COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode

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

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. This paper describes the architecture of COSTA, an abstract interpretation based COSt and Termination Analyzer for Java bytecode. The system receives as input a bytecode program, (a choice of) a *resource* of interest and tries to obtain an upper bound of the resource consumption of the program. COSTA provides several non-trivial notions of cost, as the consumption of the heap, the number of bytecode instructions executed and the number of calls to a specific method. Additionally, COSTA tries to prove *termination* of the bytecode program which implies the boundedness of any resource consumption. Having cost and termination together is interesting, as both analyses share most of the machinery to, respectively, infer cost *upper bounds* and to prove that the execution length is always *finite* (i.e., the program terminates). We report on experimental results which show that COSTA can deal with programs of realistic size and complexity, including programs which use Java libraries. To the best of our knowledge, this system provides for the first time evidence that resource usage analysis can be applied to a realistic object-oriented, bytecode programming language.

1 Introduction

Research about *automatic cost analysis* goes back to the seminal work by Wegbreit in 1975 [29], which proposes to analyze the performance of a program by deriving *closed-form* expressions for its execution behavior. This approach consists of two phases.

(1) In the first phase, given a program and the description of some *cost measure*, a set of equations is produced, which captures the cost of the program in terms of the *size* of its input data. Such equations are generated by converting the iteration constructs (loops and recursion) of the program into recurrence, and by inferring *size relations* which approximate how the size of arguments varies between calls. This set of equations can be regarded as a set of *Recurrence Relations* (RR for short).
(2) The aim of the second phase is to obtain a *non-recursive* representation (solution) of the equations, known as *closed-form* solution. In most cases, it is not possible to find an exact solution, and the closed-form corresponds to an upper bound.

There are a good number of cost analysis frameworks for a wide variety of programming languages, including functional, logic and imperative [23, 14, 3]. Despite such a large amount of work, applying cost analysis to *realistic* languages, and programs with realistic size and complexity, is still an open issue, and, there is a lack of working tools.

Termination analysis [10, 19] can be regarded as another kind of resource usage analysis, and it has also been studied in the context of several programming languages. Termination analysis tries to prove that a program cannot infinitely run by considering its iterative and recursive structures and by proving that the number of times they can be executed in a program run is bounded. Putting cost and termination analysis together in the same tool makes sense because of the tight relation between them: proving termination implies that the amount of resources used at runtime is finite. In practical terms, cost and termination analysis share most of the system machinery, as they need to consider and infer roughly the same information about the program. We will use the term *resource usage analysis* (RUA) to refer to either cost or termination analyses.

The present paper describes the design and implementation features of COSTA, a tool which is, to the best of our knowledge, the first RUA tool for an object-oriented, stack-based programming language, namely, *Java bytecode* [21]. The goal of the system is to infer the cost of a program with respect to some cost measure, and to prove its termination. COSTA sets up an accurate RR from the bytecode in an efficient way (phase 1 above) and is connected to a termination prover [1] and to an upper bound solver [2] to carry out phase 2. COSTA can currently work with different *cost models*, formalizing the idea of what a resource is, and how it is consumed at runtime: the *number of instructions*, $\mathcal{M}_{inst}$; the *heap consumption* in bytes, $\mathcal{M}_{heap}$; the *number of calls* to a given method, $\mathcal{M}_{calls}$ (e.g., the library method for sending text messages in mobile phones).

The system allows the user to decide whether the analysis has to consider *libraries* as part of the analyzed program, i.e., if it must go and analyze the library code, or take cost information in the form of *cost interfaces*. Interfaces are needed when some code is not available, or is written in another language. However, they can only be used if it is guaranteed that the *external* code will not generate *call-backs* to the user code. In the absence of interfaces, the system gives symbolic names to the cost of libraries, and they remain as unknown functions in the upper bound. These options make COSTA a flexible RUA tool, as we show in the next example.

Example 1. Figure 1 shows the Java and bytecode of the running example, whose most relevant feature is the use of Java *libraries*. The Java code at the top is only shown for clarity, since COSTA works directly on the bytecode. At the left-middle, we depict the bytecode of the method inter, which computes the intersection of a linked list l and an array a, both non-sorted and containing objects which implement the interface java.lang.Comparable. The class CompList is user-defined, and implements a linked list of Comparable elements in the standard way. The result of the intersection is stored in a java.util.ArrayList object al. The method main (right-middle) allocates memory for a, l and al by means of their constructors.

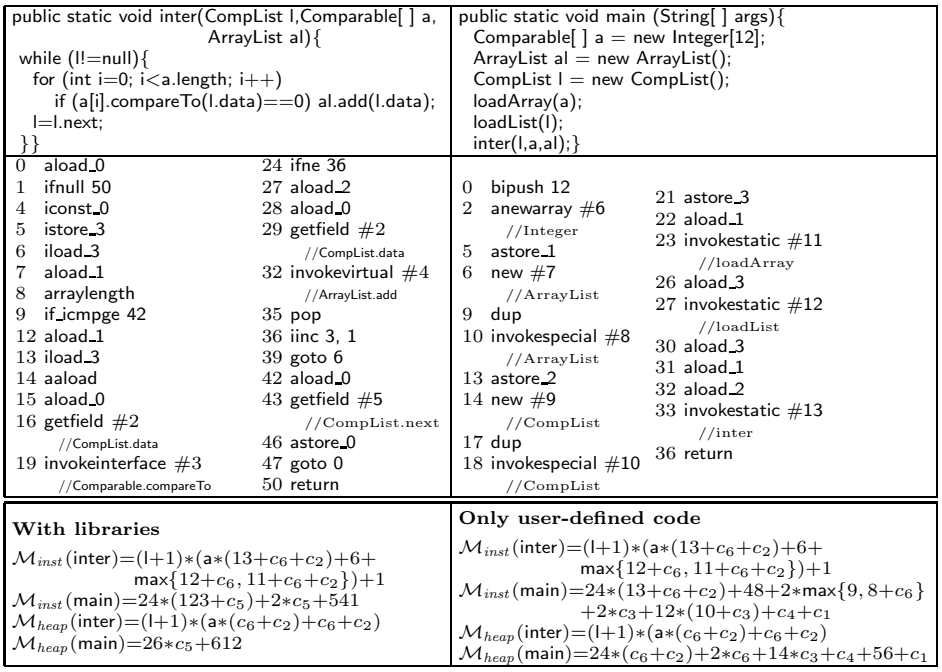

Fig. 1. The running example, with upper bounds computed for different cost models.

