An implementation of flow calculus for complexity analysis (tool paper)

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

An implementation of flow calculus for complexity analysis (tool paper)*

Clément Aubert[1][0000−0001−6346−3043], Thomas Rubiano[2], Neea Rusch[1], and Thomas Seiller[2,3][0000−0001−6313−0898]

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

Abstract. We present a tool to automatically perform the data-size analysis of imperative programs written in `C`. This tool, called `pymwp`, is inspired by a classical work on complexity analysis [10], and allows to certify that the size of the values computed by a program will be bounded by a polynomial in the program's inputs. Strategies to provide meaningful feedback on non-polynomial programs and to "tame" the non-determinism of the original analysis were implemented following recent progresses [3], but required particular care to accommodate the growing complexity of the analysis.
The Python source code is intensively documented, and our numerous example files encompass the original examples as well as multiple test cases. A pip package should make it easy to install `pymwp` on any platform, but an on-line demo is also available for convenience.

Keywords: Static Program Analysis · Automatic Complexity Analysis · Program Verification

1 Introduction

The tool we are presenting, `pymwp`, performs *static analysis* of `C` source code. In a nutshell, static program analysis opens up the possibility of specifying not only the behavior of the program, but also its resource consumption, and to obtain this specification by only *observing* (as opposed to *executing*) the program, also making the result more platform-independent. As real-time and embedded systems grow in usage and importance, certifying that programs will run on *any platform* in reasonable time or using sparsely their memory is becoming as crucial as the specification of their correctness. Multiple approaches compete and co-exist in this difficult tasks (some of which are discussed in Sect. 5), and our work

* This material is based upon research supported by the Thomas Jefferson Fund of the Embassy of France in the United States and the FACE Foundation. Thomas Rubiano and Thomas Seiller are also supported by the Île-de-France region through the DIM RFSI project "CoHOp".

takes inspiration from an original approach stemming from the "Copenhagen school" [5,11]'s take on implicit computational complexity [6], that focus on studying the *transitions* between states (e.g. commands) instead of the states in isolation. One of the core idea is that operations *in themselves* may not be "dangerous", but that the *sequence of some operations* can be.

That principle motivated the development of the "mwp analysis" that studies the relationship between the resource requirements of a computation and the way data might flow during the computation [10]. In a nutshell, this analysis attaches *flows* from the inputs to the outputs of a program, and guarantees that no variable will grow in an "unreasonable way" (read, getting bigger than a polynomial w.r.t. the inputs' sizes). While the theoretical development is of interest and can facilitate the understanding of *why* this analysis is sound, we refer the interested reader to our recent improvement of the analysis [3], or to the original paper [10] for a more in-depth understanding, and will simply restate the bare minimum to understand the interest and challenges of the implementation we are presenting here.

2 Implementing a flow calculus developed abstractly

Language and Features The author of the original flow calculus considered only an abstract imperative programming language, that they described as "very rudimentary", and our first task was to map their language to an actual programming language. The `C` programming language, because of its central role and close connection to simpler imperative language, was the ideal target, and offered little resistance[4]. This also enabled to build on previous work using a similar flow analysis to implement loop invariant detection [13,14] which was implemented on `C` code. Following the choices made in the latter work, we decided to implement the analysis in Python because of its vast collection of libraries and its plasticity.

The source code is parsed using pycparser, that uses the C99 standard (ISO/IEC 9899) [9]. `pymwp` support basic data-types, operations and decision structures (`if ... else` and `while`). Our list of features from the `C` programming languages is still modest, but rapidly growing. It should also be noted that the supported fragment is already enough to determine that e.g. basic implementations of the exponential function (using "naive" approach or a more subtle version alike) do not have polynomial bounds (something that is expressed by the presence of the ∞ coefficient between variables).

Examples and Tests We took a particular attention to testing our implementation.

[4] The only possible exception is a command `loop Xi { C }` that is iterating `Xi` times the (sequence of) command(s) `C`, and have no direct counterpart in the `C` language. Mapping this structure to a restricted form of `for` loop is currently under development, but will not impact our development nor its expressivity much. On the other hand, small improvements such as the analysis of constants was also added.

