018.

[18] Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, pages 325–333. ACM Press, 1993. doi:10.1145/158511.158659.

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

[20] Jean-Yves Moyen. *Implicit Complexity in Theory and Practice.* Habilitation thesis, University of Copenhagen, 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

[21] Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings*, volume 10482 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-68167-2_7.

[22] The Coq Development Team. The coq proof assistant, version 8.7.0, October 2017. doi:10.5281/zenodo.1028037.