Language-Parametric Compiler Validation with Application to LLVM Extended Abstract

1. Motivation

Modern optimizing compilers such as LLVM [6] and GCC [3] are huge and complex, and mature releases routinely have uncaught bugs [15]. Beyond harm to software development, the lack of formal correctness guarantees for the compilation process seriously limits the guarantees other software systems can provide. For example, the seL4 operating system microkernel [5] had to use a custom compilation verification solution in order to provide formal correctness guarantees for the microkernel [14]. As another example, most iOS applications and all watchOS and tvOS applications are shipped by developers to the Apple Store as LLVM bitcode [1, 4] and compiled to machine code on Apple's servers. This has raised concerns among developers that the machine code may not be identical to what they tested before shipping, exposing unexpected bugs [17]. Compilation verification can be valuable to increase confidence in the correctness of the final native code.

In this work, we attack the problem of *compilation verification*: providing a formal guarantee that a compilation of a program is correct. We do that not only theoretically, as an instance of program equivalence, but from a practical standpoint as well: We aim for a solution suitable for production optimizing compilers such as LLVM. Full static verification of the compiler source code itself, e.g., as in CompCert [7], has only been shown to work for compilers designed specially from the ground up for formal proofs, and requires extensive proof engineering by formal methods experts (see Section 6). We pursue an alternative approach called **Translation Validation**, better suited to production compilers.

Translation Validation (TV) [11] aims to prove correctness of a single compilation run, by considering only a specific pair of input and output programs. TV techniques are well suited to practical compilation verification because they can be retrofitted to production compilers and (in our approach) require relatively little formal methods expertise from compiler developers, once the operational semantics of compiler internal representations (IRs) are formally defined.

There are three essential components of a TV system:

1. A formal notion of program equivalence.
2. A verification condition (VC) generator that generates a sufficient set of obligations to be discharged in order to prove equivalence. Verification conditions relate program points and variables in the input and output programs.
3. A proof system that accepts the verification conditions, generates a machine-checkable equivalence proof, and checks the proof for correctness.

2. Limitations of the State of the Art

There is a rich literature of successful TV systems for compilation verification, for example [10] for GCC and [16] for LLVM, among others. The main limitation of these systems is that each of them is custom-tailored for a particular sequence of transformations, and moreover, specialized for a specific, common intermediate language for the input and output programs. The fixed language makes it difficult to use the approach for modern compilers, which typically use different intermediate languages for different stages (e.g,. LLVM uses at least three different languages). For example, Necula's work on GCC [10] is limited to the lowest-level IR form, Register Transfer Language (RTL), and does not apply to the bulk of the optimizations which are done on the higher-level GIMPLE representation [2]. Moreover, none of these previous systems would be able to verify a key phase like Instruction Selection in LLVM, which converts between two different IRs. The best effort has been to translate both input and output programs to a third, common internal representation as a preliminary step [14], which introduces two new unverified language translators in order to verify the original translator.

3. Key Insights

The key insight underlying our work is that two of the three TV system components mentioned above can be generalized to be *transformation- and language-independent*: the formal notion of equivalence, and the proof system. The only component that needs to depend on specific transformations is the VC generator, and that is conceptually the simplest because it requires little formal methods expertise, making our approach suitable for real-world compiler teams which typically lack such expertise. Together, these make our approach far more practical than previous solutions.

More specifically, we design a program equivalence checker, KEQ, that can be used *unchanged* for different transformation passes and input/output language pairs. KEQ accepts operational semantics definitions of the input and output languages as parameters, as well as the VC for a transformation sequence. The operational semantics of each IR must be defined once, and KEQ can then be reused across all the transformations found in the compilation path. Moreover, the input and output languages can be completely different, as long as programs can be related using the VC.

In this work, we showcase the power of these properties by using KEQ in a prototype TV system for the Instruction Selection phase of LLVM, a sophisticated phase that translates

LLVM IR [8] to Machine IR [9] representing the x86-64 instruction set. Moreover, in our ongoing work (not part of the current paper), we are applying KEQ *unchanged* to validate the register allocation phase of LLVM, with a VC generator that treats the allocator completely as a black box (i.e, has no knowledge of the allocation algorithm), and we plan to apply it to LLVM-to-LLVM transformations in future.

We provide a strong theoretical foundation for KEQ and its correctness. First, we present a formalization for program equivalence which we call cut-bisimulation (a variant of weak bisimulation [13] widely used in the literature) that is weak enough to enable proofs for realistic compiler transformations, and yet expressive enough to subsume most of the equivalence properties that have been used in existing TV systems. Cut-bisimulation can express equivalence of programs in two different languages, as long as a VC can relate program states in the two languages. We then use cut-bisimulation to define an equivalence checking algorithm that forms the theoretical basis for KEQ.