- All the examples from the original paper [10] have been converted to `C` and checked for consistency.
- Numerous `C` programs – from very basics to more demanding – have been implemented and checked "by hand".
- All the steps needed for the analysis – that we detail in the next section – have unit tests, and so does the analysis in itself.
- All examples can be repeatedly profiled to analyze efficiency of the implementation.

Note that since the analysis is focused with the dependencies between input and output, some of our program examples may not terminate but still "pass" the analysis (e.g. have assignment without ∞ coefficients): this comes from the fact that our analysis is interested with the data-size analysis, and not with termination. However, if a program pass the analysis *and is known to terminate*, then it *has* an execution time that is bounded by a polynomial in the program's input's size.

3 Implementation structure

3.1 Package architecture

The `pymwp` package follows a modular architecture pattern where each module performs designated tasks and has minimal dependencies. This organization is significant for three reasons: it creates a flexible design that is easy to extend, it makes the source code unit-testable, and it enables using the implementation in two different ways: through command line interface or by importing selected modules in a Python script.

The design pattern creates separation of concerns between individual modules. New functionality, such as adding optimization steps or support for additional `C` language features, can be introduced by "plugging-in" new modules. Existing functionality can be improved or replaced similarly as needed.

Importing minimal dependencies and using classes and functions to implement modules, enables unit testing the package thoroughly. Most modules are testable in isolation without mocks or stubs. Test suites are run using `pytest`, and `pytest-mock` is used to mock the built-in functionality when necessary. Continuous integration is used to execute unit tests on a build server on every opened pull request and commit to master. Second automation script runs unit tests against multiple Python versions to ensure continued compatibility across different runtimes. This testing strategy provides assurance that new changes do not break existing functionality, and if issues are discovered, it raises alerts and indicates which module is the source of failure.

The `pymwp` package can be used through a command line interface to perform analysis on `C` program files. An alternative use case is importing its modules in a Python script and performing computation steps without executing an end-to-end analysis. Modular package architecture is essential for accommodating these two use cases.

The implementation includes type annotations, uses self-documenting code style, and a build-time linter to ensure PEP8-style compliance and high code quality.

3.2 Overview of modules

The modules are precisely documented and discussed in the documentation, but we briefly discuss them below, in the order in which they are best discovered.

At the core of the analysis' mechanism is a mathematical structure called a "semi-ring", that we implemented and improved compared to the original development. The semi-ring we implemented is the original "mwp"-semiring [10], endowed with an additional ∞ value `i` representing failure [3] in a "local" way: instead of simply stopping the analysis – as was done in the original paper –, it is carried on in our implementation. This choice allows to not only "flag" the program as not respecting the polynomial data size growth, but to specifically "flag" the actual variables not respecting the bounds. We believe this is an important added value for usability, as it now pinpoints the exact part of the program where data grow "out of proportions" instead of simply stopping, providing usable feedback to the programmer. Furthermore, the actual semi-ring used is a parameter of the analysis, with the hope that finer semi-ring would allow for more precise analyses, or even analyses on different metrics on the programs – something we briefly discuss in our research paper [3, Section 3.2].

Monomial and polynomial are classes used to represent possible choices: instead of containing single values, the (unique) matrix representing the flows produced by our analysis contains polynomials, which are ordered lists of monomials. A monomial, in turn, is a value from the semi-ring along with a list of possible choices that could lead to obtain it, a technique we used to "tame" the non-determinism of the original analysis. Monomials are represented as ordered list w.r.t. an ordering having the interesting property that the product of monomials (when leading to non-zero results) is monotonous. This is used to implement more efficient algorithms for, e.g., multiplication of polynomials in a way reminiscent of work based on Gröbner bases.

Matrix represents matrices and basic operations on them. Note that since it is possible to obtain dense matrices with our analysis (as illustrated by the `dense.c` example), it is not possible to use algorithms optimised for sparse matrices. A more fine-grained study to obtain the "average density" of matrices, using existing static dependences (or other data-flow) analysis like liveness analysis, could be undertaken to better justify or revise this choice.

