On Implicit Computational Complexity with Applications to Real-World Programs

Neea Rusch

May 2, 2022

Static program analysis fuels software development by providing automated tools for evaluating various program properties without execution. Static analyzers are more powerful that tests, because tests focus on *specific* instances of input-output behavior, while the former evaluates program properties under *all* possible executions. Static analyzers can be designed for two distinct purposes: as powerful *debuggers*, to discover elusive programming errors and to suggest improvements to the source code; or as sophisticated *verifiers*, to guarantee with utmost rigor that the analyzed property holds. This choice impacts the analyzer design: debuggers must, for usability purposes, minimize the rate of false alarms; verifiers must establish with absolute certainty that if no alarms are raised, then the analyzed property is satisfied. Through this functionality, static program analysis delivers the mandatory tools for improving safety, reliability and performances of software.

The ability to analyze various program properties in an automated way is clearly highly desirable; however, the task of designing and developing such analyzers is extremely challenging. There are numerous difficulties: evaluating e.g. memory accesses and loops is demanding because memory access patterns and iterations counts may not be known statically. Then, the soundness of the analysis must be proven to ensure the correctness of the result, while also ensuring correctness of its implementation. What is worse, Rice's theorem proves that any non-trivial semantic property in undecidable. The quest of bringing to life these powerful analysis tools is therefore a simultaneous pursuit of assessing program properties while skirting around the edge of decidability—what an *exciting* landscape.

Given the general character of the problem, numerous distinct approaches to static program analysis exist. These techniques differ by the choice properties they evaluate, and can successfully answer different questions about various classes of programs, but no single approach handles all cases, thus there is continuous room for improvement and need for complementary systems. This presentation is concerned with a specific flavor of analysis, particularly leveraging Implicit Computational Complexity (ICC) theory: ICC offers a unique approach to resource analysis by "embedding" in a program itself syntactical cri-

teria that in return provides some guarantee of its runtime behaviour, typically expressed in terms of complexity. It is thus possible to establish guarantees of program resource usage or discover opportunities for optimizing those behaviors. While precisely determining the complexity of all programs is impossible, these techniques are still widely applicable to a large range of programs.

The rest of this document is organized as follows: the next section gives a brief description of the terminology used in the remainder of this document; it is meant to assist a reader unfamiliar with the bolded terms, and may be skipped without loss of continuity by those already familiar with the terms. The subsequent section will discuss the open problem in the specified domain, as they relate to the selected readings, and offer pathways of potential future exploration. Reader who is unfamiliar with the selected readings, may refer to the appendix for summaries, which are included for convenience.

1 Conceptual preliminaries

Implicit Computational Complexity (ICC) is a subfield in computational complexity theory, that 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 (Moyen 2017). An interesting general description of ICC has been given 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 (Péchoux 2020). The ICC approach to program analysis has several benefits: it drives better understanding of complexity classes, introduces new and often orthogonal analysis techniques, and offers a prospective avenue for realizing complexity analysis. By embedding in the program itself a restriction, 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: the developer can focus on writing their program and concurrently obtain guarantees of its behavior, and this works provided that the restriction is not excessively limiting in expressing algorithms in practice (Moyen 2017).

Static program analysis is a subfield of formal methods, focusing on evaluating program properties without executing its source code (as opposed to *dynamic* analysis, where runtime traces are analyzed). It is typically performed using automated tools and numerous techniques exist. One popular method is **abstract interpretation**: a mathematically rigorous framework, that relies on

abstract domains to evaluate program behaviors, and offers guarantees that the information gathered about a program is a safe approximation to its semantics. **Amortized analysis** is a method for complexity analysis, where cost is associated with resource usage of interest, then averaged over an execution sequence. The relationship between these methods and ICC can be expressed in the potential of the latter to introduce complementary techniques and solve problems that are not efficiently solvable by these alternative approaches.

Compilers transform *source* program, written in one programming language, into another *target* program, in another language, where the target is typically lower-level assembly or machine code to create an executable program. Classic compiler architecture consists of three parts: front-end, middle-end and back-end; which increases their ability to support multiple pairings of source languages and target CPU architectures. During compilation, the transformation of program proceeds over several passes, during which source program is expressed as various kinds of **intermediate representation** (IR): these are data structures and forms of code that ease further processing and optimizations. **Static single assignment** (SSA) form is a property of some IRs, where each variable is assigned a value exactly once, which simplifies and improves performing certain optimizations during compilation.

