Verification of C Programs Via Natural Semantics and Abstract Interpretation (Extended Abstract)

Roberto Bagnara[1], Patricia M. Hill[2], Andrea Pescetti[1], and Enea Zaffanella[1]

[1] Department of Mathematics, University of Parma, Italy
{bagnara,pescetti,zaffanella}@cs.unipr.it
[2] School of Computing, University of Leeds, UK
hill@comp.leeds.ac.uk

We are witnessing a substantial lack of available tools able to verify the absence of relevant classes of run-time errors in code written in (reasonably rich fragments of) C and C++. This is despite the progress made in recent years in the fields of program analysis and verification, and despite the huge impact such tools could have on the quality of a good portion of our software universe. It is interesting to observe that, among the dozens of freely available software development tools, hardly any, by analyzing the program semantics, are able to certify the absence of important classes of run-time hazards such as, say, the widely known *buffer overflows* in C code. The reason is, of course, that C and C++ are complex languages and the techniques that can be used to dominate this complexity still do not reduce tool development to simple, manageable tasks. Our overall aim for this research is to investigate how known techniques based on natural semantics and abstract interpretation can be extended so as to conveniently formalize and implement a range of analysis and verification tools for modern imperative languages such as C and C++.

The language. With this aim in mind, in [2] we define a core language —called CPM— that has much in common with C and includes several features that are problematic from the point of view of the semantic analysis of C and C++ code: recursive functions, run-time system and user-defined exceptions, realistic data and memory models, pointer types to both data objects and functions, and non-structured control flow mechanisms. Note that the contradiction between targeting "real" imperative programming languages and choosing CPM, an unreal one, is only apparent. As C misses exceptions and C++ is too hard as a starter, choosing any one of these would not have allowed us to assess the adequacy of the methodology described below with respect to the above goals.

Static analysis. Verification of many program properties using static program analysis via abstract interpretation [9, 11] is now a well-researched area. Static analysis is conducted by mimicking the concrete execution of the programs on an *abstract domain.* This is a set of computable representations of program properties equipped with all the operations required to mirror, in an approximate though correct way, the real, *concrete* executions of the program. Therefore, a formal *concrete semantics* for the language to be analyzed that models all the aspects of executions that are relevant to the properties of interest must be provided. Of course, we need to work on a language that is completely defined. For the C language, for instance, this can be achieved by converting the source programs to some more constrained language —like CIL, the *C Intermediate Language* described in [17]— where all ambiguities have been removed and by fixing an ABI (*Application Binary Interface*) so as to conform to the C implementation of interest. The problem is then to define a concrete semantics for the fully specified language so as to ensure that:

(i) this semantics is recognizable as a sound characterization of the language at the intended level of abstraction;
(ii) this semantics observes the properties that are the subject of the verification problems of interest;
(iii) this semantics allows for the computation of precise abstractions.

We now review the first two points; we will come back to the third point after introducing the *abstract semantics*.

Concrete semantics. We need a formal semantics that can be recognized (without requiring a strong mathematical background) as corresponding to the intuitive, often involved and incomplete explanations provided by standardization documents. For this purpose, we adopted the G^{∞}SOS approach of Cousot and Cousot [12] which generalizes with infinite computations the *natural* semantics approach by Kahn [16] which, in turn, is a "big-step" operational semantics defined by structural induction on program structures in the style of Plotkin [18]. A semantics for CPM is then expressed by means of a concise set of rather simple rules that are quite readable and, most importantly, directly correspond to executable Prolog clauses. What was not clear to us when we started this work is whether the approach "scales" when applied to languages like C: for example, how can run-time errors and non-structured control flow mechanisms be modeled in this framework? We now know that the natural semantics is fit for the purpose:

- the formal semantics of CPM[3] has been successfully explained to undergraduate students in their 3rd year of Computer Science;
- the Prolog implementation of this formal semantics in the ECLAIR system[4] (with the help of a C++ implementation of memory structures) is efficient enough to allow the execution of non-trivial C programs, something that enables everyone to build confidence on the fact that the concrete semantics is faithful to the intuitive, informal semantics.

Concerning point (ii) above, we can only formally reason about properties if they are observable in the chosen concrete semantics. For example, if we want to

[3] Which includes CIL as a sublanguage in addition to the exception handling features of C++ and Java.
[4] The 'Extended CLAIR' system targets the analysis of mainstream programming languages by building upon CLAIR, the 'Combined Language and Abstract Interpretation Resource', which was initially developed and used only in a teaching context (see `http://www.cs.unipr.it/clair/`).

prove that a program uses pointer arithmetic in a safe way, we need a concrete semantics that allows us to observe all the unsafe uses. Such a concrete semantics cannot simply model pointers as plain addresses, as more information is required than that to detect the violations. In the concrete semantics for CPM, these violations are optionally reported as run-time exceptions so that, proving that such an exception can never be thrown, amounts to proving the desired property. All exceptional and undefined behaviors (such as divisions by zero, overflows of signed integer variables and dangerous uses of the shift operators) are modeled by exceptions. This, besides the need to deal with user-defined exceptions as found in C++, Java and Python, is the reason for the inclusion in CPM of exception propagation and handling mechanisms. Note however that accommodating exceptions impacts on the specification of the other components of the semantic construction. For example, short-circuit evaluation of Boolean expressions cannot be normalized as proposed in [8], because such a normalization process, by influencing the order of evaluation of subexpressions, is unable to preserve the concrete semantics as far as exceptional computation paths are concerned.

