Resource Usage Analysis and its Application to Resource Certification

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2],
Germán Puebla[2], and Damiano Zanardini[2]

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

Abstract. *Resource usage* is one of the most important characteristics of programs. Automatically generated information about resource usage can be used in multiple ways, both during program development and deployment. In this paper we discuss and present examples on how such information is obtained in COSTA, a state of the art static analysis system. COSTA obtains safe symbolic upper bounds on the resource usage of a large class of general-purpose programs written in a mainstream programming language such as Java (bytecode). We also discuss the application of resource-usage information for *code certification*, whereby code not guaranteed to run within certain user-specified bounds is rejected.

1 Introduction

One of the most important characteristics of a program is the amount of resources that its execution will require, i.e., its *resource usage*. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. Resource usage information has many applications, both during program development and deployment. Therefore, automated ways of estimating resource usage are quite useful and the general area of resource usage analysis (or resource analysis for short) has received considerable attention.

Statically estimating the resource usage of realistic programs is far from trivial. Thus, in the current practice, safe resource usage guarantees are only available for critical applications with strong resource usage constraints. These include real-time applications, which are required to execute within a certain maximum amount of time. Such applications are the subject of *Worst Case Execution Time Analysis* (WCET analysis for short), which is a quite active research area. See e.g. [23]. Unfortunately, WCET analysis for mainstream hardware and software is extremely complicated. On the hardware side, modern computer architectures have multiple memory levels and internal pipelining which make it rather difficult to predict the execution time of machine instructions. On the software side, accurately estimating the number of times each program loop and recursion will execute is a rather complex problem. In order to ease the situation, on the hardware side, real-time applications often run on embedded

systems whose timing behaviour is much more predictable. On the software side, real-time applications are programmed in restricted versions of general languages such as Real-Time Java or are designed and implemented using special languages such as Hume [27] and Timber [1], which are rooted in functional programming and have tool support for performing WCET analysis. Similarly, when strong memory usage limitations are in place, the programming constructs allowed are often very restricted, disallowing recursion or, as in the case of JavaCard, even strongly discouraging the use of dynamic memory allocation after the initialization phase of the applet.

In this paper we discuss the main techniques used in COSTA [5], a static analysis system which allows obtaining safe symbolic upper bounds on the resource usage of Java bytecode (JBC for short). COSTA follows the classical approach to static resource analysis proposed in Wegbreit's seminal work [57] and which consists of two phases. First, given a program and a cost model, the analysis produces *cost relations* (CRs for short). Second, the systems tries to obtain *closed-form upper-bounds* for them. The results are *symbolic* in the sense that they do not refer to concrete, platform dependent, resources such as execution time, but rather they provide platform-independent information. This has the advantage that the results are applicable to any implementation of the Java Virtual Machine (JVM) on any particular hardware. It also has the disadvantage that the information cannot refer to platform specific resources such as run-time. The fact that the analysis handles JBC represents that, at least in principle, it can deal with general-purpose programs written in a mainstream programming language such as Java and potentially other languages compiled to JBC. The upper bounds computed by COSTA can then be compared against user-provided resource usage specifications. This allows automatically rejecting code not guaranteed to execute within the specified resources.

Note that, unlike COSTA, previous resource analyses based on Wegbreit's approach have been formulated on, less widely used, declarative programming languages [57,35,44,22]. There are very few approaches for imperative programming languages [25] and, unlike COSTA, they are formulated at the *source code* level and they do not follow Wegbreit's approach. However, analyzing compile code has wider applicability, since it is quite often the case with Java applications that the *code consumer* has access to the bytecode, often bundled in *jar* files, but no access to the source code, as usual for commercial software and in mobile code. In the context of *mobile code*, programming languages which are compiled to *bytecode* and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET. Mobile code was the motivation for the concept of *Proof-Carrying Code* [41]: in order for the mobile code to be verifiable by the user, security properties (including resource usage limitations) must refer to the code available to the user, i.e., the bytecode, so that it is possible to check the provided proof and verify that the program satisfies the requirements (e.g., that the code does not require more than a certain amount of memory, or that it executes in less than a certain amount of time).

Among all possible applications of resource analysis, in this work we describe its application to *resource certification*, whereby programs are coupled with information about their resource usage. This information allows deciding whether the resources used by the program execution are acceptable or not *before* running the program. Note that resource usage can be considered a security property of untrusted mobile code, possibly in the context of proof-carrying code. Programs whose resource usage is not certified are potentially harmful, since their execution may require more resources than we are willing to spend or they may even have monetary cost by executing *billable events* such as sending text messages or making http connections on a mobile phone. In fact, mobile devices is one of the settings where resource certification is more important, because of the limited computing power typically available on mobile devices.

The rest of the paper is structured as follows. In Section 2 object-oriented bytecode, in the style of Java bytecode, is briefly described. This is required to understand the different examples in the paper, which show how resource analysis of a bytecode program is performed. Then, in Section 3, we describe, by means of examples, how to obtain CRs from a program and a cost model, whereas in Section 4 we illustrate how to obtain closed-form upper-bounds for CRs. Section 5 describes the application of resource analysis to resource certification. In Section 6 we present an overview on the existing large body of work on resource analysis. Finally, the conclusions and some venues for future work are discussed in Section 7.

2 The Context: Object-Oriented Bytecode

In order to simplify the formalization of our analysis, a simple object-oriented bytecode language is considered, which roughly corresponds to a representative subset of sequential Java bytecode. We refer to it as *simple bytecode*. For short, unless we explicitly mention *Java* bytecode, all references to bytecode in the rest of the paper correspond to our simple bytecode. Simple bytecode is able to handle integers and object creation and manipulation. For simplicity, simple bytecode does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Anyway, such features can be easily handled in this framework, as done in the implementation of the COSTA system.

A bytecode program consists of a set of *classes* C, partially ordered with respect to the *subclass* relation. Each class $c \in C$ contains information about the class it extends, and the fields and methods it declares. Subclasses inherit all the fields and methods of the class they extend. Each method comes with a *signature* m which consists of the class where it is defined, its name and its type. For simplicity, all methods are supposed to return a value. There cannot be two methods with the same signature. The bytecode associated to a method m is a sequence $\langle b_1, \ldots, b_n \rangle$ where each b_i is a bytecode instruction. Local variables of a k-ary method are denoted by $\langle l_0, \ldots, l_n \rangle$ with $n \geq k-1$. In contrast to Java

```
public static int binarySearch(int[ ] t, int v, int l, int u) {
   int m;
   while (l <= u) {
      m = (l + u) / 2;
      if (t[m] == v) return m;
      if (t[m] > v) u = m - 1;
      else l = m + 1;
   }
   return -1;
}
```

```
0  : load 2
1  : load 3
2  : ifgt 31
3  : load 2
4  : load 3
5  : add
6  : push 2
7  : div
8  : store 4
9  : load 0
10 : load 4
11 : aload
12 : load 1
13 : ifneq 16
14 : load 4
15 : return
16 : load 0
17 : load 4
18 : aload
19 : load 1
20 : ifleq 26
21 : load 4
22 : push 1
23 : isub
24 : store 3
25 : goto 0
26 : load 4
27 : push 1
28 : add
29 : store 2
30 : goto 0
31 : push -1
32 : return
```

Fig. 1. A Java source (left) with its corresponding bytecode (right)

source, in bytecode the *this* reference of instance (i.e., non-static) methods is passed explicitly as the first argument of the method, i.e., l_0 and $\langle l_1, \ldots, l_k \rangle$ correspond to the k formal parameters, and the remaining $\langle l_{k+1}, \ldots, l_n \rangle$ are the local variables declared in the method. For static k-ary methods $\langle l_0, \ldots, l_{k-1} \rangle$ are used for the formal parameters and $\langle l_k, \ldots, l_n \rangle$ for the local variables declared in the method. Similarly, each field f has a unique signature which consists of the class where it is declared, its name and its type . A class cannot declare two fields with the same name. The following instructions are included:

$$
\begin{aligned}
\mathit{bcInstr} ::= \;& \mathsf{load}\ i \mid \mathsf{store}\ i \mid \mathsf{push}\ n \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{div} \\
& \mid \mathsf{iflt}\ j \mid \mathsf{ifgt}\ j \mid \mathsf{ifleq}\ j \mid \mathsf{ifeq}\ j \mid \mathsf{ifneq}\ j \mid \mathsf{ifnull}\ j \mid \mathsf{goto}\ j \\
& \mid \mathsf{new}\ c \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f \\
& \mid \mathsf{newarray}\ d \mid \mathsf{aload} \mid \mathsf{astore} \mid \mathsf{arraylength} \\
& \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokenonvirtual}\ m \mid \mathsf{return}
\end{aligned}
$$

Similarly to Java bytecode, simple bytecode is a stack-based language. The instructions in the first row manipulate the *operand stack*. The second row contains *jump* instructions. Instructions in the third row manipulate *objects* and their fields, while the fourth row works on *arrays*. The last row contains instructions dealing with *method invocation*. As regards notation, i is an integer which corresponds to a local variable index, n is an integer or null, j is an integer which corresponds to an index in the bytecode sequence, $c \in C$, m is a method signature, and f is a field signature.