Afterwards, calls to static methods loadArray and loadList fill, resp., the array and the list with objects of the library class java.lang.Integer, also implementing java.lang.Comparable. For brevity, the code of both static (user-defined) methods, which terminate and have constant cost, is omitted. Note that parameters a and l of inter are non-null and have constant length, and al is also non-null. At the bottom, we show the upper bounds computed by COSTA for main and inter, with the cost models $\mathcal{M}_{inst}$ and $\mathcal{M}_{heap}$. Variables a and l in the solutions denote, resp., the length of the array a and the maximal *path-length* (Sec. 5) of l. The left column is computed by analyzing all required library methods. In the right column, library methods are not analyzed; instead, their cost appears as $c_1, \ldots, c_6$, where c_1 and c_2 stand for, resp., java.util.ArrayList.ArrayList and java.util.ArrayList.add, and $c_3, \ldots, c_6$ stand for Integer.Integer, Object.Object, System.arraycopy and Comparable.compareTo, all from java.lang. When analyzing libraries, upper bounds for main depend only on the cost of c_5, which corresponds to the *native* Java method arraycopy invoked within ArrayList.add inside inter. When we analyze inter independently of main, c_2 and c_6 are not analyzed, as the objects have not been created and their class is not statically known. When libraries are not considered, c_5 is not reached. While c_1 and c_2 are invoked, resp., in main and inter, c_3 originates from loadList and loadArray, which create Integer objects by invoking their constructor. Due to *inheritance*, c_4 is also required. In $\mathcal{M}_{heap}$, inter does not consume any heap locations by itself, as it does not allocate any object. Yet, analysis considers the heap usage of c_2 and c_6, which could allocate memory. □

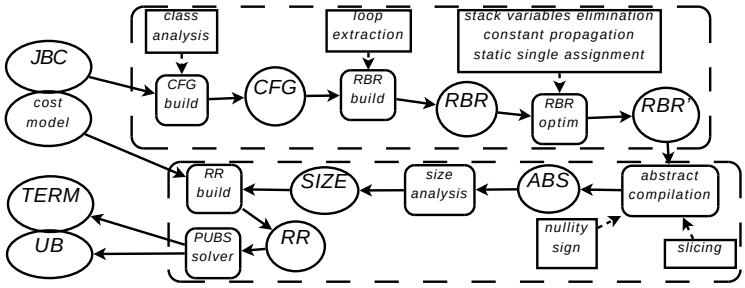

Fig. 2. Architecture of COSTA

2 Architecture of the Cost and Termination Analyzer

Figure 2 shows the overall architecture of the COSTA analyzer. The dashed frames represent the two main phases of the analysis: (i) the transformation of the bytecode into a suitable internal representation; and (ii) the actual resource usage analysis. Input and output of the system are depicted on the left: COSTA takes a Java bytecode program *JBC* and a description of the *cost model*, and yields as output an upper bound *UB* of its cost, and information *TERM* about termination. Ellipses (as *CFG*) represent *what* the system produces at each intermediate stage of the analysis; rounded boxes (as *CFG build*) indicate the *main steps* of the analysis process; square boxes (as *class analysis*), which are connected to the main steps by dashed arrows, denote auxiliary analyses which allow to obtain more precise results or to improve efficiency.

During the first phase, depicted in the upper half of the figure, the incoming *JBC* is transformed into a *rule-based representation* (*RBR*), through the construction of the *control flow graph* (*CFG*). The purpose of this transformation (Sec. 3) is twofold:

(1) to represent the unstructured control flow of the bytecode into a procedural form (e.g., `goto` statements are transformed into recursion); and
(2) to have a uniform treatment of variables (e.g., operand stack cells are represented as local variables).

Several optimizations are performed on the rule-based representation to enable more efficient and accurate subsequent analyses: in particular, *class analysis* is used to approximate the method instances which can be actually called at a given program point in case of virtual invocation; *loop extraction* makes it possible to effectively deal with nested loops by extracting loop-like constructs from the control flow graph; *stack variables elimination*, *constant propagation* and *static single assignment* make the rest of the analyses simpler, more efficient and precise. Essentially, the construction of the RBR turns out to be effective for developing a (compositional) RUA (Sec. 3.5).

In the second phase, depicted in the lower half of the figure, the system performs cost and termination analysis on the RBR. *Abstract compilation*, which is helped by auxiliary static analyses, prepares the input to *size analysis*, whose aim is to find interesting relations between execution states (Sec. 5). As usual in

object-oriented languages, *nullity* analysis improves the accuracy of size analysis, together with *class analysis*, which was performed previously. Finally, *sign* analysis helps in dealing with operations on integers. Afterwards, COSTA sets up a *Recurrence Relation* (RR) for the selected *cost model*. The latter is given as an input, selected among the available models. It is also trivial to define new cost models in the system by just associating a cost to each bytecode instruction. For the purpose of cost, the system performs *slicing* of the RBR in order to remove those variables which are *useless* to cost analysis. Up to this point, phase (1) in Sec. 1 has basically been achieved. In order to deal with phase (2), COSTA integrates the dedicated *upper bound solver* of [2, 1], which finds closed-form solutions for RRs and proves termination (Sec. 6).

3 From the Bytecode to the Rule-based Representation

Unlike other bytecode analyses performed directly on the CFG, in order to study cost and termination, an essential step is to transform the JBC into an appropriate *recursive* rule-based representation. Basically, this will facilitate the task of identifying loops (necessary for termination), and producing a recurrence relation from (the RBR of) the bytecode which represents its cost.

3.1 The Java Bytecode Language

A (sequential) JBC program consists of a set of *class files*, one for each class. A class file contains information about its name, the class it extends, and the fields and methods it defines. Each *method* has a unique *signature* m containing the class where it is defined, its name and its type. The bytecode associated to m is a sequence $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n \rangle$, where each b_i is a *bytecode instruction* and pc_i is its address. Local variables are denoted by $\langle l_0, \ldots, l_{k-1} \rangle$, where l_0 is the *this* reference (explicit in JBC) and $l_1, \ldots, l_n$, with $n < k$, are the formal parameters of m. Similarly, each *field* f has a unique signature, containing its name and the name of the class it belongs to. It is out of the scope of this paper to provide a thorough description of the JVM (see the specification [21] for details).

