COSTA: A Cost and Termination Analyzer for Java Bytecode

(Tool Demo)

E. Albert[1] P. Arenas[1]
S. Genaim[2] G. Puebla[2] D. Zanardini[2]

[1] *DSIC, Complutense University of Madrid,* {elvira,puri}@sip.ucm.es
[2] *Technical University of Madrid,* {german,samir,damiano}@clip.dia.fi.upm.es

Abstract

This paper describes COSTA, a COSt and Termination Analyzer for Java bytecode. The system receives as input a bytecode program and a selection of a *resources* of interest, and tries to bound the resource consumption of the program with respect to such a *cost model*. COSTA provides several non-trivial notions of resource, as the consumption of the heap, the number of bytecode instructions executed, the number of calls to a specific method (e.g., the library method for sending text messages in mobile phones), etc. The system uses the same machinery to infer *upper bounds* on cost, and for proving *termination* (which also implies the boundedness of any resource consumption). The demo will describe the architecture of COSTA and show it on a series of programs for which interesting bounds w.r.t. the above notions of resource can be obtained. Also, examples for which the system cannot obtain upper bounds, but can prove termination, will be presented.

Keywords: Cost analysis, Termination analysis, Resource usage analysis, Java bytecode.

1 Introduction

Research about *resource usage* goes back to the seminal work by Wegbreit in 1975 [23], which proposes to analyze the performance of a program by deriving a mathematical expression which represents its runtime behavior. In a nutshell: given an input program, *cost analysis* works as follows: (1) in the first step, it generates an associated *cost equation system* (CES) from the program, which captures the relation between the different parts of the code. CESs are sets of recurrence equations which express the cost of a program in terms of the size of its input arguments. (2) In the second step, CESs can be often solved (or approximated) by typically relying on algebraic techniques, thus obtaining a *closed form* (e.g., without recurrence) solution or *upper* (or *lower*) *bound* for it.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

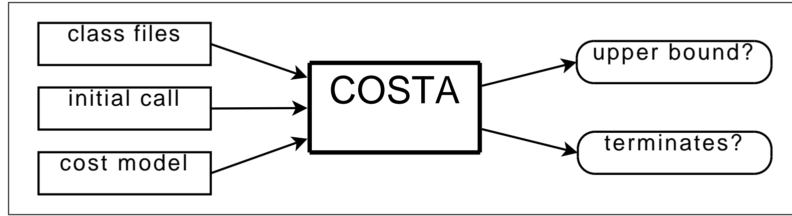

Fig. 1. Input/Output of the COSTA System

On the other hand, *termination analysis* can be classified within the category of cost analyses since termination implies that the resource consumption of a program is finite. Indeed, when a resource analyzer finds an upper bound on certain measure of resource consumption as for example the number of execution steps, then termination is directly entailed. Not surprisingly, techniques used to find upper bounds on cost are very related to those used in termination analysis, and much of the technology used by termination analyzers is also needed in cost analysis. Therefore, it is useful to have both analyses integrated in the same tool.

There exist a good number of cost usage analysis frameworks for a wide variety of programming languages, including logic [17,9], functional [18,20,11,5] and imperative languages [2,13,5,16]. In all cases, the steps described above are adapted to the specific features of the language. Moreover, several termination analyzers have been developed in the last decades [14,6,7,10]. In spite of such large amount of work in the area, the application of resource usage analysis to a *realistic* programming language, and to programs with a realistic size and complexity, is still, in many cases, an open issue, and there is a lack of resource usage analysis tools.

This paper presents costa, a tool which is, to the best of our knowledge, the first resource usage analyzer for an object-oriented, stack-based, low-level programming language. The system works on Java bytecode programs [15], a language which is widely used nowadays in a large number of applications. In the rest of the paper, we discuss the design and implementation of the components which are included in costa, and provide a brief outline of the planned tool demonstration.