We assume an operational semantics which is a subset of the JVM specification [37]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects and arrays allocated during the execution of the program. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same objects in the heap.

Example 1. (running example) Figure 1 depicts the bytecode and a possible Java source (only shown for clarity of the presentation) of our running example. The Java program implements the binary search of an element v in a sorted array t. Variables l and u represent two indexes in the array t. If there exists a position $l \leq m \leq u$ such that t[m] is equal to v, then m is returned as result. Otherwise the method returns -1. The table of local variables is indexed from 0 to 4 and it contains the values of t, v, l, u and m respectively.

The first three instructions check the negation of the loop condition. If the check succeeds, i.e., the loop condition does not hold, then the body of the loop is not executed and the control goes to instruction 31, where the constant -1 is returned as the result (instruction 32) of the method. Otherwise the value of $(l+u)/2$ is stored in m in instructions 3–8. Then, instructions 9–13 check whether t[m]! =v holds. It the check fails, meaning that t[m]==v, then instruction 14 is executed, where variable m is pushed on the stack and returned as result in 15. Otherwise, the execution jumps to line 16, where the second `if` is checked (instructions $16, \ldots, 20$). If t[m]>v then instructions $21, \ldots, 24$ store the value of $m-1$ in u and at instruction 25 the control goes back to the beginning of the loop, i.e., instruction 0. Otherwise, the value of $m+1$ (instructions $26, \ldots 30$) is assigned to l and, similarly, control returns to instruction 0. □

3 Cost Analysis: from Bytecode to Cost Relations

This section describes how a bytecode program is analyzed in order to produce a *cost relation system* (CRS) which describes its resource consumption. The analysis consists of a number of steps: (1) the *control flow graph* of the program is computed, and afterwards (2) the program is transformed into a *rule-based representation* which facilitates the subsequent steps of the analysis without losing information about the resource consumption; (3) size analysis and abstract compilation are used to generate size relations which describe how the size of data changes during program execution; (4) the chosen cost model is applied to each instruction in order to obtain an expression which represents its cost; (5) finally, a cost relation system is obtained by joining the information gathered in the previous steps.

3.1 Control Flow Graph

The *control flow graph* of a program allows statically considering all possible paths which might be taken at runtime, which is essential information for studying its cost. Unlike structured languages such as Java, bytecode features *unstructured* control flow, due to conditional and unconditional *jumps*. Reasoning about unstructured programs is more complicated for both human-made and automatic analysis. Moreover, the control flow is made even more difficult to deal with by *virtual method invocation* and the need to handle *exceptions*. Each node of a control flow graph contains a (maximal) sequence of *non-branching* instructions, which are guaranteed to be executed sequentially. This amounts

Fig. 2. The control flow graph of the running example

to saying that execution always starts at the beginning of the sequence, and the instructions in the sequence are executed one after the other until the end, without executing any other piece of code in the meanwhile.

The CFG can be built using standard techniques [2], suitably extended in order to deal with virtual method invocation. For this, it is essential to perform *class analysis* (see e.g. [51] and its references) which allows statically obtaining a safe approximation of the set of classes to which an object variable may belong to at runtime. Consider, for example, an object o, and suppose class analysis determines that a set C contains all classes to which o may belong at a given program point which contains a call of the form $o.m()$. Then, such call is translated into a set of calls $o.m_c$, one for every class $c \in C$ where m is defined. This is obtained by adding new *dispatch blocks*, containing calls $\mathsf{invoke}(c.m)$ to each implementation of m. Access to such blocks is guarded by mutually exclusive conditions on the runtime class of o.

Figure 2 shows the CFG of the running example. The graph contains 7 nodes, each composed of non-branching instructions which are always executed sequentially. All nodes end either in a return instruction (as in $binSearch_{31}$ and $binSearch_{14}$), or in an instruction labeled with $\mathsf{nop}()$, which indicates a conditional or unconditional jump in the original bytecode. Edges corresponding to conditional jumps are marked with a guard (which appears in brackets in the

figure; for clarity, conditions in the original Java program are also shown without brackets) representing the condition under which the edge can be traversed. Guards which are *true* are omitted. Instructions wrapped in a nop() have been replaced by edges in the CFG, but their place in the code is kept in order to take into account their costs when generating the corresponding cost relation system, as will be explained in Section 4 below.

3.2 Rule-based Representation

The *rule-based representation* (RBR) of a program is rich enough to preserve the information about cost, while being simple enough to develop a precise cost analysis, since some advanced features are compiled away, and control flow has been simplified. It is a *structured* procedural language with some relevant features:

1. *recursion* becomes the only iterative construct;
2. *guarded rules* are the only form of conditional construct;
3. there is only one kind of variables: *local variables*; and there is no operand stack (instead, the k-th position in the stack becomes an additional local variable s_k, exploiting the fact that, in Java bytecode, the height of the stack at each program point can be statically determined);
4. some object-oriented features are no longer present:
   - objects can be basically regarded as records including an additional field which contains their type;
   - the behaviour due to dynamic dispatch is compiled into *dispatch blocks*;
   - the language deals with the rest of object-oriented features by supporting object creation, field manipulation and arrays;
5. methods are represented as collections of related blocks, and executing a method is equivalent to executing the entry block of its representation.

These design choices help to make the generation of cost relation systems feasible, and consistent with the program structure. The rule-based representation of a program consists of a set of (global) procedures, one for each node in the CFG. Each procedure consists of one or more rules. A rule for a procedure p with k input arguments $\bar{x}$ and a (single) output argument y takes the form $p(\bar{x}, y) \leftarrow g, body$ where $p(\bar{x}, y)$ is the *head*, g is a *guard* expressing a boolean condition, and *body* is (a suitable representation of) the instructions which are contained in the node.

A guard can be either *true*, any linear condition about the value of variables (e.g., $x + y > 10$), or a type check $\mathsf{type}(x, c)$. Every (non-branching) instruction in the body is represented in a more readable (and closer to source code) syntax than the original bytecode (Figure 3). E.g., the instruction load i which loads the i-th local variable l_i into a new topmost stack variable s_{t+1} is written as $s_{t+1} := l_i$ (remember that variables named s_k originate from the k-th position in the stack but they are actually local variables in the RBR). Moreover, add is translated to $s_{t-1} := s_{t-1} + s_t$, where t is the current height of the stack in the original program, and putfield f is turned into $s_t.f := s_t$. As in the control

b_j	comp(b_j)
load i	$s_{t+1} := l_i$
store i	$l_i := s_t$
push n	$s_{t+1} := n$
pop	nop(pop)
dup	$s_{t+1} := s_t$
add	$s_{t-1} := s_{t-1} + s_t$
sub	$s_{t-1} := s_{t-1} - s_t$
lt	$s_{t-1} < s_t$
gt	$s_{t-1} > s_t$
eq	$s_{t-1} = s_t$
null	$s_t = \mathsf{null}$
¬ lt	$s_{t-1} \geq s_t$
¬ gt	$s_{t-1} \leq s_t$

b_j	comp(b_j)
¬ eq	$s_{t-1} \neq s_t$
¬ null	$s_t \neq \mathsf{null}$
type(n, c)	type(s_{t-n}, c)
new c	$s_{t+1} := \mathsf{new}\ c$
getfield f.	$s_t := s_t.f$
putfield f.	$s_{t-1}.f := s_t$
newarray$c$	$s_t := \mathsf{newarray}(c, s_t)$
aload	$s_{t-1} := s_{t-1}[s_t]$
astore	$s_{t-2}[s_{t-1}] := s_t$
arraylength	$s_t := \mathsf{arraylength}(s_t)$
invoke m	$m(s_{t-n}, \ldots, s_t, s_{t-n})$
return	$out := s_t$
nop(b)	nop(b)

Fig. 3. Compiling bytecode instructions (as they appear in the CFG) to rule-based instructions (t stands for the height of the stack before the instruction).

flow graph, branching instructions such as jumps and calls (which have become edges in the CFG, but may still be relevant to the resource consumption) are wrapped into a nop(_) construct, meaning that they are not executed in the RBR, but will be taken into account in the following steps of the analysis. RBR programs are restricted to *strict determinism*, i.e., the guards for all rules for the same procedure are pairwise mutually exclusive, and the disjunction of all such guards is always true.

A CFG can be translated into a rule-based program by building a rule for every node of the graph, which executes its sequential bytecode, and calls the rules corresponding to its successors in the CFG. Figure 4 shows the rule-based program for the CFG of Figure 2. The RBR for the *binSearch* method has an *entry procedure* which simply initializes all local variables and calls $binSearch_0$. In turn, the body of $binSearch_0$ loads l and u (corresponding to l and u), and calls its *continuation* $binSearch_0^c$, that decides which block will be executed next, depending on the comparison between s_1 and s_2 (note that the guards of the continuation rules are mutually exclusive). Procedures which are not continuations are named after the corresponding nodes in the CFG. Note that rules $binSearch_{26}$ and $binSearch_{21}$ contain a call to $binSearch_0$, which is in fact the loop condition.