Example 2. Let us explain some instructions in Fig. 1 related to object-oriented features. Indexes $0, \ldots, 3$ in the bytecode correspond, resp., to parameters l, a, al and the local variable i. As the method is static, there is no *this* reference. The instruction 15: aload_0 pushes the reference to l on the stack. Next instruction, 16: getfield #2;, fetches the field data from l: the top of the stack l is popped, and #2 is used to build an index of the runtime constant pool (RCP) of the class where the reference to the name is stored. When this reference is fetched, it is pushed on the stack. As another example, 19: invokeinterface #3 pops a[i] and l.data from the stack, and searches the *closest method* with the correct signature, by looking up first in the class of the dispatching object, and then going *up* in the inheritance chain. As before, #3 is used to search the name of the method in the RCP. The method result is then pushed on the stack. □

The execution environment of the JVM consists of a *stack of activation records* and a *heap*. Each activation record contains a program counter, a local operand

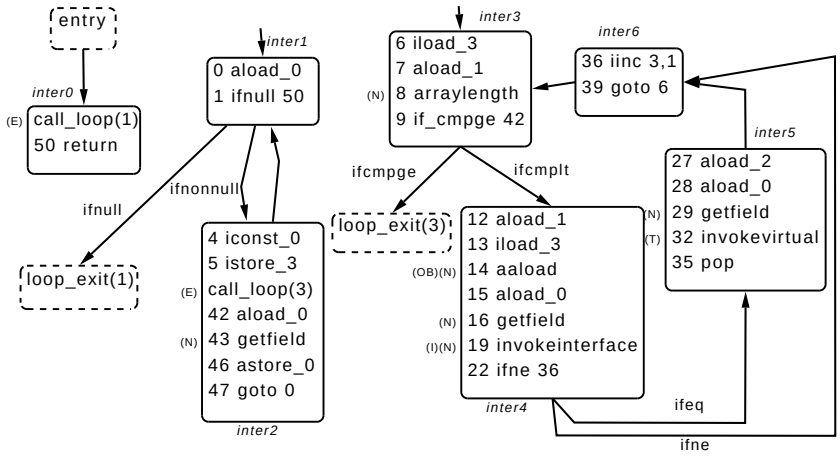

Fig. 3. CFGs for the Java bytecode program in Fig. 1 after loop extraction

stack, and a table of local variables. The heap is a global data structure which contains objects (and arrays) allocated by the program. Each method invocation generates a new activation record according to its signature, number and type of local variables, and maximum size of operand stack. Different activation may contain references to the same objects in the heap.

3.2 Generation of Control Flow Graphs guided by Class Analysis

The control flow of JBC is unstructured. Conditional and unconditional jumps are allowed, as well as other implicit sources of branching such as virtual method invocation and exception throwing. The notion of *Control Flow Graph* (CFG) facilitates reasoning about programs in unstructured languages. COSTA transforms the bytecode of a method into CFGs by using techniques from compiler theory. In particular, the instruction sequence is split into maximal sub-sequences of *non-branching* instructions, which form the *basic blocks* (nodes) of the initial graph. Basic blocks of a method m are given a unique identifier m_i, where i is an index, and are connected by *guarded* edges which describe all possible transitions.

Guarded edges are introduced by considering the last bytecode instruction of each block, and represent the condition for the control going from one block to another one. Edges take the form $\langle m_i \mapsto m_j, \phi_{ij} \rangle$, where m_i and m_j are the source and destination nodes, and ϕ_{ij} is a boolean condition. Branching instructions include conditional jumps, dynamic dispatching and exceptions.

Example 3. Figure 3 depicts the CFGs of method inter (Ex. 1). The edge from inter_1 to inter_2 takes the form $\langle \mathsf{inter}_1 \mapsto \mathsf{inter}_2, \mathsf{ifnonnull} \rangle$, indicating that the top of the stack must be non-null for the control going from *pc* 1 (last instruction of inter_1) to 4 (first one of inter_2). Guards which are always true are left implicit. □

Virtual invocation implies that more than one method can be executed 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. As customary in the analysis of OO languages, COSTA uses *class analysis* [25] in order to precisely approximate this information. First, the CFG of the initial method is built, and class analysis is applied in order to approximate the possible runtime classes at each program point. This information is used to *resolve* virtual invocations. Methods which can be called at runtime are loaded, and their corresponding CFGs are constructed. Class analysis is applied to their body to include possibly more classes, and the process continues iteratively.

Once a fixpoint is reached, it is guaranteed that all reachable methods have been loaded, and the corresponding CFGs have been generated. To handle realistic programs, we implemented a simple class analysis which does not keep class information at the level of reference variables, but just computes the set of reachable classes from any point in the program. This simple class analysis turned out to be 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. The simple class analysis used drastically reduces the number of methods to be analyzed while remaining quite efficient in practice. In the running example, class analysis detects that only one instance (the one in java.lang.Integer) of compareTo and add can be called resp. at bytecodes 19 and 32, so that virtual invocations invokeinterface and invokevirtual can be actually considered as *non-branching*.

As regards *exceptions*, COSTA handles internal exceptions (i.e., those associated to bytecodes as stated in the JVM specification), exceptions which are thrown (bytecode athrow) and possibly propagated back in methods, as well as finally clauses (even if they are compiled using the bytecode jsr). Exceptions are handled by adding edges to the corresponding handlers. When the type of the exception is not statically known, as it happens when exceptions come from calls to methods, mutually exclusive edges are generated, which capture all possible instantiations. In order to infer resource usage, COSTA provides the options of ignoring only internal exceptions, all possible exceptions or considering them all. In Fig. 3, exceptions are not explicitly shown as edges; instead, they are indicated by marks (*) (see Ex. 4 and 5) in bytecodes producing them. For instance, bytecode 8 might generate a (N) exception if a is null in the call (of a.length).

3.3 Compositional Analysis by means of Loop Extraction

A subsequent *loop extraction* transformation is applied to the initial CFG in order to separate sub-graphs corresponding to loops. Loop extraction has been well studied in the area of program decompilation [6] and it has been proposed in termination analysis [1]; yet, to the best of our knowledge, its use in cost analysis is new. It is crucial when the program contains *nested loops*, since it allows analyzing the program *compositionally*, in the sense that it is possible to reason on the termination and cost by taking one loop at a time. This is important for finding *ranking functions*, which are required to bound the number of iterations of loops (an essential piece of information for both cost and termination). COSTA implements an existing efficient algorithm [26] to identify the loops, and modifies

it to have loops which, in addition to having a single entry, also have a single exit. The latter condition is required to avoid multiple return branches from loops, and is allowed when additional exits correspond to exceptions which can be caught and thrown by the caller. Whenever a loop is extracted, the corresponding sub-graph is replaced by a new instruction $\mathsf{call\_loop}(j, \overline{o})$, where j is a fresh integer identifier, and $\overline{o}$ (often omitted for brevity) is the set of local variables of m which are modified by the execution of the loop. Besides, a new CFG is generated for each sub-graph, whose entry block has m_j as its identifier. Hence, after extracting the loops, there is one CFG which corresponds to the entry of m, and the remaining CFGs correspond to loops.

