Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Cost analysis of object-oriented bytecode programs[☆]

Elvira Albert[a], Puri Arenas[a], Samir Genaim[a], German Puebla[b], Damiano Zanardini[b,*]

[a] *Complutense University of Madrid, Spain*
[b] *Technical University of Madrid, Spain*

ARTICLE INFO

Keywords:
Cost analysis
Resource usage analysis
Bytecode languages
Static analysis
Quantitative analysis

ABSTRACT

Cost analysis statically approximates the cost of programs in terms of their input data size. This paper presents, to the best of our knowledge, the first approach to the automatic cost analysis of object-oriented bytecode programs. In languages such as Java and C#, analyzing bytecode has a much wider application area than analyzing source code since the latter is often not available. Cost analysis in this context has to consider, among others, dynamic dispatch, jumps, the operand stack, and the heap. Our method takes a bytecode program and a *cost model* specifying the resource of interest, and generates *cost relations* which approximate the execution cost of the program with respect to such resource. We report on COSTA, an implementation for Java bytecode which can obtain upper bounds on cost for a large class of programs and complexity classes. Our basic techniques can be directly applied to infer cost relations for other object-oriented imperative languages, not necessarily in bytecode form.

1. Introduction

Computational complexity theory is a fundamental research topic in computer science, which aims at determining the amount of resources required to run a given algorithm or to solve a given problem in terms of the input value. This topic has received considerable attention since the early days of computer science. The most common metrics studied are *time-complexity* and *space-complexity*, which measure, respectively, the time and memory required for running an algorithm or solving a problem. Due to its focus on measuring quantitative aspects of program executions, it is natural to consider computational complexity as a first-class citizen in the area of *quantitative analysis*. In complexity theory, algorithms and problems are often categorized into *complexity classes*, according to the amount of resources required for executing the algorithm or solving the problem by using the best possible algorithm. Although, especially in recent decades, complexity theory has produced a wealth of research results, assigning a complexity class to an algorithm is still far from being automatic, and requires significant human intervention.

In this work, rather than on the complexity of problems or algorithms, we concentrate on analyzing the complexity of *programs*. The first proposal for doing this *automatically* was the seminal work by Wegbreit [63], wherein the *Metric* system is described, together with a number of applications of automatic cost analysis. This system was able to automatically compute *closed-form cost functions* which capture the non-asymptotic cost of simple Lisp programs as functions of the size of the input arguments. Since then, a number of cost analysis frameworks have been proposed, mostly in the context of *declarative* languages (functional [45,54,62,56,19] and logic programming [33,48]). Imperative languages have received significantly less attention. It is worth mentioning the pioneering work by Adachi et al. [1]. There also exist cost analysis frameworks which do not follow Wegbreit's approach [43,25].

[☆] This work is an extended and revised version of ESOP'07 (Albert et al., 2007) [9].
[*] Correspondence to: Technical University of Madrid, 28660 Boadilla del Monte, Madrid, Spain. Tel.: +34 600317288.
E-mail address: damiano.zanardini@gmail.com (D. Zanardini).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.07.009

Traditionally, cost analysis has been formulated at the *source code* level. However, it can be the case that the analysis must consider the *compiled code* instead. This may happen, in particular, when the *code consumer* is interested in verifying some properties of third-party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. This is the general picture where the idea of *Proof-Carrying Code* [49] was born: in order for the code to be verifiable by the user, safety properties (including resource usage) must refer to the (compiled) code available, so that it is possible to check the provided proof and verify that the program satisfies the requirements.

1.1. Summary of contributions

As our main contribution, the present work formulates an automatic approach to cost analysis of real-life, object-oriented bytecode programs (from now on, we use *bytecode* for short), whose features imply dealing with the most important difficulties encountered when analyzing (source) object-oriented and low-level code: (1) as a low-level language, bytecode features *unstructured control flow*, i.e., execution flow is modified using conditional and unconditional *jumps*; (2) as an object-oriented language, bytecode includes features such as *virtual method invocation*, extensive usage of exceptions, and the use of a *heap*; moreover, (3) an additional challenge in bytecode is the use of an *operand stack* for storing the intermediate results of computations.

Our analysis takes as input the bytecode corresponding to a program and the cost model of interest, and yields a set of recursive equations which capture the cost of the program. The following steps are performed:

1 *Intermediate representation.* As it is customary in the analysis of bytecode [61,9,34], we develop our method on an intermediate *rule-based* representation (RBR for short) which is generated from the original bytecode program automatically by using techniques from *compiler theory* [2,3].

2 *Size relations.* Static analysis infers linear *size relations* (non-linear arithmetic is not supported) among program variables at different program points. Size relations are, in the case of integer variables, constraints on the values of variables, and, in the case of references, constraints on their *path length*, i.e., the length of the longest reference chain reachable from the given reference [59].

3 *Cost model.* A parametric notion of *cost model* is introduced, which allows one to describe how the resource consumption associated to a program execution should be computed. A cost model defines how cost is assigned to each execution step and, by extension, to an entire execution trace. We consider a range of non-trivial cost models for measuring different *quantitative* aspects of computations (number of steps, memory, etc.).

4 *Cost relations.* From the RBR, the size relations, and a given cost model, a *cost relation system* (CRS for short) is automatically obtained. CRSs express the cost of any block in the control flow graph (or rule in the RBR) in terms of the cost of the block itself plus the cost of its successors.

5 *Upper bound.* If possible, an exact solution or an upper bound in non-recursive form (i.e., a *closed-form* solution or upper bound) is found for the cost relation system. This step requires the use of a solver for such systems, whose details are not in the scope of this paper [7].

As another contribution, we report on the COSTA system: an implementation of our proposed framework for *Java bytecode* (JBC), which is one of the most widely used languages in *mobile code* architectures, and a candidate for building a realistic *proof-carrying code* framework for software verification.

1.2. Applications of cost analysis of object-oriented bytecode programs

Resource bound certification. This research area deals with security properties involving resource-usage requirements; i.e., the (untrusted) code must adhere to specific bounds on its resource consumption. The present work automatically generates non-trivial resource-usage bounds for a realistic programming language. Such bounds could then be translated to *certificates*.

Performance debugging and validation. This is a direct application of resource usage analysis, where the analyzer tries to verify or falsify *assertions* about the efficiency of the program. This application was already mentioned as future work by [63], and is available in a number of systems [42,5].

Granularity control. Parallel computers have recently become mainstream with the massive use of *multicore* processors. In parallel systems, knowledge about the cost of different procedures in the code can be used in order to guide the partitioning, allocation and scheduling of parallel processes [33,41].

Program synthesis and optimization. This application was already mentioned as one of the motivations by [63]. Both in program synthesis and in semantic-preserving optimizations, such as *partial evaluation* [31,52], there are multiple programs which may be produced in the process, with possibly different efficiency levels. Here, automatic cost analysis can be used for guiding the selection process among a set of candidates.

2. The rule-based representation

Bytecode is more complicated to (manually or automatically) reason about than high-level languages like Java, since it features *unstructured* control flow, where *jumps* are allowed instead of *if-then-else*, *switch* and loop structures. It uses the *operand stack* to hold intermediate results of the computation. Moreover, *virtual method invocation* makes the analysis even more difficult.

```
class A {
  int inc(int i) {
    return i+1;
  }
}
class B extends A {
  int inc(int i) {
    return i+2;
  }
}
class C extends B {
  int inc(int i) {
    return i+3;
  }
}
class M {
  int add(int n,A o) {
    int res=0;
    int i=0;
    while (i<=n) {
      res=res+i;
      i=o.inc(i);
    }
    return res;
  }
}
```

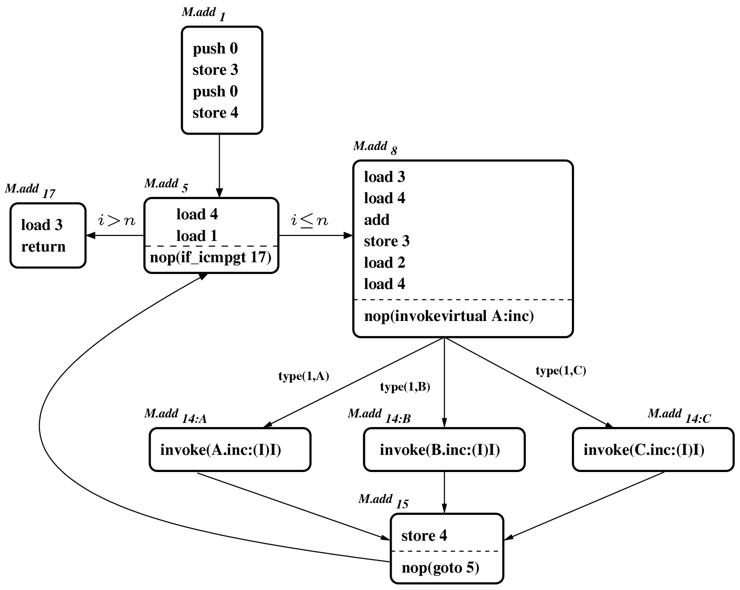

Fig. 1. A Java source (left) and bytecode for method add within its CFG (right).

Example 2.1. Fig. 1 shows to the left the Java source of our running example (shown only for clarity as the analysis works directly on the bytecode). Bytecode instructions for the method `M.add` are shown on the right within its control flow graph (CFG). Classes `A`, `B` and `C` provide different implementations for the `inc` method, which returns the result of increasing an integer by a different amount. The method `M.add` computes (1) $\sum_{i=0}^{n} i$ if the runtime class of `o` is `A`; (2) $\sum_{i=0}^{\lfloor n/2 \rfloor} 2i$ if the runtime class is `B`; or (3) $\sum_{i=0}^{\lfloor n/3 \rfloor} 3i$ if the runtime class is `C`. The block $M.add_1$ is the entry block. It initializes the local variables `res` and `i` to 0, corresponding, respectively, to indices 3 and 4 in the table of bytecode local variables. The block $M.add_5$ corresponds to the loop condition. It compares `n` and `i`; depending on the result, the execution continues to $M.add_{17}$ (i.e., exits the loop), or to $M.add_8$ (i.e., enters the loop). The instruction "`if_icmpgt 17`" is wrapped by `nop`, and its effect is "moved" to the corresponding edges. In block $M.add_8$, the first four instructions increase `res` by `i`, then the values of `o` and `i` (local variables 2 and 4) are pushed into the stack in order to perform the call to `inc`. Depending on the runtime type of `o`, we move to $M.add_{14:A}$, $M.add_{14:B}$, or $M.add_{14:C}$, and invoke the method `inc` of class `A`, `B` or `C`, respectively. On the first out-edge of block $M.add_8$, `type(1, A)` succeeds if the type of the object in stack position 1 is `A`. In the block $M.add_{15}$, the return value is stored in `i`, and the execution moves back to $M.add_5$.

Due to the challenges mentioned above, it is customary to develop analyses for bytecode on an intermediate language [61,9]. In this section, we present the rule-based *structured* language in which we will develop our analysis. The language is rich enough to allow the (de-)compilation of bytecode programs to it (and preserve the information about cost), yet is simple enough to develop a precise cost analysis. The following key features of the *rule-based representation* will make the development of the analysis easier:

1. recursion becomes the only form of iteration;
2. there is only one form of conditional construct: the use of *guarded rules*;
3. there is only one kind of variables: *local variables*; also, there is no stack;
4. some object-oriented features are no longer present: (i) classes can be simply regarded as records; and (ii) the behavior induced by dynamic dispatch is compiled into *dispatch blocks* using class analysis;
5. there is no distinction between executing a method and executing a block.

2.1. The abstract syntax

A *rule-based representation* (RBR) consists of a set of (global) *procedures*. A procedure p with k input arguments $\bar{x}$ and a single output argument y is defined by a set of *guarded rules* according to the following grammar:

$$\begin{array}{rcl} rule & ::= & p(\bar{x}, y)\ \leftarrow g, b_1, \ldots, b_n \\ g & ::= & true \mid exp_1\ op\ exp_2 \mid \mathsf{type}(x, c) \\ b & ::= & x{:=}exp \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid \mathsf{nop}(any) \mid q(\bar{x}, y) \\ exp & ::= & x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \\ op & ::= & > \mid < \mid \le \mid \ge \mid = \end{array}$$

where $p(\bar{x}, y)$ is the *head* of the rule, and $\bar{x} = x_1, \ldots, x_k$. Note that the last argument is always the output argument. n is an integer; c is a class (i.e., record) name taken from a set of class names $\mathcal{C}$; $q(\bar{x}, y)$ is a procedure call (by value); and $\mathsf{nop}(any)$ is an auxiliary instruction which takes any bytecode instruction as argument, and has no effect on the semantics (but is useful for preserving information about the original bytecode program). In the following, *Instr* denotes the set of instructions which can appear in the body of the rules. Note that, even though the RBR is more readable, all guards and instructions correspond to three-address code, as in bytecode, except for procedure calls.