An *operational semantics* can be given for the rule-based representation, which mimics the bytecode one. In particular, executing an RBR program still needs a *heap* and a *stack* of activation records. The main difference between the two semantics lies in the granularity of procedures: every method in the bytecode program has been partitioned into a set of procedures. In spite of this, it can be proven that any rule-based program is *cost-equivalent* to the bytecode program it comes from. Intuitively, cost-equivalence means that no information about the resource consumption is lost. The main cost-equivalence result states that the

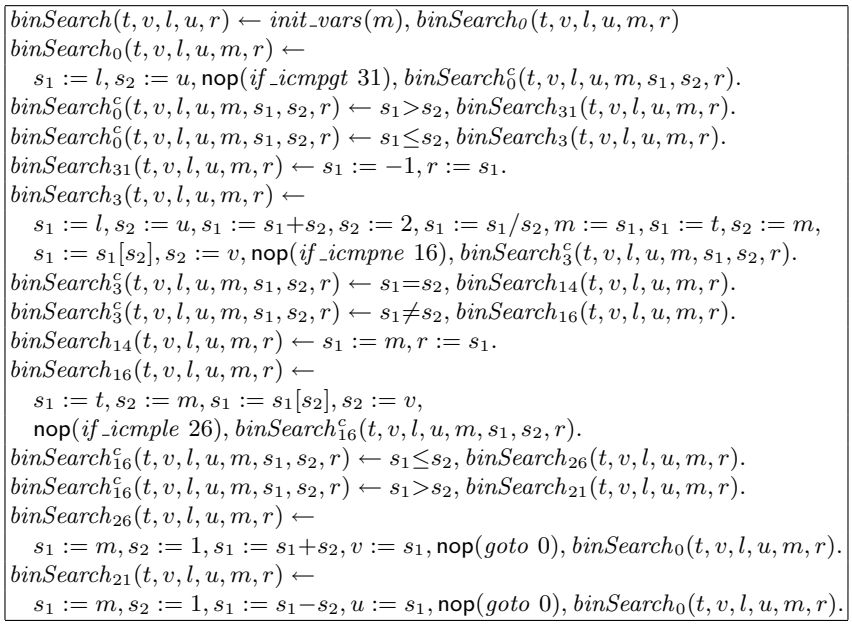

Fig. 4. RBR of the example (guards which are *true* are omitted).

execution from cost-equivalent input configurations for a bytecode program and its RBR leads to (1) non-termination in both cases; or (2) cost-equivalent output configurations.

3.3 Cost Relations

Given a program P (without loss of generality, it is supposed here that P has already been translated into its RBR form) and a cost model $\mathcal{M}$, the classical approach to cost analysis [57] consists in generating a set of *recurrence relations* (RRs) which capture the cost (w.r.t. $\mathcal{M}$) of running P on some input. As usual in this area, data structures are replaced by their *sizes* in the recurrence relations. From rule-based programs it is possible to obtain *cost relations* (CRs), an extended form of recurrence relations, which approximate the cost of running the corresponding programs. In the presented approach, each rule in the RBR program results in an equation in the CRS. Figure 5 shows the cost relation system (i.e., a system of cost relations) for the running example, where it is easy to see in the rule names the correspondence with the rule-based representation. In these equations, variables are in fact constraint variables which correspond to the sizes of those of the RBR. The right-hand side of an equation consists of an expression e which gives the cost of executing the body of the rule, and, for simplicity of the subsequent presentation, a linear constraint φ which denotes the

$$
\begin{array}{lll}
(1)\ \mathit{binSearch}(t,v,l,u) & = \mathit{binSearch}_0(t,v,l,u,0) & \\
(2)\ \mathit{binSearch}_0(t,v,l,u,m) & = 3 + \mathit{binSearch}^c_0(t,v,l,u,m,s_1,s_2) & \{s_1{=}l, s_2{=}u\} \\
(3)\ \mathit{binSearch}^c_0(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{31}(t,v,l,u,m) & \{s_1{>}s_2\} \\
(4)\ \mathit{binSearch}^c_0(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_3(t,v,l,u,m) & \{s_1{\leq}s_2\} \\
(5)\ \mathit{binSearch}_{31}(t,v,l,u,m) & = 2 & \\
(6)\ \mathit{binSearch}_3(t,v,l,u,m) & = 11 + \mathit{binSearch}^c_3(t,v,l,u,m',s_1,s_2) & \\
 & \quad \left\{ s_2{=}v, m' \in \left[\frac{l+u}{2} - \frac{1}{2}, \frac{l+u}{2} \right] \right\} & \\
(7)\ \mathit{binSearch}^c_3(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{16}(t,v,l,u,m) & \\
(8)\ \mathit{binSearch}^c_3(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{14}(t,v,l,u,m) & \\
(9)\ \mathit{binSearch}_{16}(t,v,l,u,m) & = 5 + \mathit{binSearch}^c_{16}(t,v,l,u,m,s_1,s_2) & \{s_2{=}v\} \\
(10)\ \mathit{binSearch}_{14}(t,v,l,u,m) & = 2 & \\
(11)\ \mathit{binSearch}^c_{16}(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{26}(t,v,l,u,m) & \\
(12)\ \mathit{binSearch}^c_{16}(t,v,l,u,m,s_1,s_2) & = \mathit{binSearch}_{21}(t,v,l,u,m) & \\
(13)\ \mathit{binSearch}_{26}(t,v,l,u,m) & = 5 + \mathit{binSearch}_0(t,v,l',u,m) & \{l'{=}m{+}1\} \\
(14)\ \mathit{binSearch}_{21}(t,v,l,u,m) & = 5 + \mathit{binSearch}_0(t,v,l,u',m) & \{u'{=}m{-}1\}
\end{array}
$$

Fig. 5. CRS of the running example

effect of the body on the variables. An important point to note is that, there are some cases where the simplification above may be incorrect. We opt by keeping this simplification in the presentation, though not in the implementation, because problems are rare and otherwise the presentation gets more complicated. In more detail, input-output size relations cannot always be merged together in φ. Constraints which originate from input-output relations of procedures called in the body of the rule cannot be taken into account until after the corresponding calls. This is because, by merging them, we can no longer distinguish finite failures from infinite failures. For instance, this happens when we have a procedure, say p, which never terminates. The input-output relation for p is represented with the constraint *false*, indicating that there are no successful executions for p. Any equation which has a call to p will have $\varphi = false$. If, by mistake, we take this *false* as a finite failure, we would incorrectly discard (part of) this equation as unreachable, when in reality execution never returns from this equation. In our running example, this phenomenon does not happen since even after adding constraints originating from input-output relations, no φ becames *false*.

Finally, note also that the output variable of the rule does not appear in the equation, as explained below. The generation of a cost equation for a given RBR rule goes through the following steps.

Size Measures. A *size measure* is chosen to represent and manipulate information relevant to cost, and a variable is abstracted to its *size* w.r.t. such measure. For example, (1) an array may be abstracted to its length, since this can typically give information about the cost of traversing it in a loop; or (2) an object can be abstracted to the longest path reachable from it (in this case, the size measure is well-known and is called *path-length* [52]) in order to describe the

cost of traversing data structures such as trees or linked lists. The choice of a size measure, in particular for heap structures, heavily depends on the program to be analyzed, and is intended to represent the maximum amount of relevant information. E.g., in cost and termination analysis, the measure used to abstract a piece of data or a data structure should give information about the behavior of a loop whose exit condition depends, as in the examples above, on the data.

Abstract Compilation. In the presented setting, one important issue is to capture relations between the size of a variable at different program points. For example, in analyzing $x := x + 1$, the interest usually lies in the relation "the value of x *after* is equal to 1 plus the value of x *before*".

In this steps of the cost analysis, instructions are replaced by *linear constraints* which approximate the relation between states (and, typically, between different program points) w.r.t. the chosen size measure. For instance, $s_1 := o$ is replaced by $s_1{=}o$, meaning that, after the assignment, the size of s_1 at the current program point is equal to the size of o. As another example, $x :=$ new c can be replaced, using the path-length measure, by $x{=}1$, meaning that the maximal path reachable from x after the object creation has length 1.

Importantly, the use of path-length as a size measure for reference requires extra information in order to obtain precise and sound results in the abstract compilation of instructions involving references:

(a) *sharing* information [48] is required in order to know whether two references may point to a common region of the heap; and
(b) *non-cyclicity* information [46] is required to guarantee that, at some specific program point, a reference points to a non-cyclic data structure, i.e., that the length of its longest path (therefore, the number of iteration on a typical traversing loop) is guaranteed to be finite.