Example 4. Figure 3 shows the CFGs of inter after applying loop extraction. The middle graph corresponds to the loop called in inter_0 by $\mathsf{call\_loop}(1)$, while the inner loop (right graph) is called from inter_2. The (E) mark indicates that an exception can be generated in the loop and propagated back to the caller block. $\mathsf{loop\_exit}(j)$ denotes the normal exit from loops, which transfers the control to the bytecode following call_loop (bytecode 50, in the case of the outer loop). Exceptional exits from loops are omitted for brevity. □

3.4 Rule-based Representation

As already mentioned, for a method m and its CFGs, the system obtains a *rule-based representation* (RBR) for m whose purpose is twofold: 1) to transform iteration into recursion; and 2) to *flatten* the operand stack, in the sense that its content is represented as a series of local variables. The latter is possible because, in valid bytecode, the stack height can be statically decided. This is done in one pass on the CFGs, where the stack height is computed at the entry and exit of each block, and saved. The formal translation from a CFG to the rule-based representation can be found in previous work [3, 1]. In the present paper, the CFG is different, as class analysis and loop extraction have been introduced in COSTA. This results in a more accurate and compositional representation.

Intuitively, the system computes the rule-based representation of a JBC program by producing, for each basic block m_j of its associated CFGs, a rule which:

(1) contains the set of bytecode instructions within the basic block with the variables (local and stack) it operates on, appearing explicitly in the instructions;
(2) if there is a method invocation within the instructions, includes a call to the corresponding rule; and
(3) at the end, contains a call to a *continuation rule* m_j^c. The definition of a continuation must include mutually exclusive rules to cover all possible continuations from the block, guarded by the respective conditions.

Example 5. When analyzing libraries, and by taking into account exceptions, the RBR for inter contains 59 rules. Let us show the rules associated to block inter_4 in the CFG. For clarity, exception rules are not shown but we just annotate with "%" the instructions susceptible of throwing exceptions. For them, there are rules in the RBR which capture the corresponding behavior.

$$
\begin{array}{lll}
\mathsf{inter}_4(\langle l, a, al, i\rangle, \langle i\rangle) & := & \mathsf{aload}(a, s_1),\quad \mathsf{iload}(i, s_2),\\
& & \mathsf{aaload}(s_1, s_2, s_1),\quad \%\ \mathsf{NullPointerException},\ \mathsf{IndexOutOfBoundsException}\\
& & \mathsf{aload}(l, s_2),\\
& & \mathsf{getfield}(\mathsf{CompList.data}, s_2, s_2),\quad \%\ \mathsf{NullPointerException}\\
& & \mathsf{nop}(\mathsf{invokeinterface}(\mathsf{compareTo}(\langle s_1, s_2\rangle, \langle s_1\rangle))),\\
& & \mathsf{Integer\_compareTo}(\langle s_1, s_2\rangle, \langle s_1\rangle),\\
& & \quad \%\ \mathsf{NullPointerException}\ \text{and exceptions coming from invocation}\\
& & \mathsf{nop}(\mathsf{ifne}(s_1)),\quad \mathsf{inter}_4^c(\langle l, a, al, i, s_1\rangle, \langle i\rangle).\\
\mathsf{inter}_4^c(\langle l, a, al, i, s_1\rangle, \langle i\rangle) & := & \mathsf{guard}(\mathsf{ifeq}(s_1)),\quad \mathsf{inter}_5(\langle l, a, al, i\rangle, \langle i\rangle).\\
\mathsf{inter}_4^c(\langle l, a, al, i, s_1\rangle, \langle i\rangle) & := & \mathsf{guard}(\mathsf{ifne}(s_1)),\quad \mathsf{inter}_6(\langle l, a, al, i\rangle, \langle i\rangle).
\end{array}
$$

It can be seen that there are two possible continuations (rule inter_4^c), depending on the result of comparing the method output with zero. The comparison is the bytecode ifne, which is wrapped in a nop mark, meaning that the bytecode must be ignored at this point, but its cost must be taken into account later. The continuation rule may call inter_5 or inter_6, depending on which condition holds at the entry, as made explicit by guards before the calls.

□

3.5 Optimizations on the Rule-based Representation

Several automatic transformations can be done on the RBR, to improve both accuracy and efficiency of the rest of the analysis. Basically, optimizations aim at removing variables to have a simpler program representation.

Static Single Assignment. A *Static Single Assignment* [13] (SSA) transformation is performed on the bytecodes of the RBR. SSA enables simple, yet efficient, denotational program analyses. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ is transformed into $\mathsf{iadd}(s_0, s_1, s_0')$ where s_0' refers to the value of s_0 *after* the instruction. Our implementation of SSA keeps, for each rule, a mapping from variable names (as they appear in the rule) to new variable names (constraint variables). E.g., the rule for block inter_3 takes the following form after SSA:

$$
\begin{array}{lll}
\mathsf{inter}_3(\langle l, a, al, i\rangle, \langle i'\rangle) & := & \mathsf{iload}(i, s_1),\quad \mathsf{aload}(a, s_2),\quad \mathsf{arraylength}(s_2, s_2'),\\
& & \mathsf{nop}(\mathsf{ifcmpge}(s_1, s_2')),\quad \mathsf{inter}_3^c(\langle l, a, al, i, s_1, s_2'\rangle, \langle i'\rangle).
\end{array}
$$

Stack Variable Elimination. While SSA introduces new variables, it also enables the removal of a large number of stack variables which correspond to intermediate states. COSTA *unifies* stack elements, local variables and constants occurring in instructions which move data to and from the stack, as iload, iconst, istore and ireturn. These unifications reduce the number of (distinct) variables which occur in the rule. After stack variable elimination, rule inter_3 becomes:

$$
\begin{array}{lll}
\mathsf{inter}_3(\langle l, a, al, i\rangle, \langle i'\rangle) & := & \mathsf{iload}(i, i),\quad \mathsf{aload}(a, a),\quad \mathsf{arraylength}(a, a),\\
& & \mathsf{nop}(\mathsf{icmpge}(i, a)),\quad \mathsf{inter}_3^c(\langle l, a, al, i, i, a\rangle, \langle i'\rangle).
\end{array}
$$

Note that the unification in arraylength does not mean that the length of a is written in a. Actually, this kind of unification is only meant to make size analysis easier. Most stack variables can be removed. In most cases, only those stack variables associated to operations on the heap, such as $\mathsf{aaload}(s_1, s_2, s_1')$, and the return value of methods, as s_1 (s_1' after SSA) in inter_5, are kept in the RBR. Also, arguments which are duplicated in all possible call patterns to a rule can be filtered out. Rules inter_3 and inter_3^c are transformed into:

$$\begin{array}{l}
\mathsf{inter}_3(\langle l, a, al, i\rangle, \langle i'\rangle) := \mathsf{iload}(i,i),\quad \mathsf{aload}(a,a),\quad \mathsf{arraylength}(a,a),\\
\qquad\qquad\qquad\qquad\quad \mathsf{nop}(\mathsf{ifcmpge}(i,a)),\quad \mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle).\\
\mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle) := \mathsf{guard}(\mathsf{ifcmplt}(i,a)),\quad \mathsf{inter}_4(\langle l, a, al, i\rangle, \langle i'\rangle).\\
\mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle) := \mathsf{guard}(\mathsf{ifcmpge}(i,a)),\quad \mathsf{loop_exit}(3)(\langle l, a, al, i\rangle, \langle i'\rangle).
\end{array}$$

Note that iload, aload and arraylength in the SSA form of rule inter_3 have no effect here, and can be ignored by size analysis. However, they are not removed since their cost has to be taken into account when generating the recurrence relation.