The class hierarchy of the bytecode program is used, together with class analysis, in order to generate the required dispatch blocks, namely, for resolving the virtual calls statically. Furthermore, RBR programs are *deterministic* since the guards for all rules for the same procedure are pairwise mutually exclusive, and the disjunction of all guards is always true (i.e., all possible cases are covered and only one rule can be chosen). The RBR may include rules whose name has the superscript c, which correspond to *continuation* procedures, and are used to choose one execution branch when there is more than one successor.

The compilation of bytecode programs into the RBR is done by building the CFG for the bytecode program and representing each block in the CFG by means of a rule. The arguments to the bytecode instructions are made explicit, and the operand stack is *flattened* by converting its contents into local variables. This process is rather standard (similar to [61,9]) and hence it is omitted.

Example 2.2. The RBR for methods `add` and `inc` (of class `A`) is:

$$\begin{array}{ll}
add(th, n, o, r) & \leftarrow add_1(th, n, o, res, i, r). \\
add_1(th, n, o, res, i, r) & \leftarrow s_1{:=}0, res{:=}s_1, s_1{:=}0, i{:=}s_1, \\
 & \quad add_5(th, n, o, res, i, r). \\
add_5(th, n, o, res, i, r) & \leftarrow s_1{:=}i, s_2{:=}n, \mathsf{nop}(\mathsf{if\_icmpgt\ 17}), \\
 & \quad add_5^c(th, n, o, res, i, s_1, s_2, r). \\
add_5^c(th, n, o, res, i, s_1, s_2, r) & \leftarrow s_1 > s_2, add_{17}(th, n, o, res, i, r). \\
add_5^c(th, n, o, res, i, s_1, s_2, r) & \leftarrow s_1 \le s_2, add_8(th, n, o, res, i, r). \\
add_{17}(th, n, o, res, i, r) & \leftarrow s_1{:=}res, r{:=}s_1 \\
add_8(th, n, o, res, i, r) & \leftarrow s_1{:=}res, s_2{:=}i, s_1{:=}s_1 + s_2, res{:=}s_1, \\
 & \quad s_1{:=}o, s_2{:=}i, \mathsf{nop}(\mathsf{invokevirtual}\ \mathtt{A.inc(I)I}), \\
 & \quad add_8^c(th, n, o, res, i, s_1, s_2, r). \\
add_8^c(th, n, o, res, i, s_1, s_2, r) & \leftarrow \mathsf{type}(s_1, A), add_{14:A}(th, n, o, res, i, s_1, s_2, r). \\
add_8^c(th, n, o, res, i, s_1, s_2, r) & \leftarrow \mathsf{type}(s_1, B), add_{14:B}(th, n, o, res, i, s_1, s_2, r). \\
add_8^c(th, n, o, res, i, s_1, s_2, r) & \leftarrow \mathsf{type}(s_1, C), add_{14:C}(th, n, o, res, i, s_1, s_2, r). \\
add_{14:A}(th, n, o, res, i, s_1, s_2, r) & \leftarrow A.inc(s_1, s_2, s_1), add_{15}(th, n, o, res, i, s_1, r). \\
add_{14:B}(th, n, o, res, i, s_1, s_2, r) & \leftarrow B.inc(s_1, s_2, s_1), add_{15}(th, n, o, res, i, s_1, r). \\
add_{14:C}(th, n, o, res, i, s_1, s_2, r) & \leftarrow C.inc(s_1, s_2, s_1), add_{15}(th, n, o, res, i, s_1, r). \\
add_{15}(th, n, o, res, i, s_1, r) & \leftarrow i{:=}s_1, \mathsf{nop}(\mathsf{goto\ 5}), add_5(th, n, o, res, i, r). \\
\hline
A.inc(th, i, r) & \leftarrow A.inc_1(th, i, r). \\
A.inc_1(th, i, r) & \leftarrow s_1{:=}i, s_2{:=}1, s_1{:=}s_1 + s_2, r{:=}s_1. \\
\hline
\end{array}$$

It can be observed that rules in the RBR correspond to blocks in the CFG of Fig. 1. The first rule is the entry procedure of *add*, which receives as input the method arguments. Local variables have the same name as in the source code (*th* stands for `this`). Always true guards are omitted. The call to add_1 from *add* adds local variables as parameters. The loop starts at add_5; bytecodes pushing i and n on the stack are compiled to $s_1{:=}i$ and $s_2{:=}n$. Rule add_5 calls add_5^c to check the loop condition. If $s_1 > s_2$ (i.e., `i > n` in the source), then the loop ends, and add_{17} is called, which assigns s_1 to the return value r and terminates. Otherwise, the loop continues on add_8, which accumulates i on *res* and prepares the call to *A.inc* by assigning its parameters to the stack variables. Finally, it calls the continuation add_8^c, which depends on the runtime type of o and calls the corresponding dispatch block $add_{14:\_}$ (i.e., the instance matching the guard). Calls to *inc* receive s_1, s_2 (corresponding to o and i) as input, and return s_1 to store the incremented i. The computation continues on add_{15}, which stores the top of the stack in i, and calls the loop entry for the next iteration.

2.2. The semantics

Rules in Fig. 2 define an *operational semantics* for the RBR. An *activation record* has the form $\langle p, bc, lv\rangle$, where p is a procedure name, bc is a sequence of instructions, and lv is a variable mapping. Given a variable x, $lv(x)$ refers to the value of

(1) $$\frac{b \equiv x{:=}exp,\quad eval(exp, lv) = v}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv[x \mapsto v]\rangle{\cdot}ar; h}$$

(2) $$\frac{b \equiv x{:=}\mathsf{new}\,c,\quad newobject(c) = o,\quad r\not\in dom(h)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv[x \mapsto r]\rangle{\cdot}ar; h[r \mapsto o]}$$

(3) $$\frac{b \equiv x{:=}y.f,\quad lv(y) \neq \mathsf{null}}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv[x \mapsto h(lv(y)).f]\rangle{\cdot}ar; h}$$

(4) $$\frac{b \equiv x.f{:=}y,\quad lv(x) \neq \mathsf{null},\quad h(lv(x)) = o}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv\rangle{\cdot}ar; h[o.f \mapsto lv(y)]}$$

(5) $$\frac{b \equiv \mathsf{nop}(any)}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv\rangle{\cdot}ar; h}$$

(6) $$\frac{\begin{array}{c}b \equiv q(\bar{x}, y),\quad \text{there is a rule } q(\bar{x}', y'){:=}g, b_1, \cdots, b_k \in RBR,\\ newenv(q){=}lv',\ \forall i.lv'(x_i') = lv(x_i),\ eval(g, lv') = true\end{array}}{\langle p, b{\cdot}bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle q, b_1 \cdots b_k, lv'\rangle{\cdot}\langle p[y', y], bc, lv\rangle{\cdot}ar; h}$$

(7) $$\frac{}{\langle q, \epsilon, lv'\rangle{\cdot}\langle p[y', y], bc, lv\rangle{\cdot}ar; h \rightsquigarrow \langle p, bc, lv[y \mapsto lv'(y')]\rangle{\cdot}ar; h}$$

Fig. 2. Operational semantics of bytecode programs in rule-based form.

x, and $lv[x{\mapsto}v]$ updates lv by making $lv(x) = v$ while lv remains the same for all other variables. A *heap* h is a partial map from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by r in h. We use $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$ while h stays the same for all locations different from r. For any location r and heap h, $r \in dom(h)$ iff there is an object associated to r in h. Given an object o, $o.f$ refers to the value of the field f in o, and $o[f{\mapsto}v]$ sets the value of $o.f$ to v. We use $h[o.f{\mapsto}v]$ as a shortcut for $h(r)[f{\mapsto}v]$, with $o = h(r)$. The class tag of o is denoted by $class(o)$.

Similar to bytecode programs, we assume that RBR programs have been verified for well-typedness. Hence, the types of the variables at each program point are known statically. For clarity, instead of annotating the variables with their types, we assume that, given a variable x, $\mathsf{static\_type}(x)$ denotes its static type (i.e., *int* or reference). In rule (1), $eval(exp, lv)$ returns the evaluation of the arithmetic or boolean expression *exp* for the values of the corresponding variables from *lv* in the standard way; for reference variables, it returns the reference. Rules (2)–(4) deal with objects as expected. Procedure $newobject(c)$ creates a new object of class c by initializing its fields to either 0 or null, depending on their types. Rule (5) is used for ignoring nop-wrapped instructions. Rule (6) (resp., (7)) corresponds to calling (resp., returning from) a procedure. The notation $p[y', y]$ records the association between the formal and actual return variables. *newenv* creates a new mapping of local variables for the method, where each variable is initialized to either 0 or null.

An execution in the RBR starts from an *initial configuration* of the form $\langle start, p(\bar{x}, y), lv\rangle; h$, and ends in a *final configuration* $\langle start, \epsilon, lv'\rangle; h'$, where: (1) *start* is an auxiliary name to indicate an initial activation record; (2) $p(\bar{x}, y)$ is a call to the procedure from which the execution starts; (3) h is an initial heap; and (4) lv is a variable mapping such that $dom(lv) = \bar{x} \cup \{y\}$, and all variables are initialized to an integer value, null or a reference to an object in h. Executions can be regarded as *traces* of the form $C_0 \rightsquigarrow C_1 \rightsquigarrow \cdots \rightsquigarrow C_f$ (abbreviated $C_0 \rightsquigarrow^* C_f$), where C_f is a final configuration. Non-terminating executions have infinite traces. *Confs* denotes the set of all possible configurations.

3. The notion of cost and cost model

This section introduces *cost models* for RBR programs which define the cost to be assigned to an execution step and, by extension, to an entire trace. We concentrate on cost models where the cost of a step only depends on the executed instruction and its input values, since all realistic cost models fall into this category. For this, we first define an operation that eliminates all irrelevant information from a given RBR configuration: for a non-final RBR configuration $C = \langle p, b \cdot bc, lv\rangle{\cdot}ar; h$, we let $rrinput(C) = (b, \langle v_1, \ldots, v_n\rangle)$, where each v_i is the value of the i-th parameter of b in C (we assume that b has a fixed order on its input parameters). Note that the input parameters include also static values such as c in $\mathsf{new}\,c$. The mapping *rrinput* is lifted to sets of configurations as follows: $rrinput(X) = \{rrinput(C) \mid C \in X,\ C \text{ is not final}\}$.

Definition 3.1. An *RBR cost model* $\mathcal{M}$ is a function from $rrinput(Confs)$ to $\mathbb{R}$. For $C \in Confs$, we write $\mathcal{M}(C)$ instead of $\mathcal{M}(rrinput(C))$.

Example 3.2. (1) The cost model $\mathcal{M}_i$ counts the number of instructions by assigning cost 1 to every instruction: $\mathcal{M}_i((b, \langle v_1, \ldots, v_n\rangle)) = 1$. (2) The cost model $\mathcal{M}_h$ counts the amount of (heap) memory consumption:

$$\mathcal{M}_h(I) = \begin{cases} size(c) & \text{if } I \equiv (x{:=}\mathsf{new}\,c, \langle c\rangle) \\ 0 & \text{otherwise} \end{cases}$$

The cost of the execution step $C_1 \rightsquigarrow C_2$ is $\mathcal{M}(C_1)$, and the cost of a trace $\mathcal{M}(t)$ is the sum of the cost of its steps. Since the RBR program is an intermediate representation of the bytecode program whose cost we are interested in, we need to guarantee that the cost counted at the level of RBR corresponds to the actual cost at the level of bytecode. This could be problematic since, for instance, different forms of assignment in the bytecode (e.g., load, store, etc.) have been transformed into an identical assignment instruction in the RBR, while they could contribute a different cost. This is however not a problem in our framework, as we can instrument the RBR program (at the corresponding program points) with $\mathsf{nop}(i)$ instructions where i provides extra information to be considered by the cost model. This information can be used, for example, to distinguish assignments originating from load and store.