2 An informal description of the system

Figure 1 shows a schema of costa, where the input and output of the system is made visible. The analyzer is written in Prolog and relies on the theory of *Abstract Interpretation* [8]. It uses some external libraries, such as the *Parma Polyhedra Library* [4] and the *CU Decision Diagram Package* [21], for implementing some of the underlying abstract domains (polyhedra and decision diagrams).

The input to the analyzer consists of three parts.

- A set of *Java Bytecode classes*, in the form of `.class` files.
- The *signature of the initial method*, as well as an abstract description of the *input* arguments. The description of the input is basically information about the abstract values of the arguments w.r.t. some of the abstract domains (e.g., that a variable is non-null, or that its size is greater than zero, etc). This can definitely help the rest of the analysis.

- The *cost model* of interest, either user-defined or among those available in the system. The system includes the definition of a set of useful cost models, such as the number of executed bytecode instructions and the memory consumption, and it is easy to define new cost models.

For each reachable method, the analyzer outputs, when possible, a closed form function which is an upper bound of its actual cost with respect to the selected cost model. Moreover, it is reported whether the analyzer has been able to prove termination. COSTA consists of several generic fixpoint engines, either goal-directed or goal-independent, which can be used to implement different abstract interpretation based analyses by providing the corresponding abstract operations.

We start by presenting the notion of cost model as it is formalized in the system. Afterwards, we describe the basic components of COSTA and the main steps of the analysis.

The Notion of Cost Model

This paragraph gives an informal account of how the notion of *cost* has to be interpreted. First, bytecode instructions are assumed to be *annotated* with their input and output variables. For example, $iadd(a, b, c)$ is the annotated version of $iadd$, where a and b are the input variables (i.e., the last two elements of the stack), and c is the output variable (i.e., the last element of the stack after the input has been popped).

A *cost model* M is a mapping from annotated bytecode instructions to *cost expressions*, which are non-negative symbolic arithmetic expressions defining the cost of executing the corresponding bytecode instruction [**?**]. Executions of sequential Java bytecode programs can be regarded as *traces*. Execution begins in an initial state S_0. According to the particular instruction in the current program counter S_i, another state S_{i+1} is obtained, and we say that there is a *transition* between S_i and S_{i+1}. We use $S_i \rightsquigarrow_{b_i} S_{i+1}$ to denote that, by applying the (annotated) instruction b_i, which is the current one at S_i, the state S_{i+1} is obtained. The cost of a *trace* $S_0 \rightsquigarrow_{b_0} \cdots S_n \rightsquigarrow_{b_n} S_{n+1}$ is simply the sum of the cost of all transitions, i.e. $\sum_{i=0}^{i=n} M(S_i)$, where the state S_i also contains information about the next instruction b_i to be executed.

The COSTA analyzer aims at approximating the cost with respect to a given cost model and an initial state, or proving the absence of infinite traces (i.e., the termination of the program).

As already mentioned, useful cost models include:

- the *number of executed instructions*, which gives cost 1 to every transition, and approximates the length of the execution trace;
- the *heap consumption*, which approximates the amount of heap used at execution time [3];
- the *number of method calls* for a given method, which is useful when the focus is on a specific functionality of the program (e.g., the method sending text messages in a mobile application).

It is important to note that proving the termination of a program implies a finite

bound on the cost for any reasonable cost model. However, the inverse direction does not hold, since a non terminating program could even have a null cost with respect to some cost measure (for example, the heap consumption for code which has a non-terminating loop, but does not even use the heap). This inverse direction holds, however, for cost models which always assign infinite cost to infinite traces. This is the case of models where any transition has a non-zero cost. On the other hand, in the heap consumption cost model, or in the one counting method calls, there are transitions with cost zero, and it is possible to have infinite traces which only involve such no-cost transitions.

