

The Design and Implementation of a Certifying Compiler

George C. Necula

Electrical Engineering and Computer Science Department
University of California at Berkeley
Berkeley, CA, 94720, USA
necula@cs.berkeley.edu

Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15213, USA
petel@cs.cmu.edu

Optimizing compilers continue to be critical tools in all software development processes. Even more than in 1998 when this paper first appeared, production-quality compilers have become so complex that they defy any attempt at formal verification. Compilers are debugged using large numbers of test cases, some of which are distilled at considerable expense from large application programs. Besides being expensive and time-consuming, the effectiveness of testing is reduced by the practical impossibility of adequately covering the execution paths of the compiled programs.

In this paper we proposed a technique for checking the result of each compilation instead of verifying the compiler's source code. This was inspired in part by the technique of *run-time result checking* [14] and the observation that it is often easier to check the output of a program than to verify the correctness of the program itself. In addition to the claim that this fundamental concept could be made practical for realistic compilers, we made two additional claims of engineering benefits for compiler writers. First, we claimed that this approach would be far less brittle than full compiler verification, so that the process of checking each compilation could be insensitive to most kinds of changes that might be made in the compiler during its development. Second, we claimed that failed checking attempts would point out the precise places in the compiled code that did not correspond with the source code semantics, thereby simplifying greatly the diagnosis of compiler bugs. In a nutshell, we claimed that certifying compilation was not only feasible, but also offered great engineering advantages.

For our research, this paper was significant because it gave us our first opportunity to present concrete evidence of our claims. In order to simplify matters, we checked only that the compiled code is type safe instead of checking its full correspondence with the source code. This simplified the checking procedure significantly, and it reduced only slightly the effectiveness of the approach as a bug-finding tool.

Our focus on compilers for type-safe languages was not accidental. This paper came a short time after we developed the idea of proof-carrying code (PCC) [9, 6], which allows the receiver of an untrusted program to verify quickly and reliably that the code is safe to execute. To make this possible, the code producer must also produce an easy-to-check proof of the code's safety. Thus, PCC essentially provides a mechanism to shift the burden of verifying the safety prop-

erties of the code to the code's producer.

Of course, the major practical question left largely unanswered in [9] was how to produce such proofs. While proving program correctness properties is very hard in general, proofs of type safety can be constructed easily for programs that are written according to certain typing rules. This observation suggests that it should be possible to engineer a compiler to, in essence, "preserve" such type-safety proofs through all of its optimization and code-generation phases. In previous work, the TIL compiler achieved this by expressing typing proofs using type systems of increasing expressiveness and complexity as it moves farther from the source code, and then depending on a typechecker to do the checking. (The original paper on TIL [13] also appears in this special volume.) In our certifying compiler we chose an approach that requires fewer changes to the compiler and makes it easier (albeit less systematic) to handle the complex reasoning needed to verify aggressive optimizations all the way to machine code. We essentially adapted a conventional compiler in a minimal way so that it preserved not the whole proof but just the typing loop invariants discovered by the type checker. We could then we reconstruct the proof using a standard verification condition generator (VCGen) and a simple theorem prover.

This paper briefly describes a compiler that was developed as part of the first author's Ph.D thesis [7]. It is a simplified replica of a real optimizing compiler and compiles a very small type-safe subset of the C programming language. While the language was small, the test programs were still realistic in their overall structure and complexity. Subsequently we have applied the same idea to a compiler for the full Java language [2]. The Java Touchstone compiler uses exactly the same principles as the one described in our 1998 paper, but the large scale of the programs required numerous changes in the infrastructure. We have learned that it is very important to select an appropriate language for expressing the invariants. We have also learned that it is important to place the invariants in strategic places in order to control the inherently exponential nature of the VCGen algorithm. More specifically, we have learned that a certifying compiler should place invariants not just one for each loop but one for each join point in the control-flow graph. Overall, however, we have verified our original observation that there are few changes necessary to transform a conventional compiler into a certifying one.