4. Cost analysis of rule-based programs

Static program analysis [29,50] is now a well-established technique which has allowed the inference of very sophisticated properties in an automatic and provably correct way. The basic idea in abstract-interpretation-based static analysis is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus obtaining safe approximations of the behavior of the program. The size abstraction we will perform consists in abstracting the instructions by the size constraints they impose on the variables on which they operate. This abstraction is necessary in order to approximate the cost of executing the program. More concretely, given a program P and a cost model $\mathcal{M}$, the classical approach to cost analysis [63] consists in obtaining a set of *Recurrence Relations* (RRs for short) which capture the cost w.r.t. $\mathcal{M}$ of running P on some input $\bar{x}$. Data structures are replaced by their *size* in the RRs. Soundness of the analysis guarantees that, for a concrete input, the execution of the RRs on the size abstraction of such input must return as one of its solutions the actual cost (note that, due to the standard theory of abstract interpretation, this also holds if techniques as *widening* are used).

This section describes how static program analysis can be applied to RBR programs to obtain *Cost Relation Systems* (CRSs), an extended form of RRs, which describe their costs. In our approach, each rule in the RBR program results in an equation in the CRS. For instance, using $\mathcal{M}_i$, the rule defining add_8 gives the following (after simplifying it for readability):

$$\begin{array}{rcl} add_8(th, n, o, res, i) & = & \langle 1, \{s_1'{=}res\}\rangle + \langle 1, \{s_2'{=}i\}\rangle + \langle 1, \{s_1''{=}s_1'{+}s_2'\}\rangle + \\ & & \langle 1, \{res'{=}s_1''\}\rangle + \langle 1, \{s_1'''{=}o\}\rangle + \langle 1, \{s_2''{=}i\}\rangle + \\ & & \langle 1, \{true\}\rangle + \langle add_8^c(th, n, o, res', i, s_1''', s_2''), \{r' = \_\}\rangle \end{array}$$

Here, variables are constraint variables corresponding to those of the original rule; e.g., s_1' and s_1'' both correspond to values of s_1, but at different program points. Each pair $\langle e, \varphi\rangle$ in the right hand side of the equation corresponds to an instruction in the original rule: e is the cost of executing that instruction, and φ is the effect of the instruction on the variables (in terms of linear constraints). The last pair is the cost of running add_8^c on input $th, n, o, res', i, s_1'''$ and s_2''. The constraint $r' = \_$ is the effect of calling add_8^c on the local variables and it indicates that we do not obtain any relation between the input and the output variables. The equation can be simplified by merging the pairs into:

$$\begin{array}{rcl} add_8(th, n, o, res, i) & = & \langle 7, \{res' = res + i, s_1'''{=}o, s_2''{=}i\}\rangle + \\ & & \langle add_8^c(th, n, o, res', i, s_1''', s_2''), \{r' = \_\}\rangle \end{array}$$

which states that, given values (sizes) for th, n, o, res, and i, the cost of executing $add_8(th, n, o, res, i)$ is 7 units plus the cost of $add_8^c(th, n, o, res', i, s_1''', s_2'')$. Cost equations are generated for each RBR rule as follows:

1. *size measures* are chosen to represent information relevant to cost (Section 4.1) in order to abstract variables to their *size*. E.g., a list is abstracted to its length, since this gives information about the cost of traversing it.
2. Instructions are replaced by *linear constraints* (Section 4.2) approximating the relation between states w.r.t. the size measures. E.g., $s_1{:=}o$ is replaced by $s_1'''{=}o$, meaning that the size of s_1 after the assignment (represented by s_1''') is equal to the size of o.
3. Output variables are removed from the rules by inferring the relation between the input and the output using *input–output size relations* (Section 4.3). This is why there is no argument r in the above equation (see Example 2.2).
4. Finally, a CRS is obtained by using the abstract rules, the original rules, and the selected cost model to generate *cost expressions* representing the cost w.r.t. the model (Section 4.4). In the above example, the cost expressions are the constants, corresponding to $\mathcal{M}_i$.

As notation, a *linear expression* takes the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$, where q_i are rational numbers, and x_i are variables. A *linear constraint* (over integers) takes the form $l_1\ op\ l_2$, where l_1 and l_2 are linear expressions, and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints, interpreted as a conjunction. The statement $\varphi_1 \models \varphi_2$ indicates that φ_1 implies φ_2. An *assignment* σ maps (constraint) variables to values in $\mathbb{Z}$, and $\sigma \models \varphi$ denotes that σ is a consistent assignment for φ, i.e., $\bigwedge\{x = \sigma(x) \mid x \in dom(\sigma)\} \models \varphi$. Given φ_1 and φ_2, $\varphi_1 \sqcup \varphi_2$ is their *convex-hull* [30]. We use $\varphi|_S$ to denote projection of φ on the set of variables S, i.e., eliminating all variables $vars(\varphi) \setminus S$ using, for example, Fourier–Motzkin elimination. In our system, we rely on [16] for manipulating linear constraints. We use $a \ll_c A$ to indicate that an entity a is a renamed apart (from c) element of A, i.e., we choose an element from A and then rename its variables such that it does not share any variable with c.

4.1. The notion of size measure

For the purpose of cost analysis, data structures are usually abstracted into their size. Beginning with [63], several *size measures* or *norms* have been proposed in cost and termination analysis (see, e.g., [23] and its references). The choice of a

b	b^α with respect to a renaming ρ	ρ'
x:=exp	$b^\alpha \equiv \rho(x)'$=exp^α	$\rho[x\mapsto\rho(x)']$
x:=new c	$b^\alpha \equiv \rho(x)'$=1	$\rho[x\mapsto\rho(x)']$

x:=$y.f$	if f is a numeric field: $b^\alpha \equiv \rho(x)'$=_
	if f is a reference field and y is acyclic:
	$b^\alpha \equiv \rho(y) > \rho(x)' \wedge \rho(x)' \geq 0$
	if f is a reference field and y might be cyclic:
	$b^\alpha \equiv \rho(y) \geq \rho(x)' \wedge \rho(x)' \geq 0$
		$\rho[x\mapsto\rho(x)']$

$x.f$:=y	if f is a reference field and $y \notin \mathsf{SH}_x$:
	$S = \{v \mid v \in \mathsf{SH}_x\}$ and
	$b^\alpha \equiv \wedge\{\rho(v)' \leq \rho(v) + \rho(y) \wedge \rho(v)' \geq 0 \mid v \in S\}$
	if f is a reference field and $y \in \mathsf{SH}_x$:
	$S = \{v \mid v \in \mathsf{SH}_x\}$ and $b^\alpha \equiv \wedge\{\rho(v)' \geq 0 \mid v \in S\}$
	if f is a numeric field: $S = \emptyset$ and $b^\alpha \equiv true$
		$\rho[\forall v \in S.$
		$v\mapsto\rho(v)']$

$p(\bar{x}, y)$	$b^\alpha \equiv \langle p(\rho(\bar{x}), \rho(y)'), \varphi_1 \wedge \varphi_2\rangle$ where: U_p=$\{v \mid v \in \bar{x}$, v might be updated in p $\}$ S=$\{v \mid x \in U_p, v \in \mathsf{SH}_x\}$ φ_1= $\wedge$ $\{\rho(v)' \geq 1 \mid v{\in}S$ not null before call$\}$ φ_2= $\wedge$ $\{\rho(v)' \geq 0 \mid v{\in}S$ maybe null before call$\}$	$\rho[\forall v{\in}S.$ $v\mapsto\rho(v)'$, $y\mapsto\rho(y)']$
type(x, c)	$b^\alpha \equiv x \geq 1$	ρ
$exp_1 \otimes exp_2$	$b^\alpha \equiv exp_1^\alpha \otimes exp_2^\alpha$	ρ
null	$b^\alpha \equiv 0$	ρ
x	$b^\alpha \equiv \rho(x)$	ρ
$otherwise$	$b^\alpha \equiv true$	ρ

Fig. 3. Abstract compilation of instructions.

measure, especially for heap structures, heavily depends on the program. E.g., in termination analysis, norms should describe something which strictly decreases at each loop iteration. For a list traversed by a loop, a typical example of measure is its length, which is used to bound the number of iterations. For an integer i, the actual numerical value can be a good measure to bound the number of iterations of loops with counter i.

Definition 4.1. Given a configuration $C \equiv \langle p, bc, lv\rangle \cdot ar; h$, the size of $x \in dom(lv)$ with respect to a static type stype is defined as:

$$\alpha(x, \mathsf{stype}, C) = \begin{cases} lv(x) & \text{if stype is } int \\ \mathsf{path\text{-}length}(lv(x), h) & \text{if stype is a reference type} \end{cases}$$

path-length corresponds to Def. 5.1 in [59]. It takes a heap h and a reference $lv(x) \in dom(h)$, and returns the length of the maximal *path* reachable from that reference by *dereferencing*, i.e., following other references stored as fields. The path-length of null is 0, and that of a *cyclic* data structure is ∞. Note that, due to lack of space, our language does not include *arrays*; however, they are supported in our system, and are abstracted to their length.

4.2. Abstract compilation

This section describes how to transform a rule-based program P into an abstract program P^α, which can be seen as an abstraction of P w.r.t. the size measure α. The translation is based on replacing each instruction by (linear) *constraints* which describe its behavior with respect to the size measure. E.g., x:=new c can be replaced by x=1, meaning that the length of the maximal path starting from x is 1. For simplicity, the same name (possibly primed) is used for the original variables and their sizes, i.e., given a list l, the name l also denotes its path-length (in P^α). Letting α denote the size measure of Definition 4.1, the translation of the instructions in the RBR is depicted in Fig. 3.

The presented setting is able to obtain relations between the *size* of a variable at different program points. E.g., when analyzing x:=$x + 1$, the interest can be in the relation "*the value of x after the instruction is equal to the value of x before the instruction plus 1*". This important information is obtained via a *Static Single Assignment* (SSA) transformation, which, together with the abstract compilation, produces x'=$x + 1$, where x and x' refer to, resp., the value of x before and after the instruction. To implement the SSA transformation, a mapping ρ (a *renaming*) of variable names (as they appear in the rule) to new variable names (constraint variables) is maintained. The expression $\rho[x \mapsto y]$ denotes the update of ρ, such that it maps x to the new variable y. The use of path-length as a size measure for references requires extra information to obtain precise and sound results: (a) *sharing* information [57] tells whether two variables might point (either directly, by *aliasing*, or indirectly) to a common heap location; and (b) *acyclicity* information [55] guarantees that, at some program point, a reference points to an acyclic data structure.

For the instruction x:=$y.f$ of Fig. 3: (1) for numeric fields, all information is lost; i.e., b is abstracted to $\rho(x)' =$ _ where _ is assumed to be a constraint variable not used anywhere else. In practice we use [6] for handling numeric fields. Note that, if _ appears several times, then each occurrence is assumed to be a different constraint variable; (2) for reference fields, if y is acyclic, then b is abstracted to $\rho(y) > \rho(x)'$, since the longest path reachable from y is longer than the longest path reachable from x; otherwise $\rho(y) \geq \rho(x)'$. As for $x.f$:=y, when f is a reference field, if x and y do not share, the length of the maximal path reachable from x and any variable sharing with x (SH_x is the set of variables which might share with x before the instruction, including x) might change. This change can be safely described by $\rho(v)' \leq \rho(v)+\rho(y) \wedge \rho(v)' \geq 0$, where v is a variable in SH_x. If x and y share, no safe information can be provided, it can only be said that the size is non-negative [59]. When f is a numeric field, the path-length property of x does not change, so that b is abstracted to *true*.

The abstraction of calls $p(\bar{x}, y)$ to procedures (or methods) requires computing the set U_p of the input variables pointing to data structures which may be updated by p. This set can be approximated by *constancy analysis* [36]. Note that *updating* refers to actually modifying the *structure* of the heap. If only numeric fields are modified, then the changes are not considered as updates, and the path-length is preserved. The set of updated input variables (closed under sharing), denoted by S in the figure, is renamed in order to *forget* them after the call (i.e., not to propagate the constraints involving S before the call to the state after the call). For instance, consider the call $p(x, z, r)$, and assume that z is updated by p. Let ψ be a constraint over x and z which holds before the call. Then, a fresh variable z' must be used after the call instead of z in order to distinguish between the path-length of z before and after running p. For such argument, we can still say that the final size of every x possibly sharing with it is: (a) *positive*, if x is certainly non-null; or (b) *non-negative*, otherwise. Our implementation includes a simple *nullity analysis* which can verify this condition. A newly-created object can be always guaranteed to be non-null before calling its constructor. The abstraction of calls can be improved by using *shallow variables* for the arguments. They are extra variables which are only used to record the initial value of the arguments (and are never modified), and allow inferring more precise input–output relations. This well-known technique can improve the precision, at the cost of a higher computational effort.

