Realizing Implicit Computational Complexity

Clément Aubert\textsuperscript{1}, Thomas Rubiano\textsuperscript{2}, Neea Rusch\textsuperscript{1}, and Thomas Seiller\textsuperscript{2,3}

\textsuperscript{1} School of Computer and Cyber Sciences, Augusta University, USA
\textsuperscript{2} LIPN – UMR 7030 Université Sorbonne Paris Nord, France
\textsuperscript{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) \cite{12} pioneers in embedding in the program itself a guarantee of its resource usage, using e.g. bounded recursion \cite{10, 18} or type systems \cite{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 \textit{mwp-flow analysis} \cite{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 \cite[p. 11]{20}, 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 \cite{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 \textit{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 \cite{21} and by splitting loops “horizontally” to parallelize them \cite{5}. The second project refines, extends and implements the original analysis to obtain a fast, modular static analyzer \cite{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 \cite{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 \cite[p. 641]{1}, and is implemented e.g. in LLVM as the \texttt{licm} pass. This work \cite{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 \textit{mwp’s} matricial system and typed data flows to generate dependency graphs, to compute an invariance degree for each statement or chunks of

\footnote{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 Ile-de-France region through the DIM RFSI project “CoHOp”.

}
Realizing Implicit Computational Complexity

C. Aubert, Th. Rubiano, N. Rusch and Th. Seiller

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.
Realizing Implicit Computational Complexity  
C. Aubert, Th. Rubiano, N. Rusch and Th. Seiller

References


[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-

