What Are We Trying to Prove? Reflections on Experiences with Proof-Carrying Code

Peter Lee

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3891

Abstract. Since 1996 there has been tremendous progress in developing the idea of *certified code*, including both proof-carrying code (PCC) and typed assembly language (TAL). In a certified code framework, each program (which is usually in machine-code binary form) comes equipped with a certificate that "explains", both rigorously and in a manner that is easily validated, why it possesses a formally specified security property. A substantial amount of the research work in this area has been directed towards the problem of how to make certified code a practical technology—what one might call "proof engineering". Thus, many of the advances have been in methods for representing the certificates in the most compact and efficiently checkable way. For example, early on George Necula and I used LF encodings of loop invariants and safety proofs, which were then validated by a process of verification-condition generation and LF type checking. Later, we adopted the so-called "oracle-based" representation, resulting in certificates that imposed a much smaller overhead on the size of the object files, usually much less than 20%. This made us optimistic enough about the prospects for a practical realization of certified code that we devoted considerable effort in developing the SpecialJ system for certified Java. And very recently, many researchers (led initially by Andrew Appel and Amy Felty) have been developing various "foundational" approaches to certified code. These hold out the promise of making PCC and TAL even more trustworthy and easier to administer in practical settings, and in some instances may also open PCC to verification of temporal properties.

In this talk, I will start with an overview of these and other current state-of-the-art concepts in certified code. Then, I will consider a very different but equally practical question: Just what is it that we are trying to prove? In contrast to the concept of translation validation, a certified code system does not prove the semantic equivalence between source and target programs. It only proves a specified safety property for the target program, independent of any source program. So then what is the right safety property to use? Is it necessary to prove that the target programs preserve all of the typing abstractions of a particular source program, or is simple "memory safety" enough? While I have yet to claim an answer to this question, I am able to relate specific situations from practical applications that may help to shed some light on the situation.

H. Garavel and J. Hatcliff (Eds.): TACAS 2003, LNCS 2619, p. 1, 2003.