Abstract semantics. Following the abstract interpretation approach, we also require an *abstract semantics* that has a correlation with the concrete semantics. In addition, we require that appropriate abstract domains are available that can provide correct approximations for the values and all the operations that are involved in the concrete computation [9–12]. For CPM we have formally defined in [2] an abstract semantics framework that follows and extends the approach outlined in the works by Schmidt [19–21]. We have proved that any semantics within this framework will have a safe correlation with the concrete semantics (a summarized version of this proof is available in [2]). Moreover, this framework has been designed to be both modular and generic. It is modular because the overall static analyzer is naturally partitioned into components with clearly identified responsibilities and interfaces, something that greatly simplifies both the proof of correctness and the implementation. It is generic, since it is designed to be completely parametric on the analysis domains. In particular, and here we come to point (iii) above, it provides —differently from all published proposals we know of— full support for *relational* domains (i.e., abstract domains that can capture the relationships between different data objects). Achieving this goal constrains the design of both the concrete and the abstract semantics. As was the case for the concrete semantics, the abstract semantics rules for CPM are almost directly translated to generic Prolog code that can be interfaced with specialized libraries implementing several abstract domains, including accurate ones such as those provided by the *Parma Polyhedra Library* [3–5]. So this working prototype, which is currently being extended with the pointer analysis described in [13–15], demonstrates that the proposal of Schmidt can play a crucial role in the development of reliable and precise analyzers for real imperative languages including C, Java and, we believe, C++ and RPython (`http://pypy.org/`).

Further work. Although our framework is only fully specified for the core CPM language, and this encompasses C but not C++, we do not have a definite answer concerning the appropriateness of our proposal for the verification of C++ programs. That said, we do not see what, in the current design, would prevent the extension of the core language together with its concrete and abstract semantics so as to handle any other features of mainstream, single-threaded imperative programming languages.

Our proposed analysis framework is parametric on abstract memory structures. While the literature seems to provide all that is necessary to realize very sophisticated ones, we can confidently predict that, among all the code out there waiting to be analyzed, some will greatly exacerbate the complexity/precision trade-off. The ability to analyze C programs will confront us with a huge variety of inputs and it is hardly likely that the same compromises will be able to accommodate programs as diverse as the huge, pointer-free, synthesized loops handled by *ASTRÉE*[5] and, say, libraries for manipulation of strings. However, these are research problems for the future — now we have a formal design on which analyzers can be built, our next goal is to complete the build and make this technology truly available and deployable.

References

1. *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, volume 29 of *ACM SIGPLAN Notices*, Orlando, Florida, 1994. Association for Computing Machinery.
2. R. Bagnara, P. M. Hill, A. Pescetti, and E. Zaffanella. On the design of generic static analyzers for modern imperative languages. Technical Report `arXiv:cs.PL/0703116`, Dipartimento di Matematica, Università di Parma, Italy, 2007. Available from `http://arxiv.org/`.
3. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1–2):28–56, 2005.
4. R. Bagnara, P. M. Hill, and E. Zaffanella. Not necessarily closed convex polyhedra and the double description method. *Formal Aspects of Computing*, 17(2):222–257, 2005.
5. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. Æ. Mogensen, D. A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer-Verlag, Berlin, 2002.
7. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of*

[5] The *ASTRÉE* analyzer can automatically verify the absence of some kinds of run-time errors in large safety-critical embedded control/command codes [6, 7].

the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), pages 196–207, San Diego, California, USA, 2003. ACM Press.
8. P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, NL, 1999.
9. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, New York, 1979. ACM Press.
11. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
12. P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Albuquerque, New Mexico, USA, 1992. ACM Press.
13. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* [1], pages 230–241.
14. M. Emami. A practical inter-procedural alias analysis for an optimizing/paralleling C compiler. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, August 1993.
15. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* [1], pages 242–256.
16. G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Passau, Germany, 1987. Springer-Verlag, Berlin.
17. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. N. Horspool, editor, *Compiler Construction: Proceedings of the 11th International Conference (CC 2002)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, 2002. Springer-Verlag, Berlin.
18. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
19. D. A. Schmidt. Natural-semantics-based abstract interpretation (preliminary version). In A. Mycroft, editor, *Static Analysis: Proceedings of the 2nd International Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 1–18, Glasgow, UK, 1995. Springer-Verlag, Berlin.
20. D. A. Schmidt. Abstract interpretation of small-step semantics. In M. Dam, editor, *Analysis and Verification of Multiple-Agent Languages*, volume 1192 of *Lecture Notes in Computer Science*, pages 76–99. Springer-Verlag, Berlin, 1997. 5th LOMAPS Workshop Stockholm, Sweden, June 24–26, 1996, Selected Papers.
21. D. A. Schmidt. Trace-based abstract interpretation of operational semantics. *LISP and Symbolic Computation*, 10(3):237–271, 1998.