Note that in Fig. 3, when accessing a numeric field, we use a constraint of the form $\rho(x)' = \_$. In principle, this is equivalent to *true*; i.e., it states that we cannot provide any abstract information on the corresponding instruction. However, for the correctness of Lemma 4.4, if the abstraction of b starting from a renaming ρ_1 results in b^α and ρ_2, then it is essential that $\rho_2(x)$ appears in b^α for any x for which $\rho_1(x) \neq \rho_2(x)$. This would be also the case of (numeric) array accesses, since arrays (which are not part of the language in this paper) are abstracted to their length. Likewise in the case of *non-linear* arithmetic such as $x{:=}y * z$ and $x{:=}y/z$, as linear constraints cannot approximate their behavior. In our system, *constant propagation* analysis is applied in order to identify when y or z are constants, thus improving the precision.

Sharing and acyclicity information is precise only if computed w.r.t. a *context* (e.g., an initial state). Therefore, the soundness of the transformation is guaranteed under an initial Sharing-Acyclicity context description. In practice, if the initial call is a Java-like main method, then such an initial Sharing-Acyclicity description is not required, as all data structures are created at runtime, instead of being provided as an input. An initial Sharing-Acyclicity context description takes the form $Q \equiv \langle p(\bar{x}), \mathsf{SH}, \mathsf{ACY}\rangle$ (output variables are ignored), where $\mathsf{SH} \subseteq \bar{x} \times \bar{x}$, and $\mathsf{ACY} \subseteq \bar{x}$. A statement $(x, y) \in \mathsf{SH}$ means that x and y might share, and $x \in \mathsf{ACY}$ means that x certainly points to an acyclic data structure. An initial configuration $\langle start, p(\bar{x}, y), lv\rangle; h$ is said to be safely approximated by Q if: (1) if $x, y \in dom(lv)$ share a common region on h, then $(x, y) \in \mathsf{SH}$; and (2) if $x \in dom(lv)$ points to a cyclic data structure, then $x \notin \mathsf{ACY}$. The information contained in SH and ACY is propagated by means of fixpoint computations, as described by, respectively, [57,55]. Essentially, such analyses provide the information which is required in order to answer the (program point) queries about sharing and acyclicity in Fig. 3.

Definition 4.2. Let $r \equiv p(\bar{x}, y) \leftarrow g, b_1, \ldots, b_n$, and ρ_1 be the *identity renaming* over $vars(r)$. The abstract compilation of r with respect to a size measure α is $r^\alpha \equiv p(\bar{x}, y') \leftarrow \varphi_0 \mid b_1^\alpha, \ldots, b_n^\alpha$ where:

1. g^α is the abstract compilation of g with respect to the renaming ρ_1;
2. $\varphi_0 = \{\rho_1(z) = 0 \mid z \in vars(r) \setminus \bar{x}\} \wedge g^\alpha$;
3. b_i^α is the abstract compilation of b_i using ρ_i;
4. ρ_{i+1}, $1 \leq i \leq n$, is generated from ρ_i and b_i as shown in Fig. 3;
5. $y' = \rho_{n+1}(y)$.

Given an RBR program P, an initial Sharing-Acyclicity context description Q, and a size measure α, P^α is the program obtained by abstracting all its rules using the sharing, acyclicity and constancy information induced by Q.

Note that the Sharing-Acyclicity context Q in the above definition is used to compute the sharing, acyclicity, and constancy information used in Fig. 3 (which is used in point 3 of the above definition). When generating a rule, we also produce a tuple of renamings $\rho = \langle\rho_1, \ldots, \rho_{n+1}\rangle$ which can be used (e.g., in Definition 4.15) to relate program variables to their corresponding constraint variables at each program point. For simplicity, we do not include ρ as part of the rule; rather, we assume that it can be retrieved when needed.

Example 4.3. The rule on the left is abstracted to the rule on the right.

$add_8(th, n, o, res, i, r) \leftarrow$	$add_8(th, n, o, res, i, \rho_9(r)) \leftarrow$
	$\{s_1{=}0, s_2{=}0, r{=}0\} \mid$
$s_1{:=}res,$	$\{s_1'{=}res,$
$s_2{:=}i,$	$s_2'{=}i,$
$s_1{:=}s_1 + s_2,$	$s_1''{=}s_1'{+}s_2',$
$res{:=}s_1,$	$res'{=}s_1''$
$s_1{:=}o,$	$s_1'''{=}o,$
$s_2{:=}i,$	$s_2''{=}i,$
nop(invokevirtual A.inc(I)I),	$true\},$
$add_8^c(th, n, o, res, i, s_1, s_2, r).$	$\langle add_8^c(th, n, o, res', i, s_1''', s_2'', r'), true\rangle.$

The renaming $\rho{=}\langle\rho_1,\ldots,\rho_9\rangle$ used in the translation is as follows: ρ_1 is the identity on $\{this,n,o,res,i,s_1,s_2\}$; $\rho_2{=}\rho_1[s_1{\mapsto}s_1']$; $\rho_3{=}\rho_2[s_2{\mapsto}s_2']$; $\rho_4{=}\rho_3[s_1{\mapsto}s_1'']$; $\rho_5{=}\rho_4[res{\mapsto}res']$; $\rho_6{=}\rho_5[s_1{\mapsto}s_1''']$; $\rho_7{=}\rho_6[s_2{\mapsto}s_2'']$; $\rho_8{=}\rho_7$; and $\rho_9{=}\rho_8[r{\mapsto}r']$. Note that variables which do not appear in the head are initialized in the body (first condition in Definition 4.2). E.g., when abstracting $s_1{:=}s_1+s_2$, according to Fig. 3, ρ_3 contains $s_1\mapsto s_1'$ and $s_2\mapsto s_2'$, introduced by compiling, resp., $s_1{:=}res$ and $s_2{:=}i$. First, such renaming is applied to $s_1{+}s_2$, which leads to $s_1'{+}s_2'$. Next, the abstract compilation of the expression (first row in Fig. 3) produces $s_1''{:=}s_1'+s_2'$, and adds $s_1\mapsto s_1''$ to ρ_3, generating ρ_4. In the above rule, the variable o stands for a reference, and as it is not updated in add_8^c, there is no renaming.

An abstract RBR abstracts the behavior of a program w.r.t. α. Its operational semantics is given by the following transition system:

$$\frac{p(\bar{x},y)\leftarrow\varphi\mid b_1^\alpha,\ldots,b_n^\alpha\ll_{AC}P^\alpha,\quad \psi\wedge\varphi\not\models false}{\langle\langle p(\bar{x},y),\phi\rangle\cdot bc^\alpha,\psi\rangle\leadsto_\alpha\langle b_1^\alpha\cdots b_n^\alpha\cdot\phi\cdot bc^\alpha,\psi\wedge\varphi\rangle}\qquad\frac{\psi\wedge\varphi\not\models false}{\langle\varphi\cdot bc^\alpha,\psi\rangle\leadsto_\alpha\langle bc^\alpha,\psi\wedge\varphi\rangle}$$

where $AC=\langle\langle p(\bar{x},y),\phi\rangle\cdot bc^\alpha,\psi\rangle)$. Note that the renaming in the leftmost transition is required in order to avoid name clashes between constraints variables. Hence, we always rename the rule (using $\ll_{AC}$) by using fresh variables that have not been used before. The next lemma states the *soundness* of the abstract compilation, i.e., that the size of variables in a concrete trace can be observed in the abstract trace. For this, we prove that, given a concrete trace, we can generate an abstract trace of the same length and instantiate it (i.e., give integer values to all constraint variables using a consistent assignment σ) in such a way that the size of a variable in the i-th concrete state coincides with the value of the corresponding constraint variable in the i-th abstract state. Given an initial configuration $C_0=\langle start,p(\bar{x},y),lv_0\rangle;h$, we let $\alpha(C_0)$ be $\bigwedge\{z=\alpha(z,\mathsf{static\_type}(z),C_0)\mid z\in\bar{x}\cup\{y\}\}$, where $z=\infty$ is interpreted as $z=\_$.

Lemma 4.4. *Let P be an RBR program,* $C_0{=}\langle start,p(\bar{x},y),lv_0\rangle;h$, $\varphi_0=\alpha(C_0)$, $Q=\langle p(\bar{x}),SH,ACY\rangle$ *a safe Sharing-Acyclicity description of* C_0, *and* P^α *the corresponding abstract program w.r.t. Q. The following holds: If* $C_0\leadsto^n C_n$ *is a concrete trace of P, then there exist an abstract trace* $AC_0\leadsto_\alpha^n AC_n$ *where* $AC_0=\langle p(\bar{x},y),\varphi_0\rangle$ *and* $AC_n=\langle\_,\varphi_n\rangle$, *a partial map* $f: vars(P)\times\{0,\ldots,n\}\mapsto vars(AC_n)$, *and a consistent assignment* $\sigma: vars(AC_n)\mapsto\mathbb{Z}$ *for* φ_n, *such that: for any* $C_i=\langle\_,\_,lv_i\rangle\cdot ar_i;h_i$ *and* $AC_i=\langle\_,\varphi_i\rangle$ $(0\le i\le n)$, *it holds that* $\varphi_n\models\varphi_i$; $\forall z\in dom(lv_i).\alpha(z,\mathsf{static\_type}(z),C_i)=\sigma(f(z,i))$.

The above lemma states that each (abstract) state AC_i is a safe approximation (w.r.t. α) of the corresponding activation record in C_i. The claim that $\varphi_n\models\varphi_i$ is straightforward since abstract traces basically accumulate the constraint by means of conjunction. A partial map f is used to relate program variables to their corresponding constraint variables. This mapping can be constructed by collecting the renamings (enriched with a state index) of the abstract rules used during the evaluation. We use "_" in order to indicate parts of an entity that we are not interested in, instead of assigning them names that will not be used.

4.3. Input–output size relations

CRSs are *mathematical relations*, in the same way as RRs are mathematical functions. Hence, they cannot have output variables: instead, they should receive a set of input parameters and *return a number*. This step of the analysis is meant to transform the abstract program P^α into one where output variables do not appear. The basic idea relies on computing abstract *input–output (size) relations* in terms of linear constraints, and using them to propagate the effect of calling a rule. We consider the abstract rules obtained in the previous step to approximate the input–output ("io" in abbreviations) relation for blocks. Concretely, we infer io size relations of the form $p(\bar{x},y)\to\varphi$, where φ is a constraint describing the relation between the size of the input $\bar{x}$ and the output y upon exit from p. Input–output size relations are needed in order to eliminate output variables without losing relevant information, since the output of one call may be input to another call. Consider the following abstract rule:

$$p(x,y')\leftarrow\{w{=}0,z{=}0,y{=}0\}\mid x{>}0,z'{=}x{-}1,\langle q(z',w'),true\rangle,\langle p(w',y'),true\rangle$$

Assuming that $q(z',w')$ will generate $z'{\ge}w'$, this rule becomes:

$$p(x)\leftarrow\{w{=}0,z{=}0,y{=}0\}\mid x{>}0,z'{=}x-1,\langle q(z'),z'\ge w'\rangle,\langle p(w'),\{y'=\_\}\rangle$$

which does not have output arguments. Importantly, this makes it possible to infer $x{>}w'$, which is crucial for bounding the number of iterations. The next definition introduces the notion of io relations, which can be seen as a denotational semantics for the abstract programs of Section 4.2. The definition is based on a semantic operator $\mathcal{T}_{P^\alpha}$ which describes how, from a set of io relations I, we learn more relations by applying the rules in the abstract program.