From the Java Bytecode to the Control Flow Graph

The *control flow* of a program is essential when reasoning about its cost and termination, since it provides valuable information about the possible paths which may be taken during the execution, and about the parts which are executed iteratively. Java Bytecode is a low-level, object-oriented programming language. Unlike high-level, structured programming languages such as Java, it has an *unstructured* nature, so that it may impose a rather complicated control flow: conditional and unconditional jumps are allowed, as well as virtual method invocation and exception handling. In this step, the analyzer recovers the structure of the bytecode program by means of a set of *control flow graphs* (CFGs for short), each one consisting of basic blocks (containing sequential instructions) and conditional edges, describing the condition under which control can go from the end of one block to the beginning of another block. These edges stem from explicit (jumps, etc.) and implicit (exceptions) conditions in the original program.

The use of *virtual invocation* implies the need to consider more than one method for execution at a given program point. In practice, computing a precise approximation of the *reachable* methods is not trivial, and asking the user to provide such information is not practical. To this end, COSTA uses *Class Analysis* in order to precisely approximate this information, as follows. First, the CFG of the initial method is built, and class analysis is applied in order to approximate the possible runtime classes of each reference variable at each program point. Afterwards, this information is used to decide which methods may be called for each virtual invocation. Such methods are loaded and their corresponding CFGs are constructed. Class analysis is applied to their body, and the process continues iteratively. Clearly, class analysis may compute an over-approximation of the set of possible runtime classes, but the approximation is acceptable in most cases. Once a fixed-point is reached, it is guaranteed that all reachable methods have been loaded and their corresponding CFGs have been generated. The use of class analysis to approximate the set of reachable methods is crucial for the overall practicality of the analyzer, especially to analyze methods which are defined in the `Object` class as those found in most libraries, since it drastically reduces the number of methods to be analyzed.

In addition to constructing the CFGs, the analyzer simultaneously computes the height of the *operand stack* and the types of its elements (reference, integer, etc.) at each program point. This can be done statically by means of standard techniques,

as in the virtual machine [1]. Information about the stack height is used in order to represent stack elements as local variables, and type information helps to restrict the underlying static analysis (the part approximating the *heap*) only to reference variables, thus improving performance.

Moreover, the user can choose to perform an additional analysis of the code, which tries to extract loops into separate CFGs. *Loop extraction* is implemented efficiently using algorithms which have been developed in the context of decompilation [22]. It is essential to improve the accuracy of both cost and termination when the program involves *nested loops*.

From the Control Flow Graph to the Rule-Based Language

In order to facilitate cost and termination analyses, CFGs are transformed into an intermediate language, referred to as *rule-based representation*, which has a simpler control flow: (1) there is only one iterative construct, implemented by means of *recursion*; (2) there is only one conditional construct, implemented by means of *guarded rules*; (3) there is only one kind of variable (the *local variables*), and no stack (it is transformed into additional local variables); (4) there is no virtual method invocation; and (5) the heap is similar to the one of Java Bytecode. A rule-based program consists of a set of *procedures*. Each procedure p is defined by a set of mutually exclusive *rules* of the form $p(\bar{x}, \bar{y})::=g, b_1, \ldots, b_n$, where p is the procedure name, $\bar{x}$ are the input variables, $\bar{y}$ are the output variables, g is a Boolean condition (guard) under which the rule can be applied, and each b_i is either an annotated bytecode instruction (see above) or a call (by value) to another procedure. Note that bytecode instructions are annotated by using the stack height information previously computed. Briefly, when calling a procedure p, the rule to be executed is the (unique) one whose guard is satisfiable (the rules of a procedure are mutually exclusive), and its body is executed sequentially from left to right.

