Applications of Static Slicing in Cost Analysis of Java Bytecode

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

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

1 Introduction

Java bytecode [8] is a low-level object-oriented language which is widely used in the context of mobile code due to its security features and the fact that it is platform independent. Recent works study advanced properties of Java bytecode like cost analysis [2] or termination [1]. Automatic cost analysis has interesting applications in the context of Java bytecode. For instance, the receiver of the code may want to infer cost information in order to decide whether to reject code which has too large cost requirements in terms of computing resources (in time and/or space), and to accept code which meets the established requirements [7, 5, 6]. Also, in parallel systems, knowledge about the cost of different procedures in the object code can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

Given an input program, cost analysis aims at inferring