Definition 4.5 (*Input–Output Relations*). Let the operator $\mathcal{T}_{P^\alpha}(I)$ be

$$\left\{p(\bar{x},y)\to\psi\;\middle|\;\begin{array}{ll}(1) & r=p(\bar{x},y)\leftarrow\varphi\mid b_1^\alpha,\ldots,b_n^\alpha\in P^\alpha\\ (2) & \forall\,1\le i\le n,\ \text{ either}\\ & \quad 2.1)\ b_i^\alpha \text{ is a constraint } \varphi_i;\text{ or}\\ & \quad 2.2)\ b_i^\alpha{=}\langle q_i(\bar{w}_i,z_i),\phi_i\rangle \text{ where } q_i(\bar{w}_i,z_i)\to\psi_i'\in I\\ & \qquad\ \text{and we let } \varphi_i=\phi_i\wedge\psi_i'\\ (3) & \psi=(\varphi\wedge\varphi_1\wedge\cdots\wedge\varphi_n)|_{\bar{x}\cup\{y\}}\end{array}\right\}$$

The *input–output relations* of an abstract program P^α, denoted by $\mathcal{I}(P^\alpha)$, are defined as $\bigcup_{i\in\omega}\mathcal{T}^{i}_{P^\alpha}(\emptyset)$, where $\mathcal{T}^{i}_{P^\alpha}(I) = \mathcal{T}_{P^\alpha}(\mathcal{T}^{i-1}_{P^\alpha}(I))$ and $\mathcal{T}^{0}_{P^\alpha}(I) = I$.

Computing $\mathcal{I}(P^\alpha)$ is often impractical, as it might include an infinite number of objects. However, it can be approximated using abstract interpretation techniques [29]. In particular, by using a *convex-hull* operator $\sqcup$ instead of $\cup$, and incorporating a *widening* operator to guarantee termination [30].

Example 4.6. The following io relations are obtained from the corresponding procedures in the RBR of the running example:

$$A.inc(th,i,r)\rightarrow\{r{=}i{+}1\}\quad B.inc(th,i,r)\rightarrow\{r{=}i{+}2\}\quad C.inc(th,i,r)\rightarrow\{r{=}i{+}3\}$$

In all cases, the output variable r is related only to the input variable i. This piece of information will be crucial for inferring the cost.

The following lemma, which establishes the correctness of the io size relations, is well-known in the context of logic programming [18].

Lemma 4.7. *Let P^α be an abstract program. If $t = \langle\langle p(\bar{x},y),\phi\rangle,\psi_0\rangle \rightsquigarrow^{*}_{\alpha} \langle\epsilon,\psi\rangle$ is an abstract trace, then there exists $p(\bar{x},y)\rightarrow\varphi\in\mathcal{I}(P^\alpha)$ s.t. $\psi\models\varphi$.*

Our framework only requires a *safe* approximation of the io relations, as the next definition states.

Definition 4.8. The set A is a safe approximation of the input–output relations of a program P^α iff, for any $a = p(\bar{x},y)\rightarrow\varphi\in\mathcal{I}(P^\alpha)$, there exists $p(\bar{x},y)\rightarrow\psi\in A$ such that $\varphi\models\psi$.

For simplicity, A is supposed to contain only one io relation $p(\bar{x},y)\rightarrow\psi$ for every p. This can be done by *merging* all the relations of p using $\sqcup$. In addition, for simplifying the correctness claim in Theorem 4.17, we assume that $y\in vars(\psi)$. This can be achieved by simply adding $y = \_$ to ψ where $\_$ is a new variable.

The following definition describes how to remove the output variables from P^α by using a safe approximation of the io relations: for each call $p(\bar{w},z)$ in a rule r, $p(\bar{w},z)\rightarrow\varphi\in A$ is used in order to eliminate z, but still propagate its relation (φ) with $\bar{w}$ generated by the execution of p.

Definition 4.9. Given P^α and a safe approximation A of its input–output relations, P^{io} denotes the abstract program generated from the rules of P^α, as follows: each rule $r = p(\bar{x},y)\leftarrow\varphi\mid b^\alpha_1,\ldots,b^\alpha_n\in P^\alpha$ is replaced by $p(\bar{x})\leftarrow\varphi\mid b^{io}_1,\ldots,b^{io}_n$, where (1) if $b^\alpha_i{=}\langle q(\bar{w},z),\varphi_i\rangle$, then $b^{io}_i{=}\langle q(\bar{w}),\varphi_i\wedge\psi\rangle$, where $q(\bar{w},z)\rightarrow\psi\in A$; and (2) if b^α_i is a constraint, then $b^{io}_i{=}b^\alpha_i$.

Example 4.10. Using the relations of Example 4.6, eliminating the output variables of the rules $add_{14:A}$, $add_{14:B}$ and $add_{14:C}$ (Example 2.2) results in:

$$\begin{array}{l}
add_{14:A}(th,n,o,res,i,s_1,s_2)\leftarrow\\
\quad\{r{=}0,s_1{=}0,s_2{=}0\}\mid\langle A.inc(s_1,s_2),\{s_1'{=}s_2{+}1\}\rangle,\langle add_{15}(th,n,o,res,i,s_1'),\{r'{=}\_\}\rangle.\\
add_{14:B}(th,n,o,res,i,s_1,s_2)\leftarrow\\
\quad\{r{=}0,s_1{=}0,s_2{=}0\}\mid\langle B.inc(s_1,s_2),\{s_1'{=}s_2{+}2\}\rangle,\langle add_{15}(th,n,o,res,i,s_1'),\{r'{=}\_\}\rangle.\\
add_{14:C}(th,n,o,res,i,s_1,s_2)\leftarrow\\
\quad\{r{=}0,s_1{=}0,s_2{=}0\}\mid\langle C.inc(s_1,s_2),\{s_1'{=}s_2{+}3\}\rangle,\langle add_{15}(th,n,o,res,i,s_1'),\{r'{=}\_\}\rangle.
\end{array}$$

Note that $r' = \_$ has been added to make the output variable appear explicitly when the io relation is *true*.

The generated abstract rules can be executed by using the following transition system. They are identical to the execution of the abstract rules explained in Section 4.2 (here, $AC{=}\langle\langle p(\bar{x}),\phi\rangle{\cdot}bc^{io},\psi\rangle$) , but without have output variables:

$$\frac{p(\bar{x})\leftarrow\varphi\mid b^{io}_1,\ldots,b^{io}_n\ll_{AC}P^{io},\ \psi\wedge\varphi\not\models false}{\langle\langle p(\bar{x}),\phi\rangle{\cdot}bc^{io},\psi\rangle\rightsquigarrow_{io}\langle b^{io}_1\cdots b^{io}_n\cdot\phi{\cdot}bc^{io},\psi{\wedge}\varphi\rangle}\qquad\frac{\psi\wedge\varphi\not\models false}{\langle\varphi{\cdot}bc^{io},\psi\rangle\rightsquigarrow_{io}\langle bc^{io},\psi{\wedge}\varphi\rangle}$$

The next lemma states the soundness of this step: intuitively, the result (in terms of constraints) of executing the abstract rules without output variables (but with io relations) is a safe approximation of the execution of the abstract rules with output variables.

Lemma 4.11. *Let P^α be an abstract program, and P^{io} be its corresponding program generated (following Definition 4.9) with respect to a safe approximation A of its input–output size relations. Then, if $AC_0\rightsquigarrow^{n}_{\alpha}AC_n$ is a trace in P^α where $AC_0 = \langle\langle p(\bar{x},y),\phi\rangle,\varphi_0\rangle$, then there is an abstract trace $AC'_0\rightsquigarrow^{n}_{io}AC'_n$ in P^{io} such that: (1) $AC'_0 = \langle\langle p(\bar{x}),\phi\wedge\psi\rangle,\varphi_0\rangle$, where $p(\bar{x},y)\rightarrow\psi\in A$; and (2) for any $AC_i = \langle\_,\varphi_i\rangle$ and $AC'_i = \langle\_,\varphi'_i\rangle$ $(0\le i\le n)$, it holds that $\varphi_i\models\varphi'_i$.*

4.4. Building cost relation systems

This section presents the automatic generation of *cost relation systems* (CRSs) which capture the cost of executing a bytecode method w.r.t. a cost model. CRSs are generated by incorporating *symbolic cost expressions* into the abstract rules.

Definition 4.12. A *symbolic cost expression* exp is defined as follows

$$\mathsf{exp} \quad ::= \quad n \mid x \mid \mathsf{exp}\ op\ \mathsf{exp} \mid \mathsf{exp}^{\mathsf{exp}} \mid \log_a(\mathsf{exp}) \qquad op \in \{+, -, /, *\}$$

where $a \in \mathbb{N}, a > 1$; n is real and positive; and x is an integer variable. The set of all symbolic cost expressions is denoted by *Exprs*.

Symbolic cost expressions are used for two purposes: (1) to count the resources we accumulate in the different cost models, thus, to define the cost relation systems; e.g., in Example 3.2, when we estimate memory consumption, we can obtain a symbolic cost expression where the object size is a variable; (2) to describe the *closed-form solutions* (or upper bounds) of cost relations. The above definition shows that we aim at covering a wide range of *complexity classes*: in addition to *polynomial* cost expressions, also *exponential* and *logarithmic* expressions (and any combination of them) are handled.

Definition 3.1 needs to be adapted to the symbolic level: given an instruction, a symbolic cost model returns a symbolic expression instead of a constant value.

Definition 4.13. Let α be the size measure (Definition 4.1), and $\mathcal{M}$ be a cost model (Definition 3.1). The partial map $\mathcal{M}^s : Instr \mapsto Exprs$ is said to be a *symbolic cost model* for $\mathcal{M}$ iff, for any $C{=}\langle m, b \cdot bc, lv\rangle \cdot ar; h$: if $e = \mathcal{M}^s(b)$, then $e[\forall x \in vars(e) \mapsto \alpha(x, \mathsf{static\_type}(x), C)] = \mathcal{M}(C)$.

Intuitively, given a configuration such that b is the next instruction to be executed, the *evaluation* of the symbolic expression $\mathcal{M}^s(b)$, must be equal to applying $\mathcal{M}$ to the configuration. Note that the definition of cost model depends only on the input values of a given instruction. Thus, if a cost model involves only (linear) arithmetic expressions over the input variables (which is the case of realistic cost models), one can generate a corresponding symbolic model by replacing the reference to the i-th input value by its constraint variable.

Example 4.14. The symbolic version of $\mathcal{M}_h$ (Example 3.2), is defined as follows:

$$\mathcal{M}_h^s(b) = \begin{cases} size(c) & b \equiv x{:=}\mathsf{new}\ c \\ 0 & \text{otherwise} \end{cases}$$

To understand the relation between a cost model and its corresponding symbolic model, assume that our language includes an instruction $x{:=}\mathsf{newarray}(int, y)$ for creating an array of size y whose elements are of type int. The cost model $\mathcal{M}_h$ would map such instruction to $size(int) * v$ where v is the input value that corresponds to y, i.e. $v = lv(y)$, and $size(int)$ is the space required for storing a value of type int. The symbolic cost model would map such instruction to $size(int) * y$ which is obtained by replacing v by y. For the case of $\mathcal{M}_i$, its corresponding symbolic cost model assigns 1 to each instruction.

Definition 4.15. Let P be the rule-based representation of P. Consider a rule $r{\equiv}\ p(\bar{x}, y) \leftarrow g, b_1, \ldots, b_n \in P$, its abstract compilation (after eliminating the output variables from r^α) $r^{io} \equiv p(\bar{x}) \leftarrow \varphi \mid b_1^{io}, \ldots, b_n^{io} \in P^{io}$, and a symbolic cost model $\mathcal{M}^s$. The cost equation of r is $r^{eq} \equiv p(\bar{x}) = \varphi \mid b_1^{eq} + \cdots + b_n^{eq}$, where: (1) if $b_i^{io} = \langle q(\bar{w}), \phi\rangle$, then $b_i^{eq} = \langle q(\bar{w}), \phi\rangle$; and (2) if $b_i^{io} = \varphi_i$ then $b_i^{eq} = \langle \rho_i(\mathcal{M}^s(b_i)), \varphi_i\rangle$, where $\rho = \langle \rho_1, \ldots, \rho_{n+1}\rangle$ is the renaming used to generate r^α according to Definition 4.2. P_{cr} is the cost relation system consisting of the cost equations obtained from P.

Essentially, the output of cost analysis is the above CRS, i.e., a set of *recursive* equations which have been generated from the program structure by inferring size relations between its arguments. Importantly: (1) size relations between the rule variables are associated to the cost equations (at different points) to describe how the size of data changes when the equations are applied; and (2) guards do not affect the cost: they are simply used to define the applicability conditions of the equations.

CRSs are powerful as they are not limited to any complexity class. E.g., they can capture the cost of exponential methods with several recursive calls, or logarithmic methods where the size of the data is halved at every loop iteration.

