Analysis of Arithmetical Congruences on Low-Level Code

Stefan Bygde
Mälardalen University
Västerås, Sweden
stefan.bygde@mdh.se

Abstract

Abstract Interpretation is a well known formal framework for abstracting programming language semantics. It provides a systematic way of building static analyses which can be used for optimisation and debugging purposes. Different semantic properties can be captured by so-called abstract domains which then easily can be combined in various ways to yield more precise analyses. The most known abstract domain is probably the one of intervals. An analysis using the interval domain yields bindings of each integer-valued program variable to an interval at each program point. The interval is the smallest interval that contains the set of integers possible for that particular variable to assume at that program point during execution. Abstract interpretation can be used in many contexts, such as in debugging, program transformation, correctness proving, Worst Case Execution Time analysis etc.

In 1989 Philippe Granger introduced a static analysis of arithmetical congruences. The analysis is formulated as an abstract interpretation computing the smallest (wrt. inclusion) congruence (residue) class that includes the set of possible values that that variable may assume during execution. The result of the analysis is a binding of each integer-valued variable at each program point to a congruence class. Applications for this analysis include automatic vectorisation, pointer analysis (for determining pointer strides) and loop-bound analysis (for detecting loops with non-unit strides). However, in the original presentation, the analysis is not well suited to use on realistic low-level code. By low-level code we mean either compiled and linked object code where high-level constructions has been replaced with target-specific assembly code, or code in a higher-level language written in a fashion close to the hardware. A good example of low-level code is code written for embedded systems which often is using advantages of the target hardware and/or using a lot of bit-level operations. Code for embedded systems is an increasingly important target for analysis, since it is often safety-critical. The reason that the congruence domain in its original presentation is not suitable for low-level code is mainly due to the three following properties of low-level code: A) Bit-level operations are commonly used in low-level code. Programs that contain bit-operations are not supported in the original presentation. For any computation of an expression which contain operations that has not been defined in the analysis, it has to assume that nothing is known about the result and assign the result to the largest congruence class (equal to $\mathbb{Z}$). This can potentially lead to very imprecise analysis results. B) The interpretation of the values of integer-valued variables is not obvious (e.g. they can be signed or unsigned), the original presentation assumes that values has unambiguous representations. C) The value-domain is limited by its representation (integers are often represented by a fixed number of bits). In Grangers presentation integer-valued variables are assumed to take values in the infinite set of integers. Our contribution is to extend the theory of the analysis of arithmetical congruences to be able to handle low-level or assembly code, still in the framework of abstract interpretation.

This paper provides accurate definitions to the abstract bit-operations AND,NOT,XOR, left- and right shifting and truncation for the congruence domain in order to make the domain support these operations. We provide definitions for the operations together with proofs of their correctness. In these definitions care has been taken to the finite, fixed representation of integers as well as their sometimes ambiguous interpretations as signed or unsigned. With these definitions, congruence analysis can efficiently be performed on low-level code. The paper illustrates the usefulness of the new analysis by an example which shows that variables keep important parity information after executing a XOR-swap.