We have also demonstrated in that a large number of compilation errors can be caught cheaply by using the certification approach. For example, we have found errors in

the compilation of exception-handling code that would have been hard to find by testing. But we have also learned that there are limits to the kinds of errors that are detected if we check only the type safety of the output. Most notably we have noticed that many compilation errors involving the use of the floating point unit on the Intel x86 lead to incorrect yet type-safe computation. An illustrative example is a bug that incorrectly swaps the order of operands in a subtraction; the resulting code is obviously not correct yet it is still well-typed. For those kinds of errors we must rely on traditional compiler debugging techniques.

In a subsequent paper [8] we extended the ideas of a certifying compiler to go beyond type safety and to verify full correspondence of the output code with the source one. We applied the technique of symbolic evaluation (or verification condition generation) followed by theorem proving to verify the correctness of compilations for the GNU C compiler (`gcc`). We have shown that these techniques can be used to verify the majority of the optimizations performed by `gcc` and that these techniques scale to certifying the compilation of large bodies of code such as the code the the `gcc` compiler itself or that of the Linux kernel. This line of work has similar goals with the concurrent work on translation validation by Pnueli, Siegel, and Singerman [11].

The idea of certifying compilation has now been developed further by several research groups. In a continuation of the TIL approach to certifying compilation, Morrisett, Walker, Cray and Glew showed [5] how to implement a certifying compiler that uses a successively refined type system culminating with a typed assembly language (TAL). The appeal of such an approach is that it is able to use type theory. The disadvantage is that certain compiler optimizations are hard or impossible to express in a traditional type system. This has prompted the TAL team to develop several novel refinements of the TAL type system, for example to endow it with the ability to handle stack locations [4], or with the ability to keep track of aliasing information [12]. Hornof and Jim have since extended the TAL type system with constructs for run-time code generation [3]. Also of note is the project by Appel, Felten, and Shao [1], which continues to investigate scaling issues in certifying compilation.

The closest related work in the commercial domain is Sun Microsystems' Kilo Virtual Machine with the Connected Device Configuration (KVM/CLDC) for Java. The KVM is a virtual machine that is designed for small mobile devices. One of its features is that it uses a simpler and faster byte-code verifier than regular JVM's, partly due to a design decision that requires Java compilers to produce typing loop invariants that can be verified easily, just like in the Touchstone certifying compiler.

We believe that the technique of certifying compilation continues to have significant potential. Advanced program analysis tools and domain-specific languages provide practical ways to collect more and more information about source programs. A traditional compiler is designed to focus on the executable content of the source program and to drop all other source-level information such as types or special invariants enforced in a domain-specific language. A certifying compiler on the other hand can translate such information, thereby allowing an external checker to ensure that the compiler preserves critical properties. This allows a mistrustful recipient of the compiled code to ensure that the code has certain required invariants. But for this to be possible on a

broad range of applications domains the current approaches to certifying compilation must make the transition to handling other properties than what can be expressed in traditional type systems, for example the resource constraints which we considered in [10]. In the future, we predict that as the range of properties increases, the engineering advantages will be so compelling that all compilers eventually will be certifying to at least some extent.

REFERENCES

- [1] Andrew W. Appel, Edward W. Felten, and Zhong Shao. Scaling proof-carrying code to production compilers and security policies. Technical Report YALEU/DCS/TR-1182, Computer Science Department, Yale University, January 1999.
- [2] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.
- [3] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, December 1999.
- [4] Greg Morrisett, Karl Cray, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.
- [5] Greg Morrisett, David Walker, Karl Cray, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
- [6] George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.
- [7] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.
- [8] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, May 2000.
- [9] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations*, pages 229–243. Usenix, October 1996.
- [10] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *LNCS 1419: Special Issue on Mobile Agent Security*, pages 61–91. Springer-Verlag, June 1998.
- [11] Amir Pnueli, M. Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.
- [12] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer-Verlag, April 2000.
- [13] David Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.
- [14] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.