A slightly more complicated example where non-cyclicity information is used is represented by a field access $x := y.f$: in this case

- no linear constraint can be inferred if f is a non-reference field;
- if y is detected as non-cyclic, then the size of x after the assignment can be guaranteed to be *strictly less* than the size of y before (since the data structure pointed by x is now a sub-structure of the one pointed by y);
- if y may be cyclic, then the size of x can only be taken to be *not greater* than the size of y (thus basically forbidding to find useful results on x and y in the following steps, as explained in Section 4).

The result of this *abstract compilation* is an *abstract program* which can be used to approximate the values of variables w.r.t. the given size measure.

Input-Output Size Relations. As mathematical relations, CRs cannot have output variables: instead, they should receive a set of input parameters and return a number which represents the cost of the associated computation. This

step of the analysis is meant to transform the abstract program in order to remove output variables from it. 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 procedure. Concretely, input-output size relations of the form $p(\bar{x}, y) \rightarrow \varphi$ are inferred, where φ is a constraint describing the relation between the sizes of the input $\bar{x}$ and the output y upon exit from p. This information is needed since the output of one call may be input to another call. Interestingly, input-output relations can be seen also as a denotational semantics for the abstract programs previously obtained. Sound input-output size relations can be obtained by taking abstract rules generated by abstract compilation, and combine them via a fixpoint computation [13], using abstract interpretation techniques [20] in order to avoid infinite computations.

Example 2. Consider the following RBR rules

$$\begin{aligned} incr(this, i, out) &\leftarrow incr_1(this, i, out) \\ incr_1(this, i, out) &\leftarrow s_1 := i,\ s_2 := 2,\ s_1 := s_1 + s_2,\ out := s_1 \end{aligned}$$

which basically come from the method

```
int incr(int i) { return i+2; }
```

All variables relevant to the computation are integers, so that abstract compilation abstracts every variable into itself (due to the choice of the size measure for numeric variables), and the abstract program looks like (constraints $\{s_1 = 0, s_2 = 0, out = 0\}$ describe the initial values of variables)

$$\begin{aligned} incr(this, i, out) &\leftarrow incr_1(this, i, out) \\ incr_1(this, i, out) &\leftarrow \{s_1 = 0, s_2 = 0, out = 0\} \mid \\ &\qquad s_1' = i,\ s_2' = 2,\ s_1'' = s_1' + s_2',\ out' = s_1'' \end{aligned}$$

By combining the constraints through the bodies, it can be inferred that the output value of *out* is 2 plus the input value of i, which, in the end, is represented by the input-output size relation

$$incr(this, i, out) \leftarrow \{out = i + 2\}$$

□

Cost Models. Resource usage analysis is a clear example of a program analysis where the focus is not only on the input-output behavior (i.e., *what* a program computes), but also on the history of the computation (i.e., *how* the computation is performed). Since the history of a computation can be normally extracted by its trace, it is natural to describe resource usage in terms of execution traces.

The notion of a *cost model* for bytecode programs formally describes how the resource consumption of a program can be calculated, given a resource of interest. It basically defines how to measure the resource consumption, i.e., the cost, associated to each execution step and, by extension, to an entire trace.

In the present setting, a cost model can be viewed as a function from a bytecode instruction, and dynamic information (local variables, stack, and heap) to a real number. Such number is the amount of resources which is consumed when executing the current step in the given configuration. Example 3 below introduces some interesting cost models which will be used in the next sections.

Example 3. The *instructions* cost model, denoted $\mathcal{M}_{inst}$, counts the number of bytecode instructions executed by giving a constant cost 1 to the execution of any instruction in any configuration: it basically measures the length of a trace.

The *heap* cost model, $\mathcal{M}_{heap}$, is used for estimating the amount of memory allocated by the program for dynamically storing objects and arrays (i.e., its heap consumption): it assigns to any instruction the amount of memory which it allocates in the current configuration. For instance, newarray *int* (resp., newarray c) allocates $v * size(int)$ (resp., $v * size(ref)$) bytes in the heap, where v denotes the length of the array (currently stored on the top of the stack), and $size(int)$ (resp., $size(ref)$) is the size of an integer (resp., a reference) as a memory area. □

Generation of Cost Relation Systems. Cost relation in a CRS are generated by using the abstract rules to build the constraints, and the original rule together with the selected cost model to generate *cost expressions* representing the cost of the bytecodes w.r.t. the model. Consider the cost relations identified by equations (6), (7) and (8) in the CRS of the running example (Figure 5), reproduced here for more clarity.

$$(6)\quad binSearch_3(t,v,l,u,m) = 11 + binSearch_3^c(t,v,l,u,m',s_1,s_2) \left\{ s_2{=}v, m' \in \left[\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}\right] \right\}$$

$$(7)\quad binSearch_3^c(t,v,l,u,m,s_1,s_2) = binSearch_{16}(t,v,l,u,m)$$

$$(8)\quad binSearch_3^c(t,v,l,u,m,s_1,s_2) = binSearch_{14}(t,v,l,u,m)$$

This excerpt shows that the inferred cost of executing $binSearch_3$ amounts to 11 plus the cost of executing $binSearch_3^c$. In turn, the cost of $binSearch_3^c$ can be either the cost of $binSearch_{16}$ or the cost of $binSearch_{14}$. In this case, cost expressions (as 11 in (6), or 0, left implicit in (7) and (8)) are simply constant values, which correspond to the number of executed instructions, since the cost model $\mathcal{M}_{inst}$ has been chosen. That is, eleven instruction are executed in $binSearch_3$ before calling $binSearch_3^c$, while no instructions are executed before calling $binSearch_{16}$ or $binSearch_{14}$ from $binSearch_3^c$. The constraints which appear at the end of some equations (as in (6); see also the complete CR for more examples) will be used in the following section. CRs extend recurrence relations in the sense that they allow to handle advanced features such as *non-determinism* (see for instance equations (7) and (8)) *constraints*, and *multiple arguments*, which arise in the cost analysis of realistic programs.

4 From Cost Relations to Closed-Form Upper Bounds

Though cost relations (CRs) are simpler than the programs they originate from, since all variables have integer type, in several respects they are not *as static as* one would expect from the result of a static analysis. First, cost relations are recursive, so that one may need to iterate for computing their value on a concrete input. Second, even for deterministic programs, it is well known that the loss of precision introduced by the size abstraction may result in cost relations which are non-deterministic. This happens in the above example in the loop inside binarySearch: since the array t is abstracted to its length, the contents of the array are lost in the abstraction. In particular, the value of t[m] is unknown in the CR. Hence, the pairs of Equations 7–8 and 11–12 end up having the same guards and the evaluation of this CR turns out to be non-deterministic. In order to find the worst-case cost, one would need to compute and compare many results. In some cases, the number of results may even be infinite. For both reasons, it is clear that it is interesting to compute *closed-form* upper bounds for the cost relation, whenever this is possible, i.e., upper bounds which are in non-recursive form. For example, the goal is to infer that the cost of calling $binSearch(t, v, l, u)$ is $24 * \lceil \mathsf{log}_2(\mathsf{nat}(u - l) + 1) \rceil + 40$, where $\mathsf{nat}(a) = max(a, 0)$.

Since CRs are syntactically quite close to *Recurrence Relations* (*RRs* for short), in most resource analysis frameworks, it has been assumed that cost relations can be easily converted into *RRs*. This has led to the belief that it is possible to use existing *Computer Algebra Systems* (CAS for short) for finding closed forms of the relations generated by resource analysis. As it will be shown, cost relations are far from *RRs*, and using CAS to obtain closed-form upper bounds is, in general, not practical, and requires a considerable amount of human intervention in many phases.

The main idea in the approach used in the COSTA system is to view CRSs as *programs*, and then use semantic-based static-analysis and program-transformations techniques in order to infer closed-form upper bounds [8]. We first explain the basic ideas on small examples, then we explain how they can be extended to the general case.

4.1 Bounds on the Number of Applications of Equations

The first dimension of the problem of obtaining closed-form upper bounds is to bound the number of recursive calls in each relation, which directly affects the number of times an equation can be applied. Consider, for example, the following cost relation:

$$\begin{array}{ll} C(n) = 3 & \{n \leq 0\} \\ C(n) = 9 + C(n') & \{n > 0, n' < n\} \end{array}$$

An evaluation of an initial call $C(v)$, where v is an integer value works as follows: if $v \leq 0$ then we apply the first equation and we accumulate 3 units to the cost, and if $v > 0$ then we apply the second equation, which in turn accumulates 9 units to the cost plus the cost of the recursive call $C(v')$ where v' is an integer

number such that $v' < v$ (which corresponds to the constraint $n' < n$). Clearly, if I_r and I_b are, respectively, upper bounds on the *number of applications* of the recursive and base-case equations, then $9 * I_r + 3 * I_b$ is an upper bound on the corresponding cost. In the above example, in each recursive call the argument of C decreases at least by 1 (since $n' < n$), and therefore the maximum number of applications of the second equation is $I_r = n_0$, where n_0 corresponds to the (initial) input value, and $I_b = 1$, since the base-case equation is applied only once. Note that when n_0 is negative, we do not make any recursive call, therefore in order to capture these cases we define $I_r = \mathsf{nat}(n_0)$ where $\mathsf{nat}(a) = max(a, 0)$. Putting everything together we obtain that an upper bound for the call $C(n_0)$ is $9 * \mathsf{nat}(n_0) + 3$.

