Applications of Static Slicing in Cost Analysis of Java Bytecode

Samir Genaim[⋆]

CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain
`genaim@fi.upm.es`

Java bytecode [?] is a low-level object-oriented language which is widely used in the context of mobile code due to its security features and the fact that it is platform independent. Recent works study advanced properties of Java bytecode like cost analysis [?] or termination [?]. Automatic cost analysis has interesting applications in the context of Java bytecode. For instance, the receiver of the code may want to infer cost information in order to decide whether to reject code which has too large cost requirements in terms of computing resources (in time and/or space), and to accept code which meets the established requirements [?,?,?]. Also, in parallel systems, knowledge about the cost of different procedures in the object code can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

Given an input program, cost analysis aims at inferring *Cost Equations Systems* (CES) which define the cost of the program as a function of (some of) its data input size. CES are a general form of describing the resource consumption of programs and, in a particular application, they can be used to infer heap consumption [?], number of executed bytecode instructions [?], etc. Essentially, CES are generated by abstracting the structure of the program such that when the program contains an iteration the CES contains a corresponding recursion, which in addition includes information about how the sizes of the different variables change when the program goes through this recursion. The traditional approach is to infer upper bounds of the cost by solving the CES, for instance using Computer Algebra Systems (CAS) like Mathematica, Maple, etc.

Cost Analysis of Java bytecode presents some peculiar features which stem from its unstructured and object-oriented nature:

a) loops originate from different sources (conditional and unconditional jumps, method calls, or even exceptions);
b) *size measures* must consider supported data types (primitive types, objects, and arrays);
c) data can be stored in local variables, operand stack elements or heap locations.

The approach to cost analysis as described in [?] makes two initial steps in order to obtain a structured representation for the bytecode:

[⋆] Joint work with Elvira Albert, Puri Arenas, German Puebla and Damiano Zanardini

1. The first step consists in constructing a control flow graph from the bytecode which makes all implicit branching explicit. A control flow graph consists of *guarded* basic blocks and edges which describe how control flows between blocks. Basic blocks are sequences of non-branching bytecode instructions, and edges are obtained from instructions which might branch such as virtual method invocation, conditional jumps, exceptions, etc;
2. In the next step, the CFG is represented in a procedural way by means of an *intermediate representation.* This representation consists of a set of *guarded rules* which are obtained from the blocks in the CFG. A principal advantage is that all possible forms of loops in the program are represented now in a uniform way (feature a) above).

In principle, each rule of the intermediate representation contains the following arguments: (1) the corresponding method's local variables; (2) the *active stack* elements at the block's entry and exit, i.e., the stack elements are considered as local variables; and (3) a single variable which corresponds to the method's return value. Therefore, all data is represented in a uniform way (feature b) above).

In the last step, size analysis is applied to the above intermediate representation and a CES – involving the above arguments – is generated from such representation (feature c) above).

After performing cost analysis, the CES is traditionally solved using existing CAS. An important observation is that many of the above arguments may not be relevant to the cost. For instance, typical *accumulating* parameters which merely keep the value of some temporary result do not affect the control flow nor the cost of the program. Such tools are more likely to fail in providing useful information if the corresponding CES includes information which originates from the program's parts that does not affect its cost. Therefore, eliminating those superfluous parts from the CES is crucial in practice. This elimination problem can be formalized as a *backward slicing* problem.

Basically, given a rule, the arguments which can have an impact on the cost of the program are those which may affect directly or indirectly the program guards (i.e., they can affect the control flow of the program), or are used as input arguments to external methods whose cost, in turn, may depend on the input size. The problem of computing a safe approximation of these arguments can be formalized as a backwards slicing problem using the reachable guards and external methods as the slicing criterion [**?**].

In the talk, we discuss several aspects of the role of *static slicing* to minimize the number of arguments which need to be taken into account in CES. We will focus on two benefits of eliminating the arguments which do not have an impact on the cost of the program. On one hand, analysis can be more efficient if we reduce the number of variables. And also CES are more likely to be solved by standard CAS. In addition, we discuss when slicing should be performed. We consider the scenarios of (1) applying it on the intermediate representation (2) versus applying it on the CES.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.