Inter-Block Constant Propagation. The above optimizations only achieve *intra-block constant propagation*, as variables are unified with values within the scope of a rule (i.e., a block), but are not propagated to other rules. Clearly, *inter-block constant propagation* is interesting in terms of both accuracy (more knowledge about values) and efficiency (less variables to consider). COSTA does a simple, yet effective constant propagation post-process, where constants are propagated forward to continuation rules. In a nutshell, when a block call is found, the current calling pattern is stored and, if it is guaranteed that such block is only invoked from that point, constants in the calling pattern are propagated to its body. For instance, the call pattern to inter_3 from call_loop(3) takes 0 for the counter i. However, this block is also invoked from inter_6, so that the value cannot be propagated. For correctness, constant propagation must be stopped as soon as variables whose value is being propagated are assigned a new value. This is automatically dealt with by using unification in the SSA transformation above.

4 Context-Sensitive (Pre-)Analyses to Improve Accuracy

COSTA implements two context-sensitive analyses based on abstract interpretation [11]: *nullity* and *sign*. The aim of these analyses is to improve the accuracy (and efficiency) of subsequent steps. Both analyses infer information from individual bytecodes, and propagate it via a standard, top-down *fixpoint* computation. They are designed to achieve good performance by implementing abstract operations using *bitmaps*, which allow accessing and updating the analysis information in constant time.

4.1 Nullity Analysis

A simple nullity analysis is performed on the RBR in order to keep track of *non-null* objects. For instance, the bytecode $\mathsf{new}(s_i)$ allows to assign the abstract value *non-null* to s_i. Afterwards, this information can be propagated by means of bytecodes like $\mathsf{astore}(s_i, l_j)$, which copies the non-null abstract value of s_i into l_j. The results of nullity analysis often allow to remove rules corresponding to NullPointerException, essentially those guarded by $\mathsf{guard}(\mathsf{ifnull}(s_i))$. Nullity analysis is very effective when methods are analyzed *context-sensitively*. For instance, in the main program in Fig. 1, which calls inter with non-null lists l and al, and a non-null array a, nullity analysis of inter guarantees that no NullPointerException can be thrown when accessing fields or invoking methods belonging to the arguments of inter. Thus, bytecode instructions annotated with (N) in Fig. 3 will not generate exception branches. This is clearly beneficial both in terms of precision and efficiency of the remaining analysis steps.

4.2 Sign Analysis

Sign analysis keeps track of the sign of variables. The abstract domain contains the elements $\geq$, $\leq$, $>$, $<$, $= 0$, $\neq 0$, $\top$ and $\bot$, partially ordered in a lattice. Domain operations can be efficiently implemented with bitmaps (three bits for each abstract value). For instance, sign analysis of $\mathsf{const}(s_i, V)$ evaluates the integer value V and assigns the corresponding abstract value $= 0$, $>$ or $<$ to s_i, depending, resp., on if V is zero, positive or negative [11]. Information from arithmetic bytecode instructions is inferred as expected.

Knowing the sign of data allows to remove RBR rules associated to arithmetic exceptions which are guaranteed never to be thrown. In addition, sign information plays a crucial role in cost analysis, as it allows obtaining accurate upper bounds for *logarithmic* methods. E.g., consider a method with a simple recursive call of the form `void m(int n) { .. m(n/2);..}` for which we want to measure number of instructions executed. According to the JVM specification, without knowing the sign of n, it is not possible to know whether $n/2$ will be rounded to the next (if negative) or previous (if positive) integer. Therefore, unless accurate sign information is available, it is not possible to obtain a logarithmic upper bound for `m`; instead, a less accurate (linear) upper bound is found.

After this step, a post-process on the RBR *unfolds* intermediate rules which correspond to unique continuations. This iterative process finishes when a continuation is not unique, or when direct recursion is reached.

5 Size Analysis of Java Bytecode

From the RBR, size analysis takes care of inferring the relations between the values of variables at different points in the execution. To this end, the notion of *size measure* is crucial. The size of a piece of data at a given program point is an abstraction of the information it contains, which may be fundamental to prove termination and infer cost. The COSTA system uses several size measures:

- *Integer-value* maps an integer value to its value (i.e., the size of an integer is the value itself). It is typically used in loops with an integer counter to approximate the number of iterations by detecting how the size of the counter changes at each pass through the loop body.
- *Path-length* [18] maps an object to the length of the maximum path reachable from it by dereferencing. E.g., `null` has size 0 and, in a non-null reference x, the size of x is 1 plus the maximum path-length of fields in x which are in turn references. Therefore, for a non-cyclic data structure x, the size of x is greater than the size of any reference field of x, i.e., the size of a data structure decreases as fields are dereferenced. This measure can be used to predict the behavior of loops which go through objects, since the path-length is supposed to strictly decrease through the loop.
- *Array-length* maps an array to its length and is used to predict the behavior of loops which traverse arrays.

Sec. 5.1 shows how it is possible to improve the efficiency of size analysis by simplifying the abstract compilation removing useless information. Finally, a description of the actual size analysis is given in Sec. 5.2 and 5.3.

5.1 Slicing of Useless Variables

When looking at the RBR, it is sometimes possible to note that some variables are not relevant for the specific purpose of getting cost information, and can therefore be removed in order to make the analysis more efficient and the solving process more feasible [4]. In this sense, a variable is *relevant* if it directly or indirectly affects some guards, i.e., the control flow (thus, potentially, the cost), or is needed by the cost model (e.g., in our example, $\mathcal{M}_{heap}$ needs the length of an array created by newarray to infer the allocated memory). Non-relevant variables can be removed from the RBR. As an example, an accumulator variable, which only stores partial results of a computation (e.g., the sum of the elements of a list, where a temporary variable is updated during the loop) is essential to the semantics, but can be removed since, in general, does not affect the cost.

To this end, a variant of *backward program slicing* [27] is used, where variables are removed instead of program statements. The *slicing criterion* consists of the variables occurring in guards or needed by the cost model, which are propagated backwards through the rules by means of a simple dependency calculus, so that variables which directly or indirectly affect the criterion are kept in the slice. As a result, variables which cannot affect the cost are removed.