The above example demonstrates that inferring how the values of arguments change during evaluation plays an important role in bounding the number of application of each equation. This change might come in different forms, for example if we change the second equation in the above CR to

$$C(n) = 8 + C(n') \quad \{n > 0, n' \leq \tfrac{n}{2}\}$$

then C's argument decreases by at least half in each recursive call, and therefore the maximum number of application of the recursive equation is $I_r = \lceil \mathsf{log}_2(\mathsf{nat}(n_0) + 1) \rceil$ which in turn implies that the upper bound would be $8 * \lceil \mathsf{log}_2(\mathsf{nat}(n_0) + 1) \rceil + 3$.

Another important factor that affects the number of applications of the different equations is the number of recursive calls in a single equation. For example, assuming that the recursive equation in the above CR is of the form

$$C(n) = 7 + C(n') + C(n'') \quad \{n > 0, n' < n, n'' < n\}$$

then the recursive equation would be applied in the worst-case $I_r = 2^{\mathsf{nat}(n_0)} - 1$ times, because each call generates 2 recursive calls, and in each call the argument decreases at least by 1. Note that $2^{\mathsf{nat}(n_0)} - 1$ corresponds to the number of internal nodes which a complete binary tree of height $\mathsf{nat}(n_0)$ has. In addition, unlike the above examples, the base-case equation would be applied in the worst-case $I_b = 2^{\mathsf{nat}(n_0)}$ times, and therefore the upper bound would be $7 * (2^{\mathsf{nat}(n_0)} - 1) + 3 * 2^{\mathsf{nat}(n_0)}$.

In general, a CR does not include only two equations as above. It may include several base cases and/or several recursive equations. In addition, equations are not necessarily mutually exclusive, which means that at each evaluation step there are several equations that can be applied. For example, if all three recursive equations that we have seen above are defined in the same CR, then the upper bound would be $max([7, 8, 9]) * (2^{\mathsf{nat}(n_0)} - 1) + 3 * 2^{\mathsf{nat}(n_0)}$. Note that the worst-case for the cost of each application is determined by the first equation, which contributes the largest cost, i.e., 9. The worst case for the number of applications of the recursive case is determined by the third equation, which has two recursive calls.

As we explained at the beginning, the problem of bounding the number of applications of each equation is related to bounding the number of consecutive

recursive calls, which has been extensively studied in the context of termination analysis. Automatic termination analyzers usually prove that an upper bound of the consecutive recursive calls exists by proving that there exists a function f from the loop's arguments to a *well-founded* partial order, such that f decreases in any two consecutive calls. This, in turn, guarantees the absence of infinite traces, and therefore termination. These functions are usually called *ranking functions*. If instead of proving that such function exists we actually compute one, then we can use it as the upper bound on the number of consecutive calls, which in turn can be used to bound the number of applications.

4.2 Bounds on the Cost of Equations

In the above examples, in each application the corresponding equation contributes a constant number of cost units. This is not the case in general. For example, it is common to have a CR of the following form:

$$\begin{array}{ll} C(n) = 3 & \{n \leq 0\} \\ C(n) = \mathsf{nat}(n+2)^2 + C(n') & \{n > 0, n' \leq \frac{n}{2}\} \end{array}$$

where in the second equation we accumulate a non-constant, i.e., $\mathsf{nat}(n+2)^2$, number of units in each application. In equations with a non-constant direct cost expression, a closed-form upper bound can be obtained by considering the worst-case (the maximum) value that the expression can be evaluated to, multiplied by the number of applications of the corresponding equation. For example, in the above equation, the maximum value that the expression $(n+2)$ can be evaluated to is (n_0+2), and therefore we would produce the upper bound $\mathsf{nat}(n_0+2)^2 * \lceil \mathsf{log}_2(\mathsf{nat}(n_0)+1) \rceil + 3$.

In order to infer the maximum value of non-constant expressions automatically, we first infer *invariants* (linear relations) between the equation's variables and the values which such variables had at the initial call, and then *maximize* the expression w.r.t. these values. For the above example we would infer the relation $\{n_0 \geq n > 0\}$, from which we can see that the maximum value for n is n_0. This in turn implies that (n_0+2) is the maximum value of $(n+2)$ and therefore $\mathsf{nat}(n_0+2)^2$ is the maximum value for $\mathsf{nat}(n+2)^2$. Again, if several recursive equations are involved, we should combine them all using the *max* operator on the corresponding expressions, as we have done above.

4.3 The General Case

In all the above examples, a single relation was involved and all recursions were direct. We refer to such CRs as *stand-alone* CRs. This is not the case in general. Instead, in most cases, CRSs consist of several CRs with complex call graphs. In order to cope with this, we first transform the given CRS into a structured form with only direct recursions, and incrementally apply the above techniques. We do so by first applying *Partial Evaluation* [32], a well-known program transformation technique, to each of the strongly connected components (SCC) in the

corresponding call graph. By applying partial evaluation starting from a cover point of the SCC, it is guaranteed that get rid of mutual recursion. After partial evaluation, there must be at least one stand-alone CR (does not call any other CR), therefore we can apply the techniques described above in order to solve all these stand-alone CRs. Substituting the results in their calling contexts results in more stand-alone CRs that can in turn be solved using the above techniques again. This process is repeated until there are no more CRs left.

4.4 Obtaining an Upper Bound for the Running Example

The CRS for the running example, shown in Figure 5, contains multiple relations and includes non-direct recursion, which avoids obtaining a closed-form upper bound in a compositional way. As explained above, the first step is to transform the CRS into an equivalent one where we have only direct recursion. The CRS depicted in Figure 5 has 2 SCCs: the first SCC only has Equation 1, and it is not recursive; the second SCC contains Equations 2–14 and corresponds to the loop in *binSearch* and is therefore recursive. After applying partial evaluation to the recursive SCC we obtain the following transformed CRS:

$$
\begin{array}{lll}
(1)\ \mathit{binSearch}(t,v,l,u) & = \mathit{loop}(t,v,l,u,0) & \\
(2)\ \mathit{loop}_b(t,v,l,u,m) & = 5 & \{l{>}u\} \\
(3)\ \mathit{loop}_b(t,v,l,u,m) & = 16 & \{l{\leq}u\} \\
(4)\ \mathit{loop}_b(t,v,l,u,m) & = 24 + \mathit{loop}_b(t,v,l',u,m') & \\
 & & \{l{\leq}u, m' \in [\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}], l'{=}m'{+}1\} \\
(5)\ \mathit{loop}_b(t,v,l,u,m) & = 24 + \mathit{loop}(t,v,l,u',m') & \\
 & & \{l{\leq}u, m' \in [\frac{l+u}{2}-\frac{1}{2}, \frac{l+u}{2}], u'{=}m'{-}1\}
\end{array}
$$

The recursive SCC has been transformed into Equations 2–5 above. Note that Equations 3–5 have the same guard ($l{\leq}u$), which results in a non-deterministic CR. The reason for this is that Equation 3 corresponds to the case where $t[m] == v$, Equation 4 to the case where $t[m] > v$ and Equation 5 to the case where $t[m] < v$. However, the value of $t[m]$ is not observable at the cost relation level and even though the original program is deterministic, its associated CRS is not.

Solving the above CRS starts by solving the stand-alone CR which consists in Equations 2–5. Note that Equations 2–3 are the base-cases and 4–5 are the recursive ones. Examining the recursive equations and their attached constraints, we can automatically infer that the difference between the values of u and l decreases logarithmically at each recursive call and, in particular, we can provide $I_r = \lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 1$ as an upper bound for the number of applications of the recursive equation. The base-case equations are applied only once. Therefore the closed-form upper bound is:

$$\mathit{loop}_b(t_0, v_0, l_0, u_0, m_0) = 24 * (\lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 1) + 16$$

This closed form can then be substituted in Equation 1 and after some simplification we obtain the following closed-form upper bound for *binSearch*:

$$\mathit{binSearch}(t_0, v_0, l_0, u_0) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u_0 - l_0) + 1) \rceil + 40$$

In order to illustrate the use of invariants and the maximization of cost expressions, let us now assume that the method *binSearch* is used in another method as follows:

```
public static int m(int[ ] t){
  int c=0;
  int u=t.length;
  for (int i=0; i<u;i++)
    if (binarySearch(t,i,i,u) != -1 ) c++;
  return c;
}
```

and that we are interested in inferring closed-form upper bounds for m. We first build the CRS that corresponds to m. For brevity, we show the CRS after partial evaluation:

$$
\begin{array}{lll}
(1)\ m(t) & = 9 + loop_m(t,c,u,i) & \{i{=}0, u{=}t, c{=}0\} \\
(2)\ loop_m(t,c,u,i) & = 3 & \{i{\geq}u\} \\
(3)\ loop_m(t,c,u,i) & = 12{+}binSearch(t,i,i,u){+}loop_m(t,c,u,i') & \{i{<}u, i'{=}i{+}1\} \\
(4)\ loop_m(t,c,u,i) & = 13{+}binSearch(t,i,i,u){+}loop_m(t,c',u,i') & \{i{<}u, i'{=}i{+}1, c'{=}c{+}1\}
\end{array}
$$