Obtaining the rule-based representation which corresponds to a CFG is done by generating one rule for each CFG block. The rule contains annotated bytecode instructions and possibly calls to other rules which correspond to method invocations in the bytecode of the corresponding block. Moreover, the rule contains the call to a *continuation* procedure, defined as a set of guarded rules which decide which rule the control should be transferred to. Guards of the continuation rules originate from the corresponding conditional edges in the graph. It is possible to provide an *operational semantics* for the rule-based representation which is equivalent to that of the original bytecode program. Furthermore, given an execution trace at the rule-based level, it is straightforward to obtain the trace which the JVM semantics would produce for the same initial call and input values. Therefore, thanks to this straightforward correspondence between traces at the rule-based level and the JVM level, we can develop our analysis at the simpler rule-based level and produce results about the cost of execution and termination at the JVM level.

Note that a similar approach could be done for other related programming languages, such as .NET, by proposing a translation into our rule-based representation

[1] Note that the bytecode is supposed to be a valid one, i.e., the stack height and the type of operands are consistent with respect to the static checks of the virtual machine.

together with a mapping from traces at the rule-based level to the .NET level.

Inferring linear relations between the (abstract) values of program variables

In order to bound the cost of a rule-based program or prove its termination, it is required to infer information which approximates how many times the program goes through its loops (i.e., the number of times each rule may be traversed). This is done in COSTA by an abstract interpretation based analysis which infers linear relations between the program variables with respect to the following *abstractions*:

- The abstract value for a reference variable represents the length of the maximal *path* reachable from that reference [12] by dereferencing. This is used to handle loops which traverse structures on the heap;
- The abstract value for an array reference variable represents the length of the array. This is used to handle loops which traverse the elements of arrays;
- The abstract value for an integer variable corresponds to its possible concrete values [8]. This is used to handle loops with termination conditions on integer variables, and to approximate the values of the variables which are used in the cost model.
- The remaining variables are abstracted to *unknown*.

The analysis consists of two steps. In the first step, a static-single assignment transformation (SSA) is applied to the rules, and each guard and bytecode instruction is replaced by the linear constraints it imposes on its variables. E.g., $iadd(s_0, s_1, s_0)$ results in the constraint $s_0' = s_0 + s_1$ (meaning that the output s_0', whose name is obtained by SSA, is the sum of the inputs); $getfield(f, s_0, s_0')$ is abstracted, as long as s_0 is not cyclical, into $s_0 > s_0'$ (meaning that the maximal path reachable from $o.f$ is shorter than the maximal one reachable from o); and the guard $gt(s_0, s_1)$ results in the constraint $s_0 > s_1$. This step results in an *abstract constraint program* which approximates the behavior of the original program with respect to the above abstractions. In the second step, a fixpoint computation is applied to infer the input-output denotations for each rule. This *abstract compilation* relies on several pre-analyses, such as (i) *cyclicity* analysis [19], which is required for path-length (see above the abstraction of *getfield*); (ii) *sign* and *constant propagation* analysis, which are needed to improve the precision of division operations; and (iii) *nullity* analysis, improving the analysis of reference values. In addition, annotations (relative to the chosen abstract domains) for *external code* can be incorporated in this step, provided it is guaranteed that such external code will not generate any *call back* to the user code.

Generating Cost Equation Systems

The next step consists in setting up a *cost equation system* (CES) which captures the cost of the rule-based program and its termination behavior in terms of the input values. A CES is a set of equations of the form:

$$p_0(\bar{x}_0) = exp + p_1(\bar{x}_1) + \cdots + p_n(\bar{x}_n), \varphi$$

where φ is a conjunction of linear constraints describing the relation between the values of the variables, and *exp* is a cost expression. The above equation states that, for given (abstract) values $\bar{x}_i = \bar{v}_i$ such that $\varphi \models \wedge \bar{x}_i = \bar{v}_i$, a possible cost for $p_0(\bar{v})$ is $exp[x_i \mapsto v_i]$ plus the sum of the costs of $p_1(\bar{v}_1) \cdots p_n(\bar{v}_n)$. For each rule in the rule-based representation, the analyzer generates a corresponding cost equation by using the abstract rule to generate φ, and the original rule together with the selected cost model to generate *exp* (i.e., the cost expression has to represent the cost of the bytecodes in the rule w.r.t. the model). The differences between a CES and a recurrence equation systems are explained in [**?**]. Basically, a CES is a *non-deterministic constraint functional program.* Non-determinism might appear due to the loss of precision inherent to (static) size analysis. This means that, for given input values $\bar{v}_0$, the query $p_0(\bar{v}_0)$ may result in several solutions. Yet, it is guaranteed that (1) one of the solutions corresponds to the actual cost of the rule-based program; and (2) if $p_0(\bar{v}_0)$ has a finite number of solutions and does not lead to any infinite computation, then the original bytecode program terminates for the corresponding concrete input. More details can be found in [**?**].