Formal verification of software relies on mathematical foundations for the strongest levels of assurance and correctness. The intuitive motivation for verifying tools such as static analyzer, is to obtain guarantees of the correctness of the result. Interactive theorem provers assist in this critical task of proving a program satisfies a formal specification of its behavior. **Coq** is a proof assistant for expressing mathematical assertions: it mechanically checks proofs of these assertions and extracts a certified program from the constructive proof. The **CompCert** C verified compiler is a high-assurance, industry-grade, realistic compiler for almost all of `C` language (ISO C 2011), guaranteed to preserve program semantics during compilation and proven correct in Coq.

2 How applications of ICC can help drive advancements in static program analysis

Current state of the art has established, first through Verasco, the suitability of abstract interpretation as a technique for creating a formally verified static analyzers for guaranteeing the absence of runtime errors (Jourdan et al. 2015). The result is particularly impressive as it pairs with CompCert to create a certified compilation pipeline, such that the guarantees established by the analyzer carry over to the executable code. However, at the time of its publication, the tool exhibited notable latency in analyzing short benchmark programs, hinting at significant issues in its usability, and further development of the tool has since then ceased in perpetuity. Currently, formally verified static analysis of various

program properties, particularly of complexity[1], remains an open problem.

Implicit Computational Complexity offers a potential solution for extending this current state of the art. Take for example the flow calculus of *mwp*-bounds: it offers a computation method for evaluating the growth of program variable values, such that the values computed by an imperative program are guaranteed to be bounded by polynomials in the program's inputs if the program has a satisfying derivation (Jones and Kristiansen 2009). This technique is attractive because it is inherently compositional and is not concerned with termination or exact iteration counts, thus enabling efficient analysis and avoiding issues alternative techniques must address. A recent development showed that it is possible to also obtain tight bounds on a similar class of imperative programs, once it is known that their values are polynomially bounded (Ben-Amram and Hamilton 2020). These techniques are, however, purely theoretical in presentation and designed for simplified imperative languages without memory accesses, data structures, classes, etc.; void of the kind of richness one expects in a modern programming language. This hinders their application, limits the spread of these ideas, and the true power of these systems remains under-explored.

Some insight to explain why ICC techniques have remained obscure in static analyzers can be obtained from the habilitation thesis of Jean-Yves Moyen (Moyen 2017). A methodology is plausible only if it captures enough natural algorithms: this raises an issue for many systems as their construction is inherently restrictive in nature. The classical ICC systems, such as the *mwp*-bounds, provide sound but incomplete criteria for a given property, which has the effect that the result can be trusted to be correct, but some "good" programs may be rejected by the analysis. This leads to an under-approximation of programs that fit a complexity class, and the aim is then to make that under-approximation as large as possible. Beyond these theoretical challenges, realizing the techniques is a separate matter, and currently only a few practical ICC-based implementations exist.

Compilers present a suitable target for introducing ICC-based analyses, because they already perform program transformations from source code to intermediate representations. The three-part architecture has the opportunity that an analysis performed at the middle-end could be accessible to all languages supported by the compiler front-end. The middle-end IRs still maintain structural information, but are sufficiently reduced in richness to fit the syntax of ICC systems. The very first attempt to bring these two ideas together was a compiler pass designed to perform loop optimizations based on quasi-invariants (Moyen, Rubiano, and Seiller 2017). This technique relies on an SSA form IR, but given the introduction of a formally verified middle-end for CompCert (Barthe, Demange, and Pichardie 2014), all the necessary pieces are in place to further explore this environment.

[1]A few preliminary related results exists, e.g., Heraud and Nowak and Férée et al. but these are not static analyzers.

2.1 Open problems for further exploration

The proposed ideas for further study are founded on the extension of the previously presented topics and centered on applications of ICC-based techniques, along the following 3 axes.

