Proof Carrying Code

George Necula
University of California, Berkeley, USA

In this series of lectures we describe various theoretical and implementation issues related to Proof-Carrying Code. Proof-Carrying Code (PCC) allows a code receiver to verify statically that the code has certain required properties, which are stated in the form of a trusted safety policy. To make this possible the received program is accompanied by a representation of an easily-checkable formal proof of compliance with the safety policy.

We will concentrate in the first part of the lectures on mechanisms that we use on the code receiver side to check that the proofs that accompany the code witness the compliance of the code fragment in question with the desired safety policy. For this part we will adapt techniques from program analysis and program verification, namely symbolic evaluation and verification condition generation.

The proof-representation problem is both essential for the practical applicability of PCC and also a great opportunity to exploit results from logic and type theory research. We will discuss two possible solutions. First, we discuss the use of logical frameworks (specifically, the Edinburgh Logical Framework), in which the judgments-as-types and derivations-as-terms representation strategies are used so that proofs can be checked with a simple and largely logic-independent type checker. Then I will describe how further refinements of these techniques can decrease the proof size by a large factor, making the whole process quite practical.

The second proof representation strategy that we will discuss uses techniques from logic programming to obtain even more efficient representations for proofs in many commonly occurring cases. In this approach the receiver of the code describes the inference rules of the safety policy as a logic program whose goals express basic safety issues such as the accessibility of a memory location or the typing of a value. The logic program is interpreted using a non-deterministic logic interpreter. As in other instances in computer science non-determinism gives us power without sacrificing simplicity and efficiency. The last piece of the puzzle is that the code to be checked is required to provide an oracle to guide the interpreter to the solution. The oracle here plays the same role as the proof in previous representations of PCC, but in many cases it is much smaller than the proof.

The last part of the series of lectures is dedicated to the code-producer side of a PCC system. We will discuss the design or a certifying compiler for Java, which in the process of compiling Java to machine code keeps track of enough typing information so that it can construct a proof of well-typedness of the machine code resulting from the compilation. And we will show that this can be done even in the presence of optimizations. The lectures will conclude with a demonstration of a complete PCC system for Java which will allow us to conclude that by relying on techniques developed in logic and type theory we can achieve not only elegance but also efficiency, and even in very practical settings.

References

1. R. Harper, F. Honsell, G. Plotkin. *A Framework for Defining Logics*, Journal of the Association for Computing Machinery, 40:1, pp:143-184, 1993
   `http://www.cs.cmu.edu/~fp/elf-papers/jacm93.dvi.Z`
2. G.C. Necula, P. Lee. *Safe, Untrusted Agents using Proof-Carrying Code*, LNCS 1419: Special Issue on Mobile Agent Security, ed. G. Vigna, Springer, pp: 61-91, 1998
3. G.C. Necula, S.P. Rahul. *Oracle-Based Checking of Untrusted Programs*, 28th Annual ACM Symposium on Principles of Programming Languages, pp: 142-154, ACM, 2001
4. C. Colby, P. Lee, G.C. Necula, F. Blau, M. Plesko, K. Cline. *A certifying compiler for Java* ACM SIGPLAN Notices, 35:5, pp: 95-107, 2000
   `http://www.acm.org/pubs/articles/proceedings/pldi/349299/p95-colby/p95-colby.pdf`
5. B.-Y.E. Chang, A. Chlipala, G.C. Necula, R.R. Schneck. *The Open Verifier Framework for Foundational Verifiers*, Proc. of the 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation, 2005