First we solve the CR $loop_m$ (Equations 2–4). We start by substituting the close-form upper bound of $binSearch_m$ in the corresponding calls and we obtain the following stand-alone CR

$$
\begin{array}{ll}
loop_m(t,c,u,i) = 3 & \{i{\geq}u\} \\
loop_m(t,c,u,i) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 52{+}loop_m(t,c,u,i') & \{i{<}u, i'{=}i{+}1\} \\
loop_m(t,c,u,i) = 24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 53{+}loop_m(t,c',u,i') & \{i{<}u, i'{=}i{+}1, c'{=}c{+}1\}
\end{array}
$$

Examining the recursive equations and their attached constraints we automatically infer that the maximum number of applications of the recursive equations is $\mathsf{nat}(u_0 - i_0)$, since i increases by one until it reaches u (which does not change). Now, in order to infer the closed-form upper bound we need to approximate the maximum values that $\mathsf{log}_2(\mathsf{nat}(u-i)+1)$ can be evaluated to. This happens when $u-i$ is maximal. This occurs for the maximal values of u and the minimal values of i. Since the invariant that we infer includes $i \geq i_0$ and $u \leq u_0$, we can conclude that $u_0 - i_0$ is the maximum value to which $u - i$ can be evaluated. Therefore the closed-form upper bound for $loop_m$ is:

$$loop_m(t_0,c_0,u_0,i_0) = \mathsf{nat}(u_0-i_0) * (24 * \lceil \mathsf{log}_2(\mathsf{nat}(u_0-i_0)+1)\rceil + 53) + 3$$

Substituting this upper bound in the first equation results in a non-recursive CR which consists in a single equation:

$$m(t) = 9 + \mathsf{nat}(u-i) * (24 * \lceil \mathsf{log}_2(\mathsf{nat}(u-i)+1)\rceil + 53) + 3\ \{i{=}0, u{=}t, c{=}0\}$$

and since in this context we have $i = 0$ and $u = t$, we can conclude with the following closed-form upper bound for m:

$$m(t) = 24 * \mathsf{nat}(t) * \lceil \mathsf{log}_2(\mathsf{nat}(t)+1)\rceil + 53 * \mathsf{nat}(t) + 12\,)$$

5 Application to Resource Certification

In order to motivate the interest of *resource usage certification* or (resource certification for short), we will start by describing *mobile code*. Nowadays, the use of mobile code is widespread. It includes, for example running applets and/or plug-ins downloaded from the net in a web browser or a mobile phone. The current approach to security of mobile code is a combination of static verification of certain properties, which guarantees a certain level of security, with dynamic checking, which supervises all operations which are still potentially unsecure after the static verification. For example, in the Java Virtual Machine, mobile code is subject to *bytecode verification* before being executed, while operations such as array indexing are checked at runtime. Bytecode verification, if successful, provides a number of guarantees on the program, such as being well typed, with jumps to existing instructions, etc. Note that if the bytecode verification process fails, the program is discarded.

Ideally, one would like to extend this model in order to include more sophisticated *security policies* in the static verification part. In particular, and as already sketched in Section 1, the purpose of resource certification is to consider resource usage bounds as security policies. This means that prior to executing a program, it must be guaranteed that the program satisfies a given resource usage policy. This problem can be formulated in two ways. One is to have an automatic system which given a program and a resource usage policy answers *yes* only if it succeeds to prove that the program satisfies the policy. Alternatively, we can split this process in two steps: first, an automatic system obtains an upper bound on the resource usage of the program and second, another automatic system, which in what follows we refer to as *comparator*, checks whether the computed upper bound is smaller than or equal to the resource usage policy for any possible input value. We advocate for the second alternative because we believe it is more flexible: we first use COSTA on the code producer side to infer upper bounds which are independent of any resource policy and consumer, and then, on the code consumer side we check whether the upper bound abides by the policy.

5.1 An Example of Resource Certification

We illustrate through a simple example the fundamental intuition behind resource certification. Let us assume a resource usage policy for method m in Figure 1 that imposes a resource usage policy, which we call *policy*, on the number of instructions executed of:

$$policy{=}60 * [\mathsf{nat}(t)]^2 + 120 * \mathsf{nat}(t) + 13$$

COSTA infers the upper bound $ub{=}24{*}\mathsf{nat}(t){*}\lceil \mathsf{log}_2(\mathsf{nat}(t){+}1)\rceil{+}53{*}\mathsf{nat}(t){+}12$. The code will be acceptable, provided that *policy* is guaranteed, i.e., $ub{\leq}policy$, which happens to be the case in our example and that the comparator succeeds to prove it.

Developing a comparator which handles closed-forms that involve logarithmic, exponential, polynomial expressions, etc. is far from trivial. Based on the ideas in [26], we are currently implementing in COSTA the basics of such comparator, but it is still subject of ongoing work.

Also, though in some contexts, especially when considering memory usage, non-asymptotic policies are to be expected, sometimes it is more reasonable that the policy is asymptotic. In COSTA, policies are currently non-asymptotic and handling of asymptotic policies is also subject of ongoing work. Coming back to our previous example, this would result in a new *policy*′ s.t. *policy*′=$[\mathsf{nat}(t)]^2$. The comparator should again be able to prove that *policy*′ is satisfied by method m.

5.2 Scenarios for Resource Certification

Within the alternative we propose, in which code certification is performed in two steps, there are several *scenarios* one could imagine. We now describe three different ones.

The Consumer-based Scenario In this first scenario, it is the sole responsibility of the mobile code consumer to both obtain an upper bound and to compare such upper bound with the policy. This scenario is simple, since it does not involve any further actor, but it is inefficient, since the mobile code has to be certified separately for every consumer. Also, it may be unfeasible on devices with limited computing power, such as mobile phones.

The Server-based Scenario In this second scenario, there is an additional actor which acts as the server of the mobile code. Such server not only distributes the mobile code. It also computes once and for all an upper bound for it. Assuming that this server is trusted by the code consumer, the consumer downloads a bundle which contains both the code and its upper bound. In order to guarantee that the bundle is actually produced by the trusted server, the bundle is signed using standard *Public Key Infrastructure* (PKI) techniques. Then, the code consumer, using the public key of the server, checks that the bundle is correct and uses the provided upper bound. Similarly, the comparison phase could also either be outsourced to trusted servers and be accessed using PKI or be performed locally.

The PCC-based Scenario In this final scenario, the situation is somewhat intermediate between the two other extremes. The main advantage of the pcc-based scenario w.r.t. the server-based one is that in the pcc-scenario the server does not need to be trusted by the code consumer. Unlike the simpler notion of PKI (which merely guarantees that the code has been produced or approved by the signing entity, such as a program, person, or organization), now the pcc server provides an unsigned bundle which contains the code, an upper bound,

and some *verifiable evidence*[3] about the upper bound being correct. Then, the code consumer has to have an automatic (and efficient) system for verifying that the provided upper bound is actually valid for the code, by using the provided evidence.

As it is well know from the proof-carrying code [41] theory, the main advantage of this scenario is that the evidence only needs to be generated once and the verification process which occurs at the consumer side should be much more efficient than computing the upper bounds from scratch. Essentially, the hard work is shifted from the code consumer to the code producer (i.e., the programmer and/or the compiler), which now has to not only produce the code, but also an upper bound and the verifiable evidence which must be bundled with it.

In the case of COSTA, a PCC-based scenario can obtained by using ideas from Abstraction-Carrying Code [7] (ACC), which proposes to use abstract interpretation as enabling technology for PCC. The main idea in ACC is to use, at the producer's side, a fixed point-based static analyzer, in order to automatically infer an abstract model (or simply *abstraction*) of the mobile code which can then be used to prove that this code is secure w.r.t. the given policy in a straightforward way. A simple, easy-to-trust (analysis) verifier at the consumer's side could verify the validity of the information on the mobile code. This verifier could be indeed a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixed point (in contrast to standard analyzers). Furthermore, as the process of inferring the abstraction is fully automatic, the analyzer itself could be used at the consumer side, as discussed in the consumer-based scenario above.

We are currently working on building a PCC infrastructure for COSTA by following the principles of ACC. Since the analyzer computes several abstractions of the program (size relations, invariants, ranking functions, etc.) in order to be able to compute an upper bound, the evidence should in principle contain the fixed points of multiple analyses. However, depending on the analysis times and the amount of space required to store its result, for some analyses it may be more efficient to recompute things on the consumer side than to verify the evidence provided by the server. Thus, there are still important practical decisions regarding which analyses results to include in the evidence and which to recompute on the consumer.