Example 4.16. Consider the RBR of Example 2.2, and the size relations derived by size analysis (Example 4.6). By applying Definition 4.15 w.r.t. the symbolic cost model $\mathcal{M}_i^s$, the CRS in Fig. 4 is obtained. The constraint $r' = \_$ is added just to make r' appear syntactically in the rules. Note that it has been simplified to make it more readable: (1) Some input arguments are written as $\bar{x}_1, \bar{x}_2$ and $\bar{x}_3$, where each $\bar{x}_i$ is defined at the bottom of the figure; (2) constraints which stem from the implicit variable initialization (to 0 or null) are ignored; (3) "*true*" guards are omitted; (4) if possible, consecutive pairs $\langle e, \varphi\rangle$ are grouped together (e.g., in add_8, we grouped together pairs with a constant cost); (5) constraints are simplified, e.g., equalities $x = y$ have been eliminated by unifying the corresponding variables.

$$\begin{array}{ll}
add(th,n,o) = \langle add_1(th,n,o,res,i), \{r' = \_\}\rangle & \\
add_1(\bar{x}_3) & = \langle 4, \{res'{=}0, i'{=}0\}\rangle + \langle add_5(th,n,o,res',i'), \{r' = \_\}\rangle \\
add_5(\bar{x}_3) & = \langle 3, \{s_1'{=}i, s_2'{=}n\}\rangle + \langle add_5^c(\bar{x}_3, s_1', s_2'), \{r' = \_\}\rangle \\
add_5^c(\bar{x}_1) & = \{\mathbf{s_1} > \mathbf{s_2}\} \mid \langle add_{17}(\bar{x}_3), \{r' = \_\}\rangle \\
add_5^c(\bar{x}_1) & = \{\mathbf{s_1} \le \mathbf{s_2}\} \mid \langle add_8(\bar{x}_3), \{r' = \_\}\rangle \\
add_{17}(\bar{x}_3) & = \langle 2, \{r'{=}res\}\rangle \\
add_8(\bar{x}_3) & = \langle 7, \{res'{=}res{+}i, s_1'''{=}o, s_2''{=}i\}\rangle + \langle add_8^c(th,n,o,res',i,s_1''',s_2''), \{r' = \_\}\rangle \\
add_8^c(\bar{x}_1) & = \langle add_{14:A}(\bar{x}_1), \{r' = \_\}\rangle \\
add_8^c(\bar{x}_1) & = \langle add_{14:B}(\bar{x}_1), \{r' = \_\}\rangle \\
add_8^c(\bar{x}_1) & = \langle add_{14:C}(\bar{x}_1), \{r' = \_\}\rangle \\
add_{14:A}(\bar{x}_1) & = \langle A.inc(s_1,s_2), \{s_1'{=}s_2{+}1\}\rangle + \langle add_{15}(th,n,o,res,i,s_1'), \{r' = \_\}\rangle \\
add_{14:B}(\bar{x}_1) & = \langle B.inc(s_1,s_2), \{s_1'{=}s_2{+}2\}\rangle + \langle add_{15}(th,n,o,res,i,s_1'), \{r' = \_\}\rangle \\
add_{14:C}(\bar{x}_1) & = \langle C.inc(s_1,s_2), \{s_1'{=}s_2{+}3\}\rangle + \langle add_{15}(th,n,o,res,i,s_1'), \{r' = \_\}\rangle \\
add_{15}(\bar{x}_2) & = \langle 2, \{i'{=}s_1\}\rangle + \langle add_5(th,n,o,res,i'), \{r' = \_\}\rangle \\
A.inc(th,i) & = \langle A.inc_1(th,i), \{r' = \_\}\rangle \\
A.inc_1(th,i) & = \langle 4, \{r' = i{+}1\}\rangle
\end{array}$$

Fig. 4. The CRS of the example, where $\bar{x}_1$, $\bar{x}_2$ and $\bar{x}_3$ respectively are $\langle th, n, o, res, i, s_1, s_2\rangle$, $\langle th, n, o, res, i, s_1\rangle$, and $\langle th, n, o, res, i\rangle$.

The evaluation of CRSs is defined by means of the following rules (here, $AC = \langle\langle p(\bar{x}), \phi\rangle{\cdot}bc^{eq}, exp, \psi\rangle$):

$$\frac{p(\bar{x}) \leftarrow \varphi \mid b_1^{eq} + \cdots + b_n^{eq} \ll_{AC} P_{cr},\ \ \psi{\wedge}\varphi \not\models false}{\langle\langle p(\bar{x}), \phi\rangle{\cdot}bc^{eq}, exp, \psi\rangle \leadsto_{cr} \langle b_1^{eq} \cdots b_n^{eq}{\cdot}\langle 0, \phi\rangle{\cdot}bc^{eq}, exp, \psi{\wedge}\varphi\rangle}$$

$$\frac{\psi \wedge \varphi \not\models false}{\langle\langle e, \varphi\rangle{\cdot}bc^{eq}, exp, \psi\rangle \leadsto_{cr} \langle bc^{eq}, e{+}exp, \psi \wedge \varphi\rangle}$$

which perform three actions: (1) check the satisfiability of the constraints (and accumulate them); (2) if the instruction is not a call, then add its symbolic cost expression to the accumulated cost; and (3) evaluate the next calls in the rule. We delay the application of the effects of executing a call (i.e., ϕ) by adding the pair $\langle 0, \phi\rangle$, to be considered *afterward*. The following theorem states the *soundness* of the proposed cost analysis: given a derivation in an RBR program with cost a, there is a derivation in its CRS with the same cost a.

Theorem 4.17. *Let P be an RBR program, $C_0 \equiv \langle start, p(\bar{x}, y), lv_0\rangle$; $h, \varphi_0 \equiv \alpha(C_0)$, $Q \equiv \langle p(\bar{x}), SH, ACY\rangle$ a safe Sharing-Acyclicity description of an initial context, and P_{cr} the cost relation system w.r.t. $\mathcal{M}^s$ and Q. The following holds: if $C_0 \leadsto^n C_n$ is a trace t for P, then there exists a trace $\langle b^{eq}, 0, \varphi_0\rangle \leadsto_{cr}^n \langle \_, e, \varphi_n\rangle$ and a consistent assignment $\sigma : vars(\varphi_n) \mapsto \mathbb{Z}$ for φ_n such that $e\sigma{=}\mathcal{M}(t)$.*

As it can be observed from the example, cost relations depend on the cost of other calls (i.e., they are usually *recursive*). It is useful, for practical purposes, to obtain a *non-recursive* representation of the equations, known as *closed form*, which can be an exact solution of the equations, or an upper/lower bound. Using the PUBS solver [7,14], we automatically obtain the upper (resp., lower) bound $9{+}16{*}(n{+}1)$ (resp., $9{+}16{*}(\frac{n}{3}{-}1)$) for the CRS $add(th, n, o)$ of Fig. 4 (when $n \ge 0$). Intuitively, the solving process consists of the next steps: (1) We find safe bounds for the number of times that each relation can be applied by relying on ranking functions [7]. For the example, $n{+}1$ is the maximum number of iterations that the loop can make (when increasing i always by 1), and $\frac{n}{3}{-}1$ is the minimum one (when increasing i always by 3). When the solver finds upper bounds on the number of iterations of all relations, termination of the program is proven. (2) We find the worst-case cost of all applications of the relation. This step is non-trivial and requires finding invariants, which state the range of values that each variable can take, and then maximizing the cost expressions w.r.t. such invariants. In this example, the cost of all applications is the constant 16 and, thus, there is no need to find invariants and maximize. (3) If the relation has one recursive call, a closed-form bound is obtained by multiplying the upper bound on iterations by the worst-case cost of all iterations and then adding the cost associated to executing the base cases. This is how the above upper and lower bounds are found. If the relation has several recursive calls, an exponential cost expression will be produced. All details of this process can be found in [7].

5. The COSTA system: an implementation for Java bytecode

This section describes COSTA, an abstract-interpretation-based COSt and Termination Analyzer for Java bytecode. The system receives as input a bytecode program and a resource of interest in the form of a cost model, and tries to obtain an upper bound of the resource consumption. COSTA can deal, among others, with the above non-trivial cost notions, i.e., the heap consumption, the number of instructions. Additionally, COSTA tries to prove termination, which implies the boundedness of any resource consumption. The termination module [4] is outside the scope of this article. The system is implemented in Prolog; it uses the Parma Polyhedra Library [16] for manipulating linear constraints, and the PUBS system [7] for solving the CRSs. To the best of our knowledge, this system is the first one to apply cost analysis to realistic object-oriented programs, in bytecode form. Currently, it can be used through the web interface at `http://costa.ls.fi.upm.es`.

Table 1
Runtimes of analysis.

bench	B	M	C	R	R_o	E	rbr	opt	ana	size	crs	ub	sim	tot	tr
copy	108	4	3	78	56	55	21	50	66	362	12	82	0	594	8
divByTwo	15	1	1	17	15	16	2	10	7	54	2	10	0	87	5
binsearch	68	1	1	31	30	30	6	30	24	207	4	158	0	430	14
fact	14	1	1	11	10	9	4	6	6	2	0	8	0	28	3
arrayReverse	27	1	1	28	24	25	5	20	24	137	4	37	0	226	8
concat	44	1	1	49	43	45	10	40	74	348	6	111	0	589	12
add	105	4	5	32	27	27	14	23	18	78	2	94	0	228	7
merge	170	3	2	89	61	59	20	77	348	258	13	270	0	986	11
power	15	1	1	11	10	9	0	7	10	5	0	14	0	38	3
copy_cons	92	5	4	56	34	31	15	35	50	48	4	49	0	201	4
evenDigits	31	2	1	34	30	33	8	27	12	102	3	22	0	175	5
selectOrd	51	1	1	58	55	57	11	51	68	1556	9	253	0	1948	34
doSum	27	2	1	28	25	19	5	19	11	31	0	13	0	81	3
multiply	58	1	1	75	64	67	15	89	225	2411	16	859	0	3616	48
hanoi	20	1	1	13	11	9	4	8	30	10	0	150	0	202	16
fibonacci	18	1	1	14	13	11	5	9	11	14	0	16	0	55	4
copy_bst	123	6	4	119	73	67	30	73	138	513	16	630	0	1399	12
as_push	658	7	6	104	70	59	54	59	160	17	14	67	0	372	4
ns_pop	666	9	7	132	90	77	58	84	205	27	22	100	0	496	4
nq_dequeue	748	8	7	128	90	78	62	82	214	33	22	162	2	577	5
nl_prev	1024	10	9	212	140	118	104	150	586	47	53	281	0	1220	6

Table 1 aims at assessing the efficiency of our analysis. Two sets of benchmarks are considered whose complexity ranges from constant to exponential (their code is available at the COSTA web-site). The first set, from copy to copy_bst, consists of classical examples in complexity analysis; the second set is taken from the `net.datastructures` Java package [38], which contains a collection of Java interfaces and classes implementing important data structures and algorithms [37]. Such programs are relevant since they intensively use object-oriented features. This is made even more evident by the fact that analyzing some part of the Java libraries is often required. In that package, the following classes have been selected as starting point: ArrayStack, NodeStack, NodeQueue and NodeList. Due to lack of space, we only show the results for one method per class: resp., push, pop, dequeue, and prev. COSTA handles bytecode programs for Java SE 1.4, 1.5 and 1.6. Experiments have been done in Java 1.5.0_22.

Columns **B**, **M**, and **C** in the table show, resp., the number of instructions, methods, and classes. Column **R** shows the number of RBR rules; $\mathbf{R}_o$ shows the same number after some optimizations. **E** shows the number of equations in the final cost relation system. Columns **rbr** and **opt** show, resp., the time for building the RBR and for optimizing it. Experiments have been performed on an Intel Core 2 Quad Q9300 at 2.5 GHz with 1.95 GB of RAM, running Linux 2.6.28-11. Times are in milliseconds, and have been computed as the average of five runs. **ana** is the time needed by the auxiliary analyses required by size analysis, whose time appears in **size**. Column **crs** is the time to obtain the CRS, and **ub** and **sim** is the time for, resp., obtaining a closed-form solution and for simplifying it. The total time is shown in **tot**. Finally, **tr** evaluates how the analysis time varies w.r.t. the size of the program. For this, we divide the total analysis time by the number of rules in the RBR. This number can be roughly interpreted as the average time to analyze a rule, which ranges from 3 to 48 ms. We argue that, at least in our experiments, analysis time is acceptable. Importantly, the current implementation is not optimized for efficiency.