1. Application. The current world of static complexity analysis is best represented by tools such as `RaML`, focusing on functional languages and implemented using amortized analysis; `ComplexityParser` for Java, and `AProVE` for evaluating termination and complexity of term rewrite systems. While these tools support many features, there are opportunities to take static analysis further using ICC-based methods in several ways: performing multivariate complexity analysis, evaluating resource usage in derived cases (error growth, energy usage, etc.), and using those results to discover program optimizations (loop transformations, parallelization, etc.); cases which are not supported by the other tools. It is probable that exploration of applications of ICC system will also be fruitful in assessing the powerfulness and richness of these systems and help spread ICC theories to wider communities.

2. Compiler integration. Based on the early existing works and the reasoning presented previously, compiler integration is a viable avenue for performing various ICC-based analyses, at various intermediate passes and IRs. This is a particularly intriguing opportunity because of the reach compiler integration introduces, for example, a resource-sensible LLVM would impact all its supported languages. However, the theoretical ICC-systems are void of considerations for complex constructs such as memory accesses, yet those constructs are inherently present in IRs. Exploring this direction will push forward the need to develop analysis systems capable of handling rich and realistic programming languages.

3. Formal verification. As demonstrated by Verasco, a formally verified static analyzer is achievable. Verifying the analyzer itself is desirable, because otherwise guarantee of the correctness of its implementation cannot be obtained, which lowers the level of trust that can be placed on the computed result. ICC systems such the *mwp*-analysis, which is based on inference rules of data flow, provide an amiable pathway for extending in this direction. Some questions of interest include verification of the original technique in Coq; then its integration into a verified compiler such as the SSA-version of CompCert to ensure the result of the analysis carries to the executable program; and evaluating the impact of the SSA form on the analysis itself, since the technique uses matrices whose size depends on number of variables, and it is unclear whether it remains tractable when carried out on programs expressed in SSA form.

Each of the presented axes is individually hard, but they are also interesting, because they are compositional: one could conceivably form a *very hard* problem by exploring the application of ICC systems on rich languages, through an implementation verified with a proof assistant, and integrating that into a certified compiler. These directions are also unconventional in a sense that ICC community has historically focused heavily on theoretical aspects and not on

their applications. Bringing those theories to life will require great effort, but the necessary developments for doing so are in place. It is conceivable that results in implementation will in return drive further enhancements on the theoretical side. On long-term basis, advancements in this domain are significant for the continued need to ease the development of performant and correct software, while the programs themselves are getting increasingly more complex.

3 References

Barthe, Gilles, Delphine Demange, and David Pichardie. 2014. "Formal Verification of an SSA-Based Middle-End for CompCert." *ACM Transactions on Programming Languages and Systems* 36 (1): 1–35. https://doi.org/10.1145/2579080.

Ben-Amram, Amir M, and Geoff Hamilton. 2020. "Tight Polynomial Worst-Case Bounds for Loop Programs." *Logical Methods in Computer Science* 16 (2): 4:1–39. https://doi.org/10.23638/LMCS-16(2:4)2020.

Jones, Neil D., and Lars Kristiansen. 2009. "A Flow Calculus of *Mwp*-Bounds for Complexity Analysis." *ACM Trans. Comput. Log.* 10 (4): 28:1–41. https://doi.org/10.1145/1555746.1555752.

Jourdan, Jacques-Henri, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. "A Formally-Verified c Static Analyzer." *ACM SIGPLAN Notices* 50 (1): 247–59. https://doi.org/10.1145/2775051.2676966.

Moyen, Jean-Yves. 2017. "Implicit Complexity in Theory and Practice." Habilitation à Diriger des Recherches (HDR). University of Copenhagen. https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

Moyen, Jean-Yves, 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.

Péchoux, Romain. 2020. "Complexité implicite : bilan et perspectives." Habilitation à Diriger des Recherches (HDR). Université de Lorraine. https://hal.univ-lorraine.fr/tel-02978986.

4 Appendix: Paper Summaries

4.1 A Flow Calculus of *mwp*-Bounds for Complexity Analysis

(Jones and Kristiansen 2009)

Paper presents an automatable computational method for certifying that the values computed by an imperative program will be bounded by polynomials in the program's inputs. This method drives better understanding of the relationship between syntactical constructions in programming languages and the computational resources required to execute programs. A major contribution is a syntactical proof calculus, the *mwp*-calculus, which allows formal derivations of true statements about programs.