Given those, only the VC generator needs to be designed per transformation (or set of transformations). Such generators need to provide a candidate relation between input and output program states that the proof system can verify to be indeed a cut bisimulation relation. We add a small number of compiler hints in the Instruction Selection phase of LLVM for our prototype VC generator.

In short, this work presents the first TV system that is reusable across the whole compilation path and requires the minimum amount of customization per transformation and (intermediate) language: a semantic definition of every language found in the compilation path and one or more proof generators that use transformation-specific information.

4. Main Artifacts

We present five artifacts, the first three of which are transformation-independent.

1. The KEQ program equivalence checker is implemented as a tool within the $\mathbb{K}$ Framework [12].
2. The LLVM IR semantics definition is implemented in $\mathbb{K}$.
3. The x86 semantics definition is also implemented in $\mathbb{K}$.
4. The (transformation-specific) verification condition generator for Instruction Selection is implemented as a Python script and relies on a minimal hint generator added to the LLVM compiler.
5. Finally, a rigorous formalization of cutbisimulation along with an equivalence checking algorithm based on cut-bisimulation that is the theoretical foundation behind KEQ.

We evaluate these artifacts on 2798 functions of the GCC SPEC 2006 benchmark with supported features. We correctly validate the translation of 97.6% of the supported functions in GCC, i.e., 2730 / 2798 functions.

5. Key Results and Contributions

This paper presents a TV system design for compilation verification in real-world optimizing compilers that requires minimal customization for the various transformations and intermediate languages of the compilation path.

The main contributions of this paper are:

- KEQ, a new tool for checking program equivalence that accepts the operational semantics of the input and output languages as parameters, and is independent of the transformation used to generate the output. This is the first program equivalence checking tool known to the authors that is language-parametric instead of containing hard-coded language semantics as is the norm in the literature.
- A rigorous formalization, namely *cut-bisimulation*, for weak bisimulation variants that have been traditionally used in different TV systems. We use cut-bisimulation as the basis of the KEQ equivalence checking algorithm, and provide a correctness proof for that algorithm.
- A prototype of a Translation Validation system for the Instruction Selection pass of the LLVM compiler infrastructure, able to automatically prove equivalence for translations from LLVM IR when compiling to the x86-64 instruction set. This is a mature, sophisticated translation phase of a production compiler. Moreover this is a transformation that uses different input and output languages, and as such has not been previously addressed by the state of the art.

6. Why ASPLOS

This work aims to provide practical methodologies and tools for enforcing correctness of, and increasing confidence in, modern optimizing compilers. These goals are increasingly important to many in the ASPLOS community who are interested in security, software reliability, and trust in today's computing systems. The work makes technical contributions to programming languages (language semantics and parametric program equivalence), compilers (practical TV system for compilation verification), and formal methods (cut-bisimulation formalization). This work brings modular validation of production compilers such as LLVM within reach of today's compiler teams.

7. Citation for Most Influential Paper Award

The ASPLOS 2021 paper "*Language-Parametric Compiler Validation with Application to LLVM*" presented the first Translation Validation approach where all but a small part is transformation-independent and parameterized in the source and output languages, allowing it to be reused across most phases of production compilers, including difficult-to-check ones such as Instruction Selection in LLVM. This work brings modular validation of production compilers within reach of today's compiler teams.

References

[1] Apple Corp. iOS App Distribution Guide. https://developer.apple.com/library/etc/redirect/DTS/iOSAppDistGuide, 2019. Accessed: February 2019.

[2] Free Software Foundation. GNU Compiler Collection Internals. %https://gcc.gnu.org/onlinedocs/gccint/Passes.html, 2019. Accessed: August 2020.

[3] GCC. GNU Compiler Collection. https://gcc.gnu.org, 2020. Accessed: August 21, 2020.

[4] Jeremy Horwitz. Apple Watch apps instantly went 64-bit thanks to obscure Bitcode option. *VentureBeat*, 2018.

[5] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

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

[8] LLVM. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, 2020. Accessed: August 21, 2020.

[9] LLVM. LLVM Target-independent Code Generator. http://llvm.org/docs/CodeGenerator.html#machine-code-representation, 2020. Accessed: August 21, 2020.

[10] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM.

[11] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, pages 151–166, London, UK, UK, 1998. Springer-Verlag.

[12] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[13] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.

[14] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 471–482, New York, NY, USA, 2013. ACM.

[15] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 294–305, New York, NY, USA, 2016. ACM.

[16] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for llvm. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 295–305, New York, NY, USA, 2011. ACM.

[17] David Wheeler. To bitcode, or not to bitcode? *iovation*, 2015.