Inferring Upper Bounds and Proving Termination from CES

Unless a *closed form solution* which describes the cost of a program only in terms of its input variables (i.e., with no references to other equations) is found, cost equation systems turn out not to be practical. Due to the *non-decidable* and *non-deterministic* features of CESs, in most cases, it is not possible to obtain an exact solution (see [**?**]). Therefore, the aim is to obtain *non-asymptotic* [2] upper (or lower) bounds. COSTA includes a practical CES solver [**?**], which, for a given CES, computes a non-asymptotic closed form upper bound. The solver handles a large set of complexity classes, such as logarithmic, linear, polynomial, and exponential.

In addition to the cost of a program, the focus of the analysis may also be on its termination. As already mentioned, proving termination involves guaranteeing that a *finite upper bound* for the system exists, even if it cannot be found explicitly. Our system incorporates techniques used in constraint logic programming, as described in [1], to prove termination on the above non-deterministic constraint functional representation, which in turn implies termination of the Java bytecode program.

Plan of the Demo

The demonstration of COSTA will basically go through some examples which illustrate how the analyzer works. In particular, it will be shown how COSTA automatically deals with the analysis, from the original bytecode program to the cost upper bound or the termination behavior, without user intervention. We will show the definition of the current cost models that the system incorporates. Significant examples with different cost models will be provided, which will shed light on the capability of the system to deal with differently structured programs and different notions of resource consumption. Moreover, the demo will include an example where an upper bound cannot be found, but termination can be proven by guaranteeing

[2] I.e., which hold *for every input value*, not only *for values greater than a threshold*.

that such a bound exists. This shows how most parts of costa are common to both analyses, and how, on the other hand, termination and cost can pose different problems to static analyzers.

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 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.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *9th International Workshop on Termination, WST'07*, June 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[4] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at http://www.cs.unipr.it/Publications/. Also published as arXiv:cs.MS/0612085, available from http://arxiv.org/.

[5] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[6] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, Tucson, Arizona, USA, 2006.

[8] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[9] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[10] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with aprove. In *Proc. International Conference on Rewriting Techniques and Applications (RTA)*, pages 210–220, Aachen, Germany, 2004.

[11] G. Gómez and Y. A. Liu. Automatic time-bound analysis for a higher-order language. In *PEPM*. ACM Press, 2002.

[12] Patricia M. Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[13] J. Jouannaud and W. Xu. Automatic Complexity Analysis for Programs Extracted from Coq Proof. *ENTCS*, 2006.

[14] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *Proc. International Conference on Logic Programming (ICLP)*, pages 63–77, Leuven, Belgium, 1997.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[16] J.-Y. Marion and R. Pèchoux. Resource control of object-oriented programs. In *International LICS affiliated Workshop on Logic and Computational Complexity (LCC 2007)*, Wroclaw, Poland, 2007.

[17] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP 2007)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[18] M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[19] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI*, volume 3855 of *LNCS*. S-V, 2006.

[20] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[21] F. Somenzi. *CUDD: CU Decision Diagram Package*, 2005. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.

[22] W. Zou T. Wei, J. Mao and Y. Chen. A New Algorithm for Identifying Loops in Decompilation. In *Proc. of SAS'07*, LNCS, 2007.

[23] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.