Table 2 shows the closed-form upper bounds obtained for the same examples. In all cases, the result for the number of instructions cost model $\mathcal{M}_i$ is shown. We also show upper bounds w.r.t. another model, $\mathcal{M}_o$, which counts the number of objects allocated in the heap. In Table 1 only the times for $\mathcal{M}_i$ were shown, because the differences are small. Calls to *native* methods appear as symbolic constants in the upper bounds. This is the case of fillInStackTrace in `java.lang.Throwable`, which is represented by the constant c(fST). We evaluate the precision by comparing the inferred upper bounds with the *actual* number of instructions and memory consumption. For this aim, we implemented a JVMTI agent (see http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/) which tracks object allocations and counts the number of bytecodes executed in concrete traces. Column **act** contains the exact number of bytecode instructions or objects required by concrete executions of the methods. Since COSTA approximates the worst-case behavior, we selected as input parameters those which lead to the worst execution cost of the programs. The column **est** shows the value obtained by evaluating the upper bound computed by COSTA on the given input data. Finally, **acc** indicates the accuracy **act**/**est**$*$100. Soundness requires **act**$\leq$**est**, 100 indicates that the upper bound is exact.

For $\mathcal{M}_i$, COSTA obtains an exact upper bound in four cases: fact, merge, power, and copy_cons. Except for fibonacci and nl_prev, the accuracy obtained for the remaining benchmarks ranges from 50% to 97%, which we argue is quite good for many applications. The main reason for the loss of precision in these benchmarks is that there are loops (or recursion) whose body contains computations with a different cost at each iteration. In this case, the CRS solver takes the larger cost, and multiplies it by the number of iterations. This is the case of binsearch, selectOrd, doSum, hanoi, copy_bst, as_push, ns_pop, nq_dequeue, and nl_prev. In other cases, the problem comes from exceptional behaviors as ArrayIndexOutOfBoundsException. This is the case of copy, arrayReverse, concat, add, and multiply, where COSTA computes the exact bound if exceptions are not considered. Finally, divByTwo has a division in the loop guard. This loss of precision

Table 2
Upper bounds ($n(X) = max(0, X)$).

	$\mathcal{M}_i$				$\mathcal{M}_o$			
bench	**est**	**act**	**acc**	**ub**	**est**	**act**	**acc**	**ub**
copy(a)	228	220	96	228	2	2	100	2
divByTwo(a)	40.58	38	94	$6+8log_2(1+n(2a-1))$				0
binSerch(a, b, c, d)	119.72	101	84	$16+24log_2(1+n(2d-2c+1))$				0 0
fact(a)	94	94	100	$4+9*n(a)$				0
arrayReverse(a)	162	152	94	$22+14a$	1	1	100	1
concat(a, b)	389	265	68.12	$39+(11*(a+b)+13b)$	1	1	100	1
add(a, b, c)	214	207	97	$16+18*n(1+b)$				0
merge(a, b)	597	597	100	$27+30*n(a+b-1)$	20	20	100	$1+n(a+b-1)$
power(a, b)	104	104	100	$4+10*n(b)$				0
copy(a)	247	247	100	$23+26*n(a-1)$	10	10	100	$1+n(a-1)$
evenDigits(a)	502.59	369	73	$9+n(a)*(16+8log_2(1+n(2a-3)))$				0
selectSort(a)	1882	942	50	$37+n(a)*(36+30*n(a-1))$				0
doSum(a)	1271	677	53	$6+n(1+a)*(16+9*n(1+a))$				0
multiply(a,b,c)	36 609	28 117	77	$27b^2(c+1)+59b(c+1)+37c+49$				0
hanoi(a, b, c, d)	20 463	19 940	97	$20*2^{n(a)}-17$				0
fibonacciMethod(a)	9203	1589	17	$18*2^{n(a-1)}-13$				0
copy(a)	25 549	24 527	96	$100*2^{n(a-1)}-51$	1022	1022	100	$4*2^{n(a-1)}-2$
push(a, b)	40	22	55	$40+c(fST)$	1	1	100	$1+c(fST)$
pop(a)	38	31	82	$38+c(fST)$	1	1	100	$1+c(fST)$
dequeue(a)	33	32	97	$33+c(fST)$	1	1	100	$1+c(fST)$
prev(a, b)	130	43	33	$130+3*c(fST)$	3	1	33	$3+3*c(fST)$

also affects evenDigits, which calls divByTwo. COSTA is not accurate in two cases: fibonacci and nl_prev. In the first, the precision loss is bigger since [7] approximates the length of execution paths (generated by recursive calls) by $a - 1$, while in practice there are many execution paths that are shorter than $a - 1$. In the second, the loss comes from exceptional branches enclosed in if statements whose condition depends on fields: those are lost in the abstraction, so that the cost of all exceptional branches is accumulated in the upper bound (although only one exception can be raised at runtime).

As for $\mathcal{M}_o$, the results are more precise. In this setting, the worst case occurs when exceptions are raised and corresponding exception objects are created. In all cases, except for nl_prev, we obtain an accuracy of 100%. In nl_prev, objects are created in exceptional branches, which, as mentioned above, generates a loss of precision. We argue that the computed upper bounds are useful since they are both reasonably accurate and simple.

6. Precision issues, limits, and extensions

This section discusses the possible sources of precision loss, and the limits of COSTA when handling full sequential Java bytecode. It also explains how our approach can be naturally extended to handle most of these problems.

Field-sensitive analysis. When the cost depends on a value which is stored in a *field*, as in "while (x.f>i) i++;", we cannot provide cost bounds. This is because the size abstraction of Section 4.2 does not provide information about field values. To overcome this limitation, techniques making numeric and reference fields observable at the abstract level can be incorporated in the size analysis; e.g., numerical abstract domains [47], or program transformations at the level of the RBR, as proposed [6,8] and already included in COSTA. The latter consists in a pre-analysis which first infers which fields can be treated as if they were local variables and then, those fields are actually transformed into local variables. This would allow performing field-sensitive cost analysis by relying on a fully field-insensitive analysis. Therefore, this approach does not require any conceptual change to the presented framework.

Arrays. A similar problem arises when the cost depends on *array* elements, since array accesses are abstracted to *true*. Dealing with such cases requires modifying the size analysis in order to incorporate information about array contents, which is known to be one of the most challenging problems in program analysis [40]; therefore, it is difficult to provide a general solution. However, solutions can be provided for typical programming patterns which naturally fit in our approach: e.g., programs where the number of iterations of a loop depends on a value stored in an array. (1) Loops like "while (x[i]>0) x[i]−−;" can be handled similarly to numeric fields, since x[i] can be seen as the *i*-th field of the array object x; indeed, the techniques for fields are applicable here if x and i can be proven not to change during the loop using

[6,8]. (2) Array searches: "for(int i=0; e! =x[i]; i++)" can be handled (Section 4.2) since the RBR contains a branch which (exceptionally) exits the loop when $i \geq x.length$; therefore, the number of iterations can be bounded by the length of the array.

Non-linear, floating-point, and bitwise arithmetic. Our language does not include such classes of instructions; however, considering full Java bytecode requires to provide suitable abstractions for them. A sound (yet very imprecise) abstraction is to lose all information about variables affected by such instructions. COSTA currently uses this abstraction; however, in the case of *non-linear integer arithmetic*, it tries to improve the precision by applying *constant propagation*. E.g., $z{:=}x * y$ becomes a linear constraint when x or y are constants. A more general solution for non-linear arithmetic would require sophisticated numerical abstract domains [39], which comes at a high price in performance. *Linear floating-point arithmetic* can be easily included using existing techniques [27] available in PPL [16] (already used by COSTA). As for *bitwise arithmetic*, in some cases, the behavior of some operations can be reasonably approximated with linear constraints. However, a general solution requires incorporating more complex methods which can reason at the level of bits [22].

Cyclic data structures and other properties of the heap. We cannot provide bounds for programs traversing *cyclic data structures*. This is mainly due to the difficulty in bounding the number of loop iterations. Consider the loop while(x.data != e) x = x.next; and assume that x points to a cyclic linked list. In order to bound the number of iterations, one needs to (1) verify that there is an element equal to *e* in x; (2) verify that the loop will eventually visit all the elements; and (3) bound the number of elements in the data structure. The difficulty lies in verifying (1) and (2), since they require under-approximations. Another source of imprecision is due to the over-approximation applied by the analyses which infer sharing, acyclicity, and constancy information (e.g., the analysis can infer that a variable might point to a cyclic data-structure while in practice it does not). One can develop more precise analyses for inferring such properties and overcome precision problems at the price of performance.

Scalability. It is known that very precise analysis (like the global size analysis which is used in order to precisely infer the cost) and scalability are frequently at odds. The application of our analysis to code of large size (e.g., when the Java libraries must be analyzed) should be done in a compositional way. This means that small fragments of code are analyzed (often in a context-insensitive way) and the results are stored in some form of assertion or method summary. The potential benefit is that such precomputed information can be reused when analyzing other fragments of code. The drawback is that, if the analysis does not take context information into account, the results are less precise. Modularity in static analysis has been studied in several contexts. Recent work in the context of our COSTA system [53] studies the modular extension of the termination component. It is subject of future work to study compositionality (and incrementality) of the whole cost analysis framework.

Non-cumulative resources. Standard recurrence relations (like those in Section 4.4) can be used only to estimate cumulative resources. There exist cost models, like the peak of the memory consumption in garbage-collected languages [13] or the peak of active-tasks in concurrent languages [11] that can increase and decrease along the execution. Approximating these models requires non-standard forms of recurrence relations. In these cases, Section 4.4 is not applicable, but the remaining parts of the analysis can be directly used.

7. Related work

Since the advent of *mobile code*, the analysis of Java bytecode has become an active research area, and a number of analysis tools are currently available. Although they do not perform cost analysis, especially relevant are the analyses developed on the Soot framework [61] and the generic analyzer Julia [58]. Soot is a framework for the development of analyses for Java bytecode which includes points-to analysis, purity analysis, and dynamic data structure analysis. Julia features a generic analysis engine in which sharing, cyclicity, class, non-nullness, information flow, escape, constancy, and static initialization analysis have been integrated. Julia is nowadays an industrial-strength termination analyzer for Java bytecode [60]. Although Julia concentrates on termination analysis while we also perform cost analysis, the work in Julia is closely related to ours. Both systems contain path-length analysis [59] as a key component. Also, following the idea originally proposed in [4], Julia produces constraint logic programs whose termination implies the termination of the initial bytecode. Another interesting proposal for an *intermediate representation* for program analysis and verification of object-oriented (bytecode) programs is BoogiePL [34], which has been used to represent .NET and Java bytecode programs.

Focusing on cost analysis, significant effort has been devoted to extend the first, general framework [63] to different programming paradigms. Most work on automatic cost analysis refers to the context of high-level declarative languages. In the imperative paradigm, a lot of work has been devoted to WCET (*worst-case execution time*) analysis (see e.g. [64]), which in many respects can be considered complementary to our work. In WCET, most of the effort has been devoted to obtaining precise platform-dependent cost models, i.e., to estimating the time taken by the different instructions in the current, rather complex computing architectures. In contrast, we produce reasonably accurate platform-independent results. It should be noted that, in some contexts (like in real-time systems), platform dependence is inevitable. WCET has been applied to industrial code [35]. There is also work which studies the relationship between syntactical constructions of programming languages and their computational complexity [44,17]. These analyses are developed on simple imperative languages which are far from the presented bytecode and, unlike our work, *complexity classes* instead of CRSs are inferred (CRSs are valid not only to infer the complexity class, but also to compute non-asymptotic upper bounds).

Recent work [46] applies *sub-interpretation* (first used in first-order functional programming to deal with complexity) to object-oriented programs without recursion to provide upper bounds on stack usage. Not being based on generating CRSs, the approach does not follow the original framework [63]. Also, it is restricted to polynomial bounds and to a particular resource (stack usage). More recent work develops cost analyses to estimate the memory consumption. In particular, a technique for Java-like languages is proposed [21], which computes symbolic polynomial approximations of the amount of memory required by a program, and a study of memory consumption (including both heap space and stack usage) is done [28] on low-level programs which are similar to our bytecode programs. Both analyses are less general than ours, in both the properties they estimate (only memory consumption) and the kind of upper bounds they generate (polynomial).