In our opinion, regardless of which of the three scenarios are put into practice, generalized use of resource certification will not be a reality until there are fully automatic resource analyzers available which are capable of computing accurate upper bounds for real-life applications. This is the requirement which COSTA aims at solving. Once this is sufficiently solved, the rest of the infrastructure will be in place relatively easily.

[3] In the original PCC framework, this *evidence* was called *certificate*. We prefer avoiding the use of such terminology since it is already rather overloaded.

6 Related Work

In this section, we review related work by focusing first in existing tools developed for the analysis and transformation of Java bytecode in Section 6.1. Then, in Section 6.2, we give a brief overview of the features that resource analyses have on the different programming paradigms and the most interesting aspects of each of them. Later, in Section 6.3, we compare our system for obtaining closed-form upper bounds with existing solvers. Finally, in Section 6.5, we summarize the work on certification of the resource consumption of programs.

6.1 Tools for Analysis of Java Bytecode

Analysis of Java bytecode is currently an active research area with a number of analysis and transformation tools available. Especially relevant are the analyses developed on the Soot framework [54] and the Julia generic analyzer [50]. Soot is a framework for the development of optimizations and analyses for Java bytecode which already includes points-to, purity, and dynamic data structure analyses, among others. The most similar part between these systems and COSTA is the transformation of the bytecode into an intermediate (procedural) representation. Indeed, intermediate representations are common practice to develop analysis and transformations on JBC. Of relevant importance is BoogiePL [36] as well. The main differences with our RBR are: (1) they do not provide a uniform treatment of all kinds of loops by means of recursion, (2) they do not perform the loop extraction transformation we propose, which is important for compositionality in resource analysis; and (3) the intermediate representation called Shimple in Soot performs SSA, but neither Shimple nor BoogiePL convert stack variables into local variables as COSTA does. In our representation, in one pass, we can eliminate almost all variables which originate from stack variables, which results in a more efficient subsequent size analysis. The Julia Java bytecode analyzer [50] provides a generic analysis engine for which sharing, class, non-nullness, information flow, escape, constancy, and static initialisation analyses have been integrated. Neither Julia nor Soot include a resource analysis, though Julia also contains implementations of some of the pieces (in particular the class, nullity, sharing, and cyclicity analyses) which are required in the size analysis component.

6.2 Resource Analysis for Different Programming Paradigms

Focusing on resource analysis, important effort has been devoted to extend Wegbreit's framework [57] to different languages and programming paradigms. The main objective in this task is to define a resource analysis framework in which it is possible to generate CRS from the programs in the corresponding language. As mentioned in Section 1, most of the extensions to Wegbreit's framework have taken place in the context of high-level declarative languages, whose recursive structure simplifies the process of generating cost relations. In general, these analyses consider languages without a mutable heap, and they do not deal with

objects and exceptions as in our case. We are not aware of any work, apart from ours, that applies Wegbreit's framework to imperative languages. Below we review several frameworks defined for the corresponding declarative programming paradigms.

Cost Analysis in Functional Programming. Early work on resource analysis [57,35,44] was developed for a first order subset of Lisp. Rosendahl [44] presented a system based on transforming a program into a step-counting version which was then analyzed by relying on abstract interpretation. The result of such analysis was expressed as a CRS which was then attempted to be transformed into a closed form by relying on a series of source-to-source transformations. Theoretical advances for analyzing lazy functional languages were made by [56] and [15]. They used projections and demand analysis to model a call-by-need reduction strategy of typed lambda calculus. Still in the context of functional languages, the technique of cost counting programs mentioned above [44,35] was extended in [47] to higher-order programs. Recent work [33] describes a complexity analysis for programs extracted from proofs carried out with the Coq proof assistant. The generated CRSs are solved in this case by relying on MAPLE. Again, the first transformational part is not required and size analysis does not have to deal with object-oriented features. An automatic complexity analysis for computing upper bounds on the time complexity of higher-order Nuprl programs is presented in [14]. The analysis derives recursive cost equations which are passed to Mathematica. In general, in functional programming, resource analysis focuses on dealing with higher-order functions and lazy evaluation.

Cost Analysis in Logic Programming. One of the first resource analysis frameworks [22] was developed in the context of logic programming. In this setting, resource analysis needs to consider peculiar features of logic languages, such as approximating the number of solutions (due to non-deterministic computations), type and mode inference, and non-failure information. The CASLOG system [22] was designed to solve CRSs for logic program and it is currently used in the `CiaoPP` system [28]. As in functional programming, obtaining CRSs is simplified by the fact that they already start from a recursive programming language where recursion is the only form of iteration. Also, size analysis in logic programming differs from ours as it does not support object-oriented features. The resource analysis integrated in the `CiaoPP` system includes a resource analysis [40] based on a size analysis for logic programs and hence differs fundamentally from ours.

6.3 Systems for Computing Closed-Form Upper-Bounds

There are two main ways of viewing CRSs which lead to different mechanisms for finding closed-form upper-bounds. We call the first view *algebraic* and the second view *transformational*. The algebraic one is based on regarding CRSs as *recurrence relations*. This view was the first one to be proposed and it is the one which is advocated for in a larger number of works. It allows reusing the large existing body of work in solving recurrence relations. Within this view,

two alternatives have been used in previous analyzers. One alternative consists in implementing restricted recurrence solvers based on standard mathematical techniques within the analyzer, as done in e.g. [57,22]. The other alternative, motivated by the availability of powerful *computer algebra systems* (CASs for short) such as Mathematica, MAXIMA, MAPLE, etc., consists in connecting the analyzer with an external solver, as proposed in [56,47,14,6,38].

The transformational view consists in regarding CRSs as (functional) programs. In this view, closed-form upper-bounds are produced by applying (general-purpose) program transformation techniques on the CRS [44] until a non-recursive program is obtained. The transformational view was first proposed in the ACE system [35], which contained a large number of program transformation rules aimed at obtaining non-recursive representations. It was also used by Rosendahl in [44], who later in [45] provided a series of program transformation techniques based on super-compilation [53] which were able to obtain closed-forms for some classes of programs.

The need for improved mechanisms for automatically obtaining closed-form upper-bounds was already pointed out in Hickey and Cohen [29]. A significant work in this direction is PURRS [10], which has been the first system to provide, in a fully automatic way, non-asymptotic closed-form upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRSs to be deterministic. The problem with all the approaches mentioned above is that, though they can be successfully applied for obtaining closed-forms for CRSs generated from simple programs, they do not fulfill the initial expectations in that they are not of general applicability to CRSs generated from real programs.

The main motivation for developing the solver [8] that we use in COSTA was our own experience in trying to apply the algebraic approach on the CRSs generated by [6]. We argue that automatically converting CRSs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. In contrast, our approach can produce correct and comparatively simple results even in the presence of non-determinism.

6.4 Other Approaches to Cost Analysis

In the imperative programming paradigm, most of the work has been done by the real-time and embedded systems community. It has mainly focused on real-time aspects, with major inroads made in WCET analysis, see e.g. [23], which is technically different from our resource analysis, the main similarity being the need to infer upper bounds on the number of iterations of loops.

There exist other approaches to resource analysis which are not based on Wegbreit's framework. These include analyses based on type-and-effect systems [11,49,55]. Type-and-effect systems [42] are a well-known technique for automatic program analysis. They main difference w.r.t. abstract interpretation approaches like ours is that they avoid having the implementation of specialised

inference engines that may be required by abstract interpretation and they simplify the construction of the soundness proofs through analogy with similar and well-understood proofs for the underlying type system. The latest work by [49] uses a type-and-effect system based on Hindley-Milner types to expose constraints on sized types [31] for higher order, recursive functional programs, to provide improved quality of resource analysis. Apart from the underlying differences between the considered languages, in contrast to our proposal, this approach to resource analysis is restricted to linear upper bounds. Besides, the language does not support recursion and the analysis is restricted to a cost model that counts the number of steps. The analysis presented in [11] proposes an extension of the λ-calculus to ensure that resources are correctly used. They also rely on a type-and-effect system to over approximate the set of histories of events (i.e., the usage of resources) that a program can generate at runtime. A model-checking technique then validates such approximations. In essence, this work is focused on the enforcement of resource usage policies, but their techniques cannot be used to generate upper bounds on the resource usage as our method does.

There is also work which studies the relationship between syntactical constructions of programming languages and their computational complexity [34,12]. These analyses are developed on simple imperative languages which are far from our bytecode and, in contrast to our work, they cannot be used to compute non-asymptotic upper bounds.

The work in [39] shows how to apply sub-interpretation (firstly used in first order functional programming to deal with computational complexity) to object-oriented programs without recursion in order to provide upper bounds on their stack usage. This approach is restricted to polynomial bounds and to the particular resource of stack usage.

More recent work develops resource analyses to estimate the memory consumption. In particular, [16] describes a technique for Java-like languages which computes symbolic polynomial approximations of the amount of memory required by a program. The work by [19] studies the memory consumption (including both heap space and stack usage) of low-level programs which are similar to our bytecode programs. In both cases, the analyses are less general than ours, both in the kind of properties they can estimate (specific to memory consumption) and in the kind of upper bounds that they can generate (polynomial bounds).