Unlike in normal slicing, *soundness* is not an issue here: removing variables which are actually relevant may result in a loss of precision, but the correctness of (upper bound) cost and termination results is preserved. In fact, losing precision would make the upper bound bigger (possibly infinite, meaning that it was impossible at all to infer the cost of the program), or make it impossible to prove termination, but such result would not lose correctness (since a bigger upper bound is correct whenever a smaller one is correct, and *not proving* termination is trivially correct). Because of this, the treatment of calls to methods or loop rules can be simplified: when a call to m is found, relevant variables of m are taken (i.e., those which affect its cost), but relevant variables in the caller rule are not propagated through the call (*context-insensitivity*). Such a slicing on the rules is unsound, and different with respect to a previous, analogous algorithm [4], where this information is correctly dealt with (*context-sensitivity*). This results in a less precise and unsound, but more efficient and importantly, scalable slicing.

5.2 Abstract Compilation

The purpose of size analysis is to detect how the size of variables changes during execution [14]. For example, when analyzing a loop where an integer counter i goes from 0 to a threshold, as in the inner loop of Ex. 1, size analysis w.r.t. Integer-value should see that the size of i in the n-th iteration of the loop is greater by 1 than its size in the $n-1$-th iteration. This information is essential for inferring how many times the loop body will be executed, which is a crucial piece of information in cost and termination analyses. Each bytecode, call or guard is *abstracted* by *linear constraints* on the size of its variables: for example, $\mathsf{iadd}(s_0, s_1, s'_0)$ will be abstracted by the constraint $s'_0{=}s_1{+}s_0$, meaning that the size of s_0 after executing the instruction is the sum of the size of s_0 and

s_1 before. Similarly, $\mathsf{getfield}(f, s_0, s_0')$ is abstracted by $s_0 > s_0'$, meaning that the (Path-length) output size is less than the input size, due to the field access. This only holds if *non-cyclicity* of s_0 can be proven; otherwise, no information can be obtained, and an empty constraint is produced. We refer to [18] for details on path-length and its requirements. This step results in an *abstract constraint program*, or simply *abstract compilation*, which approximates the cost and termination behavior of the original program w.r.t. the chosen size abstractions. E.g., rules inter_3 and inter_3^c, after RBR optimizations, are abstract-compiled into:

$$\mathsf{inter}_3(\langle l, a, al, i\rangle, \langle i'\rangle) := \{\} \quad \diamond\, \mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle)$$
$$\mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle) := \{i{<}a\} \diamond\, \mathsf{inter}_4(\langle l, a, al, i\rangle, \langle i'\rangle)$$
$$\mathsf{inter}_3^c(\langle l, a, al, i\rangle, \langle i'\rangle) := \{i{\geq}a\} \diamond\, \mathsf{loop\_exit}(3)(\langle l, a, al, i\rangle, \langle i'\rangle)$$

Expressions in brackets are constraints which describe the behavior of the bytecodes. Abstract rules for the loops in the example are:

$$\mathsf{inter}_2(\langle l, a, al, i\rangle, \langle l'', i'''\rangle) := \{i'{=}0, l{>}l'\} \diamond \mathsf{inter}_3(\langle l, a, al, i'\rangle, \langle i''\rangle), \mathsf{inter}_1(\langle l', a, al, i''\rangle, \langle l'', i'''\rangle)$$
$$\mathsf{inter}_6(\langle l, a, al, i\rangle, \langle i'\rangle) := \{i'{=}i{+}1\} \diamond \mathsf{inter}_3(\langle l, a, al, i'\rangle, \langle i'\rangle)$$

The first rule corresponds to the outermost loop, which calls the inner loop with $i = 0$. Note that, provided l is non-cyclic and does not share memory locations in the heap with other variables, size analysis finds a size decreasing in the outer loop. Moreover, by applying the Integer-value measure, it is inferred that i (the counter of the internal loop) increases by one between the input of rule inter_6 and that of inter_3 (the condition of the loop). In both cases, a useful size relation has been found, thus allowing the subsequent cost analysis to understand the behavior of loops.

5.3 Bottom-Up Fixpoint Computation

Linear constraints replacing parts of the program can be propagated via a standard, bottom-up *fixpoint* computation, in order to combine the information about single rules. The goal of this global analysis is to have *size relations* on variables between the input of a rule (i.e., a block in the CFG) and that of another one which can be (directly or indirectly) called by the first one.

In practice, we can often take a trivial over-approximation where for any rules there is no information, i.e., $p(\overline{x}, \overline{y}) \leftarrow \mathit{true}$. This is often enough to prove termination and find upper bounds on the cost of many programs, and results in a more efficient implementation. It is enough in our example, but not in cases where the call modifies the data structure over which a loop of the caller goes. For instance, it would be needed in the example if methods invoked within the loop (either compareTo or add) modify the length of l or the value of i. However, experiments suggest that this is not very likely to occur in imperative programs.

6 Inferring Cost and Termination

Once the bytecode program has been transformed into its RBR (Sec. 3 and 4), and size relations have been inferred (Sec. 5), all the pieces are available to prove

termination and infer a *closed-form upper bound* for the cost of the bytecode. To this purpose, COSTA first sets up a recurrence relation system (RR) which captures the cost of the rule-based program and its termination behavior in terms of the input values, and, afterwards, uses a generic RR solver [2] to obtain an upper bound and prove termination.

6.1 Setting up Recurrence Relations

Setting up a RR from the bytecode culminates the phase 1 of cost analysis (Sec. 1). In particular, for each rule in the RBR, COSTA generates a cost equation of the form $r_p(\overline{x}_p) = exp + [c_j(\overline{x}_j)+]r_q(\overline{x}_q), \varphi$ 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). Here, the optional c_j is the cost of a method invoked from within a rule. Variables $\overline{x}$ are the set of corresponding variables relevant to the cost. Essentially, the equation states that, for given (abstract) values $\overline{v}_p$ such that $\varphi \models \wedge \overline{x}_p = \overline{v}_p$, a *possible* cost for $r_p(\overline{v_p})$ is $exp[x_p \mapsto v_p]$ plus the sum of the costs of $c_j(\overline{v}_j)$ and $r_q(\overline{v}_q)$, where values v_j and v_q are obtained from v_p and the constraints. For example, in $\mathcal{M}_{inst}$, the RR for inter comes to be (as in Ex. 1, c_2 is the cost of add, while c_6 is the cost of compareTo):

$$
\begin{array}{rll}
\mathsf{inter}(l,a) = 1 + r_1(l,a), & \{\} \\
r_1(l,a) = 2 + r_2(l,a,l), & \{\} \\
r_2(l,a,l) = 6 + r_3(a,0) + r_1(l',a), & \{l > l', l' \geq 0, a \geq 0\} \\
r_2(0,a,0) = 0, & \{\} \\
r_3(a,i) = 4 + r_4(a,i,i,a), & \{\} \\
r_4(a,i,i,a) = 6 + c_6 + r_5(a,i,s_1), & \{a > i\} \\
r_4(a,i,i,a) = 0, & \{a \leq i\} \\
r_5(a,i,s_1) = r_6(a,i), & \{s_1 \neq 0\} \\
r_5(a,i,s_1) = 4 + c_2 + r_6(a,i), & \{s_1 = 0\} \\
r_6(a,i) = 2 + r_3(a,i'), & \{i' = i + 1\}
\end{array}
$$

Consider the outer loop: the *execution* of r_1 (corresponding to block inter_1) costs 2 bytecodes plus the cost of r_2. In r_2, 6 bytecodes are executed (those in block inter_2) in the loop body, so that the cost is 6 plus that of the call r_3 to the inner loop, and of r_1. This goes on until a call to $r_2(0,a,0)$ ends the loop. Note that r_1 is called by r_2 with the first argument decreased, which guarantees termination. The above RR has been simplified by eliminating intermediate equations by means of *unfolding*, as COSTA actually does.