Resource usage certification [32,15,43,26,51] proposes the use of security properties involving cost requirements; i.e., it requires that the (untrusted) code adheres to specific bounds on resource consumption. Our work shows, for the first time, that it is possible to automatically generate resource usage guarantees, not restricted to polynomial bounds, for a realistic *mobile* language. Related work in the context of Java bytecode includes the MRG project [15], which can be considered complementary to ours. MRG focuses on building a *proof-carrying code* [49] architecture for ensuring that programs are free from runtime violations of resource bounds. Their cost model deals with heap consumption: applications to be deployed on devices with limited memory, such as *smartcards*, must be rejected if they require too much memory. Unlike ours, the framework is restricted to polynomial bounds and to the above cost model. Further related work [24] also focuses on one particular notion of cost (memory consumption) and aims at verifying that the program executes in bounded memory by making sure that it does not create new objects inside loops. However, this approach does not infer bounds on resource usage.

8. Conclusions

The presented framework is, to the best of our knowledge, the first automatic approach to the cost analysis of object-oriented bytecode, a theoretical model for low-level languages (such as Java bytecode) which, most likely, come from compiling higher-level languages. The analysis is based on the generation of *cost relation systems* w.r.t. a *cost model* which provide useful approximations of the computational cost. We believe that our work opens the door to applying *resource usage analysis* in the context of realistic programming languages like Java bytecode. The theoretical framework has already been the basis for (1) the inference of the number of *executed instructions* of well-known programs used in research on complexity analysis [10]; and (2) the computation of the *heap consumption* of object-oriented programs with an extensive use of the heap [12]. In the latter case, cost relations were refined in order to consider the heap space which can be safely deallocated by *garbage collection* upon exit from a method, as approximated by *escape analysis* [20].

Current work is basically being focused on extending both the theoretical foundations and the practical implementation in order to handle a larger class of programs, and obtain improvements both in terms of efficiency and accuracy. Future work includes supporting *assertions*: COSTA will be able to (1) *save* the result of analyzing a method, together with information about the context of the analysis, in order to reuse it; and (2) *load* such results when analyzing methods for which an assertion is available, provided the current context is compatible. Assertions can also be used to specify the behavior of native or unavailable code.

Acknowledgements

We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

Appendix. Supplementary data

Supplementary data associated with this article can be found, in the online version at doi:10.1016/j.tcs.2011.07.009.

References

[1] A. Adachi, T. Kasai, E. Moriya, A theoretical study of the time analysis of programs, in: Proc. of MFCS'79, in: LNCS, vol. 74, Springer, 1979, pp. 201–207.
[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[3] A.V. Aho, R. Sethi, J.D. Ullman, Compilers – Principles, Techniques and Tools, Addison-Wesley, 1986.
[4] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, D. Zanardini, Termination analysis of Java bytecode, in: Proc. of FMOODS'08, in: LNCS, vol. 5051, Springer, 2008, pp. 2–18.
[5] E. Albert, P. Arenas, S. Genaim, I. Herraiz, G. Puebla, Comparing cost functions in resource analysis, in: Proc. of FOPARA'09, in: LNCS, vol. 6234, Springer, 2010, pp. 1–17.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla, Field-sensitive value analysis by field-insensitive analysis, in: Proc. of FM'09, in: LNCS, vol. 5850, Springer, 2009, pp. 370–386.
[7] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-form upper bounds in static cost analysis, Journal of Automated Reasoning 46 (2) (2011) 161–203.
[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Ramírez, From object fields to local variables: a practical approach to field-sensitive analysis, in: Proc. of SAS'10, in: LNCS, vol. 6337, Springer, 2010, pp. 100–116.
[9] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of Java bytecode, in: Proc. of ESOP'07, in: LNCS, vol. 4421, Springer, 2007, pp. 157–172.
[10] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Experiments in cost analysis of Java bytecode, in: Proc. of BYTECODE'07, ENTCS 190 (2007) 67–83.
[11] E. Albert, P. Arenas, S. Genaim, D. Zanardini, Task-level analysis for a language with async-finish parallelism, in: Proc. of LCTES'11, ACM Press, 2011, pp. 21–30.
[12] E. Albert, S. Genaim, M. Gómez-Zamalloa, Heap space analysis for Java bytecode, in: Proc. of ISMM'07, ACM Press, 2007, pp. 105–116.
[13] E. Albert, S. Genaim, M. Gómez-Zamalloa, Parametric inference of memory requirements for garbage collected languages, in: Proc. of ISMM'10, ACM Press, 2010, pp. 121–130.
[14] E. Albert, S. Genaim, A.N. Masud, More precise yet widely applicable cost analysis, in: Proc. of VMCAI'11, in: LNCS, vol. 6538, Springer, 2011, pp. 38–53.
[15] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, I. Stark, Mobile resource guarantees for smart devices, in: Proc. of CASSIS'04, in: LNCS, vol. 3362, Springer, 2005, pp. 1–27.
[16] R. Bagnara, P.M. Hill, E. Zaffanella, The parma polyhedra library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems, Science of Computer Programming 72 (1–2) (2008).
[17] A.M. Ben-Amram, N.D. Jones, L. Kristiansen, Linear, polynomial or exponential? complexity inference in polynomial time, in: Proc. of CiE'08, in: LNCS, vol. 5028, Springer, 2008, pp. 67–76.
[18] F. Benoy, A. King, Inferring argument size relationships with CLP(R), in: Proc. of LOPSTR'97, in: LNCS, vol. 1207, Springer, 1997, pp. 204–223.
[19] R. Benzinger, Automated higher-order complexity analysis, Theoretical Computer Science 318 (1–2) (2004).
[20] B. Blanchet, Escape analysis for object oriented languages. Application to Java (TM), in: Proc. of OOPSLA'99, ACM Press, 1999, pp. 20–34.
[21] V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine, Parametric prediction of heap memory requirements, in: Proc. of ISMM'08, ACM Press, 2008, pp. 141–150.
[22] J. Brauer, A. King, Automatic abstraction for intervals using Boolean formulae, in: Proc. of SAS'10, in: LNCS, vol. 6337, Springer, 2010, pp. 167–183.
[23] M. Bruynooghe, M. Codish, J.P. Gallagher, S. Genaim, W. Vanhoof, Termination analysis of logic programs through combination of type-based norms, ACM Transactions on Programming Languages and Systems 29 (2) (2007).
[24] D. Cachera, T. Jensen, D. Pichardie, G. Schneider, Certified memory usage analysis, in: Proc. of FM'05, in: LNCS, vol. 3582, Springer, 2005, pp. 91–106.
[25] D. Cachera, T.P. Jensen, A Jobin, P. Long-run, Cost analysis by approximation of linear operators over dioids, Mathematical Structures in Computer Science 20 (4) (2010) 589–624.
[26] A. Chander, D. Espinosa, N. Islam, P. Lee, G. Necula, Enforcing resource bounds via static verification of dynamic checks, in: Proc. of ESOP'05, in: LNCS, vol. 3444, Springer, 2005, pp. 311–325.
[27] L. Chen, A. Miné, P. Cousot, A sound floating-point polyhedra abstract domain, in: Proc. of APLAS'08, in: LNCS, vol. 5356, Springer, 2008, pp. 3–18.
[28] W.-N. Chin, H.H. Nguyen, C. Popeea, S. Qin, Analysing memory resource bounds for low-level programs, in: Proc. of ISMM'08, ACM Press, 2008, pp. 151–160.
[29] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. of POPL'77, ACM Press, 1977, pp. 238–252.
[30] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proc. of POPL'78, ACM Press, 1978, pp. 84–97.
[31] S.J. Craig, M. Leuschel, Self-tuning resource aware specialisation for prolog, in: Proc. of PPDP'05, ACM Press, 2005, pp. 23–34.
[32] K. Crary, S. Weirich, Resource bound certification, in: Proc. of POPL'00, ACM Press, 2000, pp. 184–198.
[33] S.K. Debray, N.W. Lin, Cost analysis of logic programs, ACM Transactions on Programming Languages and Systems 15 (5) (1993).
[34] R. DeLine, K.R.M. Leino, BoogiePL: a typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
[35] A. Ermedahl, J. Gustafsson, B. Lisper, Experiences from industrial wcet analysis case studies, in: Proc. of WCET'05, volume 1 of OASICS, 2005.
[36] S. Genaim, F. Spoto, Constancy analysis, in: 10th Workshop on Formal Techniques for Java-like Programs, 2008.
[37] M. Goodrich, R. Tamassia, Data Structures and Algorithms in Java, 3rd ed., John Wiley, 2004.
[38] M.T. Goodrich, R. Tamassia, R. Zamore, The net.datastructures Package, version 3. Available at http://net3.datastructures.net, 2003.
[39] B. S. Gulavani, S. Gulwani, A numerical abstract domain based on expression abstraction and max operator with application in timing analysis, in: Proc. of CAV'08, in: LNCS, vol. 5123, Springer, 2008, pp. 370–384.
[40] N. Halbwachs, M. Péron, Discovering properties about arrays in simple programs, in: Proc. of PLDI'08, ACM Press, 2008, pp. 339–348.
[41] M. Hermenegildo, E. Albert, P. López-García, G. Puebla, Abstraction carrying code and resource-awareness, in: Proc. of PPDP'05, ACM Press, 2005, pp. 1–11.
[42] M. Hermenegildo, G. Puebla, F. Bueno, P. López-García, Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor), Science of Computer Programming 58 (1–2) (2005).
[43] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: Proc. of POPL'03, ACM Press, 2003, pp. 185–197.
[44] L. Kristiansen, N.D. Jones, The flow of data and the complexity of algorithms, in: Proc. of CiE'05, in: LNCS, vol. 3526, Springer, 2005, pp. 263–274.
[45] D. Le Metayer, ACE: an automatic complexity evaluator, ACM Transactions on Programming Languages and Systems 10 (2) (1988).
[46] J.-Y. Marion, R. Pèchoux, Resource control of object-oriented programs, in: Proc. of LICS affiliated Workshop LCC'07, 2007.
[47] A. Miné, Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics, in: Proc. of LCTES'06, ACM, 2006, pp. 54–63.
[48] J. Navas, E. Mera, P. López-García, M. Hermenegildo, User-definable resource bounds analysis for logic programs, in: Proc. of ICLP'07, in: LNCS, vol. 4670, Springer, 2007, pp. 348–363.
[49] G. Necula, Proof-carrying code, in: Proc. of POPL'97, ACM Press, 1997, pp. 106–119.
[50] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, 2nd ed., Springer, 2005.
[51] K.-H. Niggl, H. Wunderlich, Certifying polynomial time and linear/polynomial space for imperative programs, SIAM Journal on Computing 35 (5) (2006).
[52] G. Puebla, C. Ochoa, Poly-controlled partial evaluation, in: Proc. of PPDP'06, ACM Press, 2006, pp. 261–271.
[53] D. Ramírez, J. Correas, G. Puebla, Modular termination analysis of Java bytecode and its application to phoneme core libraries, in: Proc. of FACS'10, LNCS, vol. 6921, Springer, 2010 (in press).
[54] M. Rosendahl, Automatic complexity analysis, in: Proc. of FPCA'89, ACM Press, 1989, pp. 144–156.
[55] S. Rossignoli, F. Spoto, Detecting non-cyclicity by abstract compilation into Boolean functions, in: Proc. of VMCAI'06, in: LNCS, vol. 3855, Springer, 2006, pp. 95–110.
[56] D. Sands, A Naïve time analysis and its theory of cost equivalence, Journal of Logic and Computation 5 (4) (1995).
[57] S. Secci, F. Spoto, Pair-sharing analysis of object-oriented programs, in: Proc. of SAS'05, in: LNCS, vol. 3672, Springer, 2005, pp. 320–335.
[58] F. Spoto, JULIA: a generic static analyser for the java bytecode, in: Proc. of FTfJP'05, 2005.
[59] F. Spoto, P.M. Hill, E. Payet, Path-length analysis of object-oriented programs, in: Proc. of EAAI'06, 2006. Available at http://profs.sci.univr.it/~spoto/papers.html.
[60] F. Spoto, F. Mesnard, É Payet, A termination analyser for Java bytecode based on path-length, Transactions on Programming Languages and Systems 32 (3) (2010).

[61] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, P. Co, Soot — a Java optimization framework, in: Proc. of CASCON'99, IBM, 1999, pp. 125–135.
[62] P. Wadler, Strictness analysis aids time analysis, in: Proc. of POPL'88, ACM Press, 1988, pp. 119–132.
[63] B. Wegbreit, Mechanical program analysis, Communications of the ACM 18 (9) (1975).
[64] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem — overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems 7 (36) (2008).