Relation class represents the variables of a `C` program and a matrix of polynomials. Relation list holds a list of relations and enables representing multiple intermediate matrices and variables during analysis and performing operations on them collectively.

Delta_graphs class (currently in a separate branch) introduces a data structure designed to keep track of assignment leading to infinite coefficients. The information is kept within a graph whose vertices are monomials populated during the analysis by adding those monomials that appear with an infinite coefficient.

This graph is structured in layers where each layer corresponds to the size of the monomials it contains; this extra structure is used to implement efficiently a "fusion" method which simplifies the structure by performing some algebraic simplifications. This is discussed in more details in our research paper [3, Section 5.2].

Analysis is the most important class, that calls pycparser to parse the `C` source code given as input, and then proceed to analyze the obtained Abstract Syntax Tree (AST). The `compute_relation` method recursively analyses each AST node and applies the rules of the modified flow calculus to obtain a relation list corresponding to all possible matrices of that AST node. Relations are iteratively composed, resulting in one final relation that represents all variables of the `C` program under analysis and its corresponding matrix. Next, this relation is evaluated to determine which flows stay within polynomial bounds. Analysis returns the relation and a list of passing choices. This list is empty when no choices exist, indicating the analyzed program does not have polynomial bounds.

4 Obstacles: computing with matrices of choices

Our main challenge was the efficiency of our analysis. Unlike the original flow calculus, where analysis is performed on a matrix representing singular derivation and matrices are resized repeatedly, each matrix in our implementation represents multiple derivations, and as such their sizes can grow quickly and operations on them become more costly. In the analysis of program flows, all nondeterminism is contained in assignment and operations, meaning a very simple program of n lines can have 3^n different derivations and becomes untractable, as exemplified by explosion.c.

Running analysis evaluation phase to check all possible flows is particularly costly if no prior optimization is performed. Hence, removing redundant, useless and uninteresting choices and unnecessary operations, especially when we focus on worst cases, is required to be able to carry out the analysis in a timely manner.

When adding or multiplying polynomials, which consist of monomials, we check if a monomial is contained or included by another, and exclude all redundant cases (cf. a4ec807 or 6f8e694). This is also be done when inserting monomials, that way we keep polynomials free of implementation choices that we would otherwise have to handle in during evaluation. Profiling – using `cProfile` – was carried out to ensure this similification resulted in performance gain.

Choices that lead to "infinite" flow also are saved and removed from the possible choices in the evaluation. This elimination of choices is done using clique detection and deletion over a weighted graphs representing a distance between monomials (in delta_graphs.py).

To address the need to resize matrices frequently, we implemented a suitable data structure and method to handle this efficiently. First by modelling analysis state as a relation, which combines program variables and its associated matrix, and allows comparing this data for equality. Then, during analysis while com-

posing relations, the `homogenisation` method checks if resizing is necessary and omits it otherwise.

5 Limitations and strengths

This work is a proof of concept and has its set of limitations. The list of features details what can and cannot be analyzed currently. Since the analysis does not track memory uses, it handles pure functions only. Modelizing and managing memory is not impossible, but is a feature for future enhancement. Analysis of external function calls remains to be implemented, but functions calls can be analyzed by clever inlining, as explained in the research paper. A feature for saving analysis results in a file has been implemented in preparation for future enhancement like exports of evaluation when including external functions. Handling of operations is currently limited to binary operations, but analysis of operations of greater arity can be handled by splitting such operations into multiple statements. Other significant future enhancements include adding support for arrays and pointers.

The strength of the analysis is to focus only on characterizations of "chunks" – i.e. sequences of commands – of any size of the program allowing to abstract values and their encoding. Here the problems of intervals and sets of possible values does not exist anymore. Of course this comes with imprecision we try to minimize using eliminations and optimizations techniques. The implementation itself is cross-platform compatible, heavily and continuously tested, and available for installation through Python Package Index, making its installation and use accessible and user-friendly.