6.2 Finding Closed-form Upper Bounds and Proving Termination

RRs have a great potential: they are not limited to any complexity class, and can be used for counting different resources. However, unless a *closed-form solution* describing the cost of a program only in terms of its input variables is found (i.e., with no references to other equations), RRs turn out not to be practical (see the applications pointed out in Sec. 9). This is the so-called phase 2 in Sec. 1.

Basically, a RR is a *non-deterministic constraint functional program* which allows to use generic tools both to find closed-form solutions and to prove termination. Non-determinism might occur due to the loss of precision inherent

to (static) size analysis. This means that, for given input values $\overline{v}_p$, the query $C_p(\overline{v}_p)$ may result in several solutions. It can be seen in the above example that size relations are inexact: e.g., size analysis has inferred that the size of a data structure l_0 decreases, but does not tell how much. In such cases, size relations cannot be applied; instead, they are kept in the cost equations. Yet, it is guaranteed that (1) one of the solutions corresponds to the actual cost of the rule-based program; and (2) if $C_p(\overline{v}_p)$ has a finite number of solutions and does not lead to any infinite computations, then the original bytecode program terminates for any corresponding concrete input. Due to the *non-decidable* and *non-deterministic* features of RRs, in most cases, it is not possible to obtain an exact solution (see [2]). Rather, the aim is to obtain *non-asymptotic*[3] upper bounds.

Upper bounds. RR are independent of the language in which the original program was written. This traditionally has allowed relying on existing computer algebra systems (e.g., Maple, Mathematica, Maxima) to carry out phase 2 of cost analysis. In our case, COSTA is connected to an existing upper bound solver [2], which is especially designed to handle RR output by automatic cost analysis. The differences between a RR and a standard recurrence equation system are explained in detail in that work. The solver is available on the web (`http://www.cliplab.org/Systems/PUBS`). It is independent of the language the RR is obtained from, and handles a large set of complexity classes, such as *logarithmic*, *linear*, *polynomial*, and *exponential*. In the example, the obtained upper bound is $\mathcal{M}_{inst}(\mathsf{inter})$, shown in Fig. 1. Details of the solving process are rather technical, and are outside the scope of this paper [2].

Termination from RR. As already mentioned, proving termination involves guaranteeing that a *finite upper bound* for the system exists, even if it cannot be found explicitly. As a RR is a non-deterministic constraint functional program, well-studied techniques used for proving termination in such languages can be directly adapted to our setting. The solver actually proves termination on the above representation by using semantic-based techniques, relying on *binary unfolding* combined with *ranking functions*, as those in [10]. In the example, it is able to prove termination of inter alone, and also of main. Termination on the non-deterministic constraint functional representation implies, in turn, termination of the Java bytecode program, as proven in previous work [1].

7 Experimental Results

The COSTA system is implemented in Prolog and, as an external component, it uses the Parma Polyhedra Library [8] for manipulating linear constraints and it is connected to the solver of [2, 1] to find upper bounds and prove termination. In contrast to previous experimental work on cost analysis, a main goal of our experiments is to be able to analyze *realistic* programs which are not

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

Bench	#M	CFG	#R	Null	Sign	$\#R_r$	AC	SA	#T	#UB
compInter	5	84	146	28	36	113	76	1124	5	5
	19	4600	1997	908	1104	293	216	1828	19	19
stackRev	17	436	496	104	96	332	248	1536	17	17
	27	1848	602	112	112	390	300	1856	27	27
josephus	23	924	986	260	280	780	520	11713	23	23*
	89	6752	2993	1112	1364	2187	1732	16049	77	8
arrayMax	3	120	163	24	28	137	96	536	3	3
	34	3096	1096	344	476	786	588	4504	29	1
ArrayList	34	8917	1649	308	332	1381	1096	3124	33	30
	529	51567	15828	6400	6704	11413	9137	63120	26	23
Character	43	14337	758	116	160	684	364	1684	43	43*
	166	53971	2829	560	464	2464	1544	2296	166	166
Integer	52	24054	2059	784	928	1704	3008	5468	46	43*
	217	49043	8758	5320	6488	4963	11029	29198	103	18

Table 1. COSTA Analysis Times and Results for Benchmarks using Libraries

hand-crafted but rather are taken from different benchmark suites, namely from the own Java libraries and the book [15] and do *not* use predefined assertions but rather analyze all necessary code. The first benchmark, compInter, is our running example which, as we have seen through the paper, uses several classes and interfaces from the Java libraries. The next set of benchmarks stackRev, josephus and arrayMax appear in [15] and all of them use Java libraries. The next three benchmarks are Java libraries: java.util.ArrayList, java.lang.Character and java.lang.Integer.

Table 1 shows the efficiency and accuracy of COSTA on the above examples. For each benchmark, we have two rows: the upper one corresponds to the case where we analyze only user-defined code, and the lower row includes the analysis of all required library methods. The column **#M** shows the number of methods to be analyzed for each benchmark. We can observe that the benchmarks are reasonably large, up to 529 methods analyzed for arrayList (with libraries). The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM. Times are in milliseconds and measure the runtime of each of the phases undertaken by the analyzer. In particular, columns **CFG**, **Null**, **Sign**, **AC** and **SA** show, resp., the time of building the CFG, nullity analysis, sign analysis, abstract compilation and size analysis. We argue that analysis times are reasonable given the large size of the benchmarks. Only size analysis is comparatively more expensive. Interestingly, it is often not required in order to prove termination nor to infer upper bounds, in particular, when the loops conditions do not depend on the return value from a method. In the table, we mark the upper bounds with "*" in the three cases when size analysis is required. Columns **#R** and $\#\mathbf{R}_r$ show the number of rules in the RBR of the bytecode program, resp., prior to nullity and sign analysis and after applying them (as explained in Sec. 4). It can be observed that the reduction is significant in all benchmarks. This is crucial for both the efficiency and accuracy of the analysis.