The calculus uses 2-dimensional matrices that record data flows between variables as commands are executed. There are 3 different types of flows: m (maximum), w (weak polynomial), and p (polynomial); which characterize the impact each variable has onto another. The *mwp*-bounds of a program are then obtained from the matrix. Certain patterns of data-flow guarantee polynomial bounds on the computed values while others do not.

For program C and matrix M, there exists a semantic "growth bound" relation, $\models C : M$, if and only if every value computed by C is bounded by polynomial in inputs. This semantic relation is undecidable, but a corresponding provability relation, $\vdash C : M$, holds if and only if there exists a derivation in the calculus where $C : M$ is the bottom line. The paper's main achievement is a soundness theorem proving $\vdash C : M$ implies $\models C : M$. The derivability problem is NP-complete.

The analyzed syntax is restricted to a simple imperative language consisting of variables, Boolean expressions, expressions, and commands. Analysis of arrays, pointers, classes, recursion, inductive data types, etc. is out of scope. Main open questions raised by the authors are evaluating the power of this method and its possible extensions to support richer languages.

4.2 A Formally-Verified C Static Analyzer

(Jourdan et al. 2015)

Paper describes the design and implementation of *Verasco*: a static analyzer based on abstract interpretation, that establishes absence of runtime errors in ISO C99 programs, and is proven correct using the Coq proof assistant. It is compatible with CompCert compiler, such that guarantees established by Verasco carry over to the compiled code.

When using static analysis for program verification, soundness of the analysis is paramount: when no errors are reported, then it must be the case that no errors exist in the analyzed program and all possible execution paths have been

accounted for. However, analyzers themselves may contain bugs therefore ensuring soundness necessitates a mechanical proof. Mechanical soundness proofs is not a new idea, but applying them to abstract interpretation or dataflow analysis is less developed. Compared to previous works, the richness of analyzed language and sophistication of the analysis make Verasco a quantitative increment from previous works.

Verasco uses CompCert's front-end to produce `C#minor` IR, which is then analyzed. The architecture consists of an abstract interpreter, state abstraction layers, and an extendable layer of numerous, cross-communicating abstract domains to keep track of various program properties (array bounds, memory accesses, etc.). Loops are analyzed by computing widening/narrowing fixpoints, function calls are unrolled dynamically, and recursive functions are outside scope. Errors in analysis are logged and analysis runs to completion. Programs that pass analysis without error can be compiled with CompCert, to produce runtime error-free assembly code.

The implementation was tested on small a number of benchmarks, and while absence of runtime errors was established correctly, high analysis times were obtained in some cases. It was also established that use of rich `C` language did not present an issue for implementing Coq proof, and based on similar works, *aposteriori* validation could help reduce runtime latency.

4.3 Formal Verification of an SSA-based Middle-end for CompCert

(Barthe, Demange, and Pichardie 2014)

In compilation, *static single assignment* (SSA) form is a property of intermediate representation (IR), where variables are assigned exactly once. SSA form enables writing simpler, faster and higher quality optimizations. While SSA form offers such benefits and is widely used in standard industry compilers (e.g. GCC and LLVM), formally verifying a SSA-based compiler is challenging and has remained an open problem.

The formally verified C compiler, CompCert, does not use the SSA form. There are two reasons for this: formalizing the semantics of SSA has historically been difficult, and it creates challenges with applying local optimizations since the SSA property is inherently global. This paper addresses both outstanding issues by presenting the first verified SSA-based middle-end, the first formal proof of an SSA-based optimization, and provides intuitive semantics for SSA. This shows that a compiler can simultaneously be realistic, verified *and* rely on SSA form.

The technical contribution is the middle-end implementation, which "pluggs in" at the level of RTL pass, where most CompCert optimizations take place. The implementation reuses the CompCert front-end and back-end, first to translate program from `C` source code to RTL; then to translate again from RTL to assembly. The middle-end operates in 4 phases: first it normalizes the RTL IR,

then translates from RTL to SSA, then applies SSA-based optimizations, and finally performs de-SSA translation back to RTL form.

The paper gives experimental evaluation of the efficiency of the SSA validation, execution of time of the generated code, and evaluates the effectiveness of the performed optimization. On the benchmarked examples, the results do not deviate significantly from those recorded with standard CompCert or the GCC compiler.

4.4 Implicit Complexity in Theory and Practice

(Moyen 2017)

The *habilitation* thesis of Jean-Yves Moyen, summarizes the foundations, advancements and future directions in Implicit Computational Complexity (ICC), over the years 2003-2016. It contains various related papers, grouped into 3 sections that coincide conceptually and temporally.

The first section describes advances in ICC. It introduces foundational theoretical concepts and early attempts to unify the various ICC systems

The second part focuses on the authors current works, centering around equivalences between programs. Influenced by Rice's theorem and dichotomy between programs and functions, this section represents the more recent advances in ICC theory.

The last section is dedicated to the future directions and named experiments in ICC. The theme of this section is concerned with taking these theoretical ICC analysis techniques and applying them in real-world settings e.g., in compilers.

Collectively this work elaborates on the developments and advances in ICC, putting the ideas in chronological order, and delivers structural understanding that is missing from isolated works in the same domain.

4.5 Loop Quasi-Invariant Chunk Detection

(Moyen, Rubiano, and Seiller 2017)

In modern compilers, loop optimization is a standard task and typically focuses on detection of *loop invariant code*: statements that do not change over the iterations of the loop. Following detection of such statements, the invariants can be hoisted from the loop to improve performance.

A related concept is *quasi-invariance*, where the behavior of a statement becomes fixed after a certain finite number of iterations. The number of iterations to reach a fixed state is called the *invariance degree*. While quasi-invariants are a known concept in compiler community, analysis techniques to purposely discover and optimize quasi-invariants do not exist, although it may occur as a byproduct of other transformations.

The paper presents a novel, ICC-inspired optimization technique for quasi-invariant detection. By analyzing loops in an imperative programming language through data flow graphs, the algorithm detects independence between statements or sequences of statements called "chunks," and hoists the invariant code out of loops. If the hoisted block is a nested loop, then the computational complexity of the program decreases. This technique has additional advantages: it can detect arbitrary degree of invariance and can optimize chucks of code, when most optimizations focus on statements individually.

The authors describe two implementations of this technique, one of which is implemented on a simplified toy language and another as a pass in the LLVM compiler. Collectively the paper presents both advancements to ICC theory and the first gallant attempts to move those theories into the real world.

4.6 Tight Polynomial Worst-Case Bounds for Loop Programs

(Ben-Amram and Hamilton 2020)

For a class of programs defined in a non-deterministic imperative language, with bounded loops and restricted arithmetic expressions, it was previously shown that it is decidable whether computed result is polynomially bounded. This paper extends the earlier work by introducing a computational method for obtaining a tight Θ-bound. The bound is multivariate and expressed as values of variables at the conclusion of program, in terms of initial values. The bound is tight up to a multiplicative constant factor, and the computational method is complete: if a bound exists, it will be found.

The problem of analyzing a core language program reduces to analyzing a single simple disjunctive loop. Loop body may contain exponential growth: since polynomially bounded variables can be identified, the rest are substituted with dedicated unmodified variables, which yields a polynomially bounded loop. Then, analyzing the loop is intuitively a question of simulating any finite number of iterations over that loop and whether the computation reaches a fixpoint. Using a generalization operation, it is possible to detect when iteration does not generate anything new and analysis can stop, yielding a tight bound.

Paper includes a detailed correctness proof, which shows that analysis covers all loop behaviors, for lower and upper bound, thus tightness follows. Method can be applied compositionally to nested loops. Time complexity of the algorithm is $|P| \cdot 2^{n^{d+1}}$, where $|P|$ is the size of program, n is number of variables, and d is the highest degree reached. The method is susceptible to combinatorial explosion related to representation of its result.

Improving this method is an open problem. Handling of variable resets, explicit constants, increments and decrements, and deterministic loops are out of scope. Some of these enhancements would make the polynomial boundedness problem undecidable.