The SPEED system [25,24] is able to automatically compute symbolic complexity bounds of procedures written in C/C++. The basic idea of their methodology is to instrument monitor variables to count the number of loop iterations and then statically compute an upper bound on these counter variables in terms of programming inputs using invariant generation tools. They allow the user the possibility of defining some quantitative functions over abstract data-structures to avoid the need of shape analysis. Besides, SPEED performs some program transformations to improve the precision of the analysis when inferring bounds

on certain types of loops. Some of these ideas could be applied in order to improve our framework.

6.5 Resource Usage Certification

As already mentioned in Section 5, resource usage certification [21,9,30,18,43] proposes the use of security properties involving resource requirements, i.e., that the untrusted code adheres to specific bounds on resource consumption. Related work in the context of Java bytecode includes the work in the MRG project [9], which can be considered complementary to ours. MRG focuses on building a proof-carrying code [41] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The cost model which has been used to develop the analysis is heap consumption, since applications to be deployed on devices with a limited amount of memory, such as smartcards, should be rejected if they require more memory than that available. The framework is restricted to polynomial bounds and to the above cost model, while our resource analysis can infer a wider set of bounds (including exponential, algorithmic, etc.) and it is parametric with respect to the cost model. More related work is the one proposed by [17], where a resource usage analysis is presented. Again, this work focuses on one particular notion of cost, memory consumption, and it aims at verifying that the program executes in bounded memory by making sure that the program does not create new objects inside loops, but it does not infer resource usage bounds. The analysis has been certified by proving its correctness using the Coq proof assistant. Compared to previous work, our system shows, for the first time, that it is possible to automatically generate resource bounds guarantees, not restricted to polynomial bounds, for a realistic mobile language.

7 Conclusions and Future Perspectives

In this paper we have illustrated, by means of examples, how the COSTA system performs resource analysis. The analysis is done in two steps. First, *cost relation systems* are generated for an input bytecode w.r.t. a *cost model*. Such relations provide useful approximations of the resource usage of the program w.r.t. the considered cost model, in terms of the size of the input arguments, and provided an accurate *size analysis* is used to establish relationships between arguments. Second, *closed-form upper bounds* for the cost relation systems are obtained. This is possible provided that *ranking functions* are found for all loops which affect the cost and that accurate *invariants* are obtained. To the best of our knowledge, COSTA is the first system to perform fully automatic resource analysis of object-oriented bytecode and we believe that COSTA opens the door to the application of *resource usage analysis* in the context of general purpose applications written in mainstream programming languages.

Though the efficiency and robustness of the system can be considerably improved, COSTA can already deal with a relatively large class of JBC programs, and gives reasonable results in terms of precision and efficiency for different

cost models: the number of executed bytecode instructions, heap consumption, and number of calls to user-specified methods. We plan to distribute the system as free software soon. Currently, it can be tried out through a web interface available from the COSTA web site: `http://costa.ls.fi.upm.es`.

The system can deal with most features of JBC. However, non-sequential code, dynamic class loading and reflection are not supported. Java API methods used by programs are analyzed much in the same way as user code, since their bytecode is available to the analyzer. As for native code, i.e., methods not implemented in Java, calls to native methods are shown in upper bounds as symbolic constants, since the code for those methods is not written in Java and COSTA cannot analyze them. This could be further improved by providing assertions which describe the cost of the native method for the different cost models and (optionally) a safe approximation of their input-output behavior, but it is not supported.

In addition to the web interface, COSTA has a command-line interface and an Eclipse plugin which make interaction with the analyzer quite straightforward, even during program development. The different interfaces allow customizing the behaviour of COSTA by modifying the value of several options, including:

1. whether the code of Java API classes should also be analyzed or not;
2. whether auxiliary analyses (sign, nullity, slicing, constant propagation) should be included, thus possibly improving both precision and performance;
3. whether input-output size relations have to be computed (Section 3.3);
4. if exceptions, either explicitly thrown in the code or resulting from semantic violations, have to be taken into account;
5. which cost model has to be considered.

Also, although not discussed in this paper, COSTA also performs termination analysis of Java bytecode programs (see [3]). When COSTA fails to find an upper bound for a program, sometimes it may be useful to try and find out whether the program is guaranteed to terminate. Maybe COSTA fails because the program contains a bug and loops unexpectedly with a non-zero cost associated to each iteration of the loop. In that case, there exist no upper bound for the program and there is no way that COSTA can find an upper bound. Although COSTA results are safe, they are obviously incomplete, since finding an upper bound is an undecidable problem. This means that there are programs for which it is possible to find an upper bound, but COSTA fails to find one.

As regards future work, there are plenty of ways in which both the theoretical foundations and the practical implementation can be improved in order to handle a larger class of programs, and obtain improvements both in terms of efficiency and accuracy. On the foundations side, progress in the area of object-oriented languages of any of the analyses used by the system will be potentially applicable to COSTA. For example, one of the most challenging problems is to account for loops and recursion where the number of iterations depends on *numeric fields*. Here, an approach working in all cases might not be practical; however, heuristics may allow us to account for special, simple but quite common cases which can significantly enlarge the class of analyzable programs. A first step in

this direction, in the context of termination analysis, has been taken in [4]. On the implementation side, currently, COSTA handles bytecode programs for Java SE 1.4.2_13 and Java ME. The reason for this is that Java SE 1.4.2_13 is the version of Java taken as starting point for Java ME and, in particular, for MIDP. The latter is the profile used by mobile phone applications, i.e., *midlets*, which are the main target in the MOBIUS project. However, there is no fundamental reason for not supporting more recent versions of Java and we plan to extend COSTA to also handle Java 5 and 6 soon.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. The Timber Language. http://www.timber-lang.org.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *Proceedings of the IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.
4. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In Marieke Huisman, editor, *10th Workshop on Formal Techniques for Java-like Programs*, July 2008.
5. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. The COSTA System web site. `http://costa.ls.fi.upm.es`.
6. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
7. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code: A Model for Mobile Code Safety. *New Generation Computing*, 26(2):171–204, March 2008.
8. Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.
9. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

10. R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, 2005. `arXiv:cs/0512056` available from `http://arxiv.org/`.
11. M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007*, volume 4423 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2007.
12. Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.
13. Florence Benoy and Andy King. Inferring Argument Size Relationships with CLP(R). In *Workshop on Logic-based Program Synthesis and Transformation (LOPSTR)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, August 1997.
14. R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.
15. B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.
16. V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.
17. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *13th International Symposium on Formal Methods (FM'05)*, number 3582 in LNCS, pages 91–106. Springer-Verlag, 2005.
18. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.
19. W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.
20. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
21. K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.
22. S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
23. Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20. IEEE Computer Society, April 2006.
24. B. S. Gulavani and S. Gulwani. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*, LNCS 5123, pages 370–384. Springer, 2008.
25. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

26. Sumit Gulwani and Ashish Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
27. K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2003.
28. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
29. T. Hickey and J. Cohen. Automating program analysis. *J. ACM*, 35(1), 1988.
30. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.
31. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423, 1996.
32. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
33. J. Jouannaud and W. Xu. Automatic Complexity Analysis for Programs Extracted from Coq Proof. *ENTCS*, 2006.
34. Lars Kristiansen and Neil D. Jones. The flow of data and the complexity of algorithms. In S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.
35. D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.
36. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.
37. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
38. Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program transformation by solving recurrences. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 121–129, New York, NY, USA, 2006. ACM.
39. J.-Y. Marion and R. Pèchoux. Resource control of object-oriented programs. In *International LICS affiliated Workshop on Logic and Computational Complexity (LCC 2007)*, Wroclaw, Poland, 2007.
40. J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.
41. G. Necula. Proof-Carrying Code. In *Proc. of ACM Symposium on Principles of programming languages (POPL)*, pages 106–119. ACM Press, 1997.
42. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005. Second Ed.
43. K-H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
44. M. Rosendahl. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

45. M. Rosendahl. Simple driving techniques. In T. Mogensen, D. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 404–419. Springer, 2002.
46. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*. S-V, 2006.
47. D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.
48. S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *Static Analysis Symposium (SAS)*, pages 320–335, 2005.
49. H. R. Simões, K. Hammond, M. Florido, and P. B. Vasconcelos. Using intersection types for cost-analysis of higher-order polymorphic functional programs. In *Types for Proofs and Programs, International Workshop, TYPES 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 221–236. Springer, 2006.
50. F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.
51. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
52. Fausto Spoto, Patricia M. Hill, and Etienne Payet. Path-length analysis of object-oriented programs. In *Proc. International Workshop on Emerging Applications of Abstract Interpretation (EAAI)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
53. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
54. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.
55. P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag, September 2003.
56. P. Wadler. Strictness analysis aids time analysis. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 119–132. ACM Press, 1988.
57. B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.