The last two columns **#T** and **#UB** indicate, resp., the number of methods for which we are able to prove termination and infer an upper bound for $\mathcal{M}_{inst}$. We believe our results are quite encouraging. We have proved termination and obtained upper bounds for all methods in compInter, stackRev and Character. As expected, obtaining upper bounds for $\mathcal{M}_{inst}$ is strictly more difficult than proving termination: if we fail to find a well-founded decreasing measure for a loop which ensures its termination, we also fail to bound the number of iterations

of such loop. Most of the examples where COSTA fails, e.g., in arrayMax and josephus with libraries, contain loops whose number of iterations depends on the values of fields. This is currently not supported by our size analysis and, moreover, we are not aware of any analysis that can infer such information. In other examples, PUBS [2] fails to find an upper bound because the RR obtained is too large. This happens for some methods in arrayMax, josephus and ArrayList.

Regarding the language, there are some features of Java bytecode that COSTA does not support such as non-sequential, native code, dynamic code generation and reflection. COSTA can still deal with some of them (like native code) by giving symbolic names to their cost, as we have shown along the paper. All in all, we believe that our experiments thus far allow us to conclude that RUA can be applied to a realistic programming language, and to programs with a realistic size and complexity.

8 Related Work

Since the advent of mobile code, Java bytecode analysis has become an active research area, and a number of tools are now available, e.g., the *Soot* framework [28] and the generic analyzer *Julia* [24]. Soot is a framework to develop analyses of Java bytecode, and already includes *points-to*, *purity* and *dynamic data structure* analysis. Similarly to COSTA, such systems transform bytecode into a procedural representation. Indeed, intermediate representations are common practice in JBC analysis (see also *BoogiePL* [20]). The main differences w.r.t. our rule-based representation are: (1) though Soot also performs SSA when generating the *Shimple* representation, neither Shimple nor BoogiePL do the optimizations described in Sec. 3.5: our system can eliminate, in one pass, almost all stack variables in the RBR and, besides, slice out variables which do not affect the cost; this results in a more efficient subsequent size analysis. (2) Neither Soot nor BoogiePL perform loop extraction, which is important for compositionality in cost analysis. Julia provides a generic analysis engine where *sharing*, *class*, *nullity*, *information flow*, *escape* and *static initialization* analyses have been integrated. None of these systems include *resource usage* analysis, though Julia implements some components (in particular class, nullity, sharing and *cyclicity* analyses) which are required by *size* analysis (Sec. 5).

Focusing on *cost* analysis, important effort has been devoted to adapt the general framework by Wegbreit [29] to different languages and programming paradigms. A main goal in this line is defining a setting where RRs can be generated from different languages. In the context of Java bytecode, a cost analysis framework is presented in [3] which shows that standard cost analysis can be performed on Java bytecode. Moreover, the framework has been instantiated to *heap consumption* inference [5]. Essentially, it proposes to (1) transform the bytecode into a high-level recursive representation; and (2) perform size analysis on it to generate the RR. This work has heavily influenced the design of COSTA, which follows the same basic steps. However, though providing convincing arguments for the feasibility of cost analysis in a bytecode language, this work has not yet provided the components needed for the design and implementation of a

scalable and realistic resource analyzer. In particular, the recursive representation lacked class analysis (Sec. 3.2), loop extraction (Sec. 3.3) and optimizations in Sec. 3.5, which are fundamental to design a manageable bytecode representation to infer resource usage. The removal of useless variables is the subject of previous work [4], but that algorithm is less efficient, as already discussed in Sec. 5.1. As regards the cost process itself, it lacked the analysis steps described in Sec. 4.1 and 4.2, and did not perform *abstract compilation* to implement the size analysis (Sec. 5.3). All the new components presented in this paper are required to achieve efficient and accurate cost and termination analyses, and apply them to realistic benchmarks.

9 Discussion and Applications

The COSTA system provides a platform for integrating resource usage analysis for Java bytecode by providing the notion of resource as a *black box* component. The analyzer follows the traditional approach to cost analysis, i.e., generating and solving *recurrence relations*. This approach is very powerful, as it is not restricted to any complexity class, and can be used to measure several interesting resources. Also, a unique feature of COSTA is that it works at the bytecode level, which makes it possible to obtain more accurate upper bounds w.r.t. the source level, as compiler optimizations at the level of the JVM are already accounted for. Java bytecode analysis implies problems typically occurring in those arising in the object-oriented paradigm. Our approach handles these issues, and can be applied in the usual fields related to resource usage analysis:

Granularity Control [14, 16]. Parallel computers have currently become mainstream with multicore processors. In parallel systems, knowledge about the cost of different procedures in the object code can be used to guide the partitioning, allocation and scheduling of parallel processes.

Performance Debugging and Validation [17]. This is a direct application of cost analysis, where the analyzer checks assertions about the efficiency of the program, written by the programmer. Assertions possibly refer to source code, but can be easily translated to be understandable by the bytecode analyzer. Likewise, analysis results obtained on the bytecode are somehow closer to the actual runtime behavior, and can be easily related to the Java program.

Resource Bound Certification [12, 7, 9]. It refers to the certification of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on resource consumption. This is a key point in the design of *Proof-Carrying code* [22] architectures, where the user wants some guarantees that running the code will not take too much an amount of resources. Previous work deals with linear bounds [12, 7], semi-automatic techniques [9], or source code [17]. Our approach shows that it is possible to automatically generate *cost-bound* certificates for realistic mobile, Java bytecode languages.

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.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18. Springer-Verlag, 2008.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*, LNCS, pages 221–237, 2008.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *Proc. SAC*. ACM Press, 2008.
5. E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07*, October 2007.
6. F. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.
7. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.
8. R. Bagnara, E. Ricci, E. Zaffanella, and P.M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In *SAS,02*, LNCS 2477, pages 213–229. Springer-Verlag, 2002.
9. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, LNCS 3444, pages 311–325, 2005.
10. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
11. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252. ACM, 1977.
12. K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.
13. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4), 1991.
14. S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.
15. M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. John Wiley, 3rd edition, 2004.
16. M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proc. of PPDP'05*. ACM Press, July 2005.

17. M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Comp. Progr.*, 58(1–2), 2005.
18. P.M. Hill, E. Payet, and F. Spoto. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTS. Elsevier, 2006.
19. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, pages 81–92. ACM, 2001.
20. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.
21. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
22. G. Necula. Proof-Carrying Code. In *POPL'97*, pages 106–119. ACM Press, 1997.
23. D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4), 1995.
24. F. Spoto. JULIA: A generic static analyser for the java bytecode. In *FTfJP'05*, 2005.
25. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
26. W. Zou T. Wei, J. Mao and Y. Chen. A new algorithm for identifying loops in decompilation. In *SAS'07*, LNCS 4634, pages 170–183, 2007.
27. F. Tip. A Survey of Program Slicing Techniques. *J. of Prog. Lang.*, 3, 1995.
28. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.
29. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.