There exists other static analysis tools with similar goals. Resource Aware ML can statically and automatically determine upper and lower bounds of resource usage and evaluate other metrics to include evaluation steps and heap space, but RaML analyzes programs written in `OCaml`, not `C`. SPEED [8] and Costa [2] are similarly designed for different programming languages. CerCo project aimed at building a `C` compiler with built-in resource analysis, but focused on runtime. By performing data size analysis on `C` programs, `pymwp` fits a niche not met by these existing alternatives. Further its packaged design and minimal dependencies enable its integration into larger systems, or it can be used independently as a standalone tool.

6 Conclusion

This work attempts to answer questions [10] asked years ago, showing that, taken as it was described, the analysis cannot scale to realistic program in a real programming language as `C`: while the considered analysis is definitely powerful and elegant, its mathematical nature let some costly operations go unchecked. However we have shown that, extended and coupled to optimizations techniques, its result allows the development of a novel, powerful and realistic static analysis

tool. We think that from here, much more can be done, but that our first steps are encouraging.

We have the mathematical theory to handle function definitions and calls [3, Section 4], other operators (unary or n-ary), and suspect that simple data-types such as arrays or lists should oppose little to no resistance. This will open up the possibility of *composing* analysis, and as such enable the call to external libraries in the analyzed programs: to prepare for this, the outputting to files of the result of the analysis was already implemented in the `file_io` class. Adding e.g. pointers to the theory will probably be a greater challenge, but there seems to be no intrinsic reason for the analysis not to be able to handle them.

Another complementary direction we find very exciting is to certify the analysis using the Coq proof assistant [1], and to implement the analysis in certified tools such as the Compcert compiler [12] (or, more precisely, its "static single assignment" version COMPCERT-SSA [4]) or certified-llvm [15]. The plasticity of both tools and of the implemented analysis should allow to enable to port our results and approaches to programs written in other languages than `C`. Furthermore, as complexity analysis is notably difficult in Coq [7], we believe a push in this direction – allowed by the plastic nature of our tools – would be extremely welcome.

References

1. Coq documentation, `https://coq.github.io/doc/`
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Costa: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects. pp. 113–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
3. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: An extended and more practical mwp flow analysis. In: Submitted to APLAS 2021 (2021)
4. Barthe, G., Demange, D., Pichardie, D.: Formal verification of an ssa-based middle-end for compcert. ACM Trans. Program. Lang. Syst. **36**(1), 4:1–4:35 (2014). `https://doi.org/10.1145/2579080`
5. Ben-Amram, A.M., Jones, N.D., Kristiansen, L.: Linear, polynomial or exponential? complexity inference in polynomial time. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings. LNCS, vol. 5028, pp. 67–76. Springer (2008). `https://doi.org/10.1007/978-3-540-69407-6_7`
6. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. LNCS, vol. 7388, pp. 89–109. Springer (2011). `https://doi.org/10.1007/978-3-642-31485-8_3`
7. Guéneau, A.: Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes). Ph.D. thesis, Inria, Paris, France (2019), `https://tel.archives-ouvertes.fr/tel-02437532`
8. Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: Precise and efficient static estimation of program computational complexity. In: Proceedings of the 36th Annual

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 127–139. POPL '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1480881.1480898
9. Joint Technical Committee ISO/IEC JTC1: Iso/iec 9899:1999 - programming languages — c. Tech. rep., International Organization for Standardization and the International Electrotechnical Commission (12 1999), https://www.iso.org/standard/29237.html
10. Jones, N.D., Kristiansen, L.: A flow calculus of *mwp*-bounds for complexity analysis. ACM Trans. Comput. Log. **10**(4), 28:1–28:41 (2009). https://doi.org/10.1145/1555746.1555752
11. Jones, N.D., Nielson, F.: Abstract Interpretation: A Semantics-Based Tool for Program Analysis, Handbook of Logic in Computer Science, vol. 4, pp. 527 – 636. Oxford University Press (1995)
12. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
13. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) ATVA. LNCS, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7
14. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk motion by peeling with statement composition. In: Bonfante, G., Moser, G. (eds.) Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017. EPTCS, vol. 248, pp. 47–59 (2017). https://doi.org/10.4204/EPTCS.248.9, http://arxiv.org/abs/1704.05169
15. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formal verification of ssa-based optimizations for LLVM. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. pp. 175–186. ACM (2013). https://doi.org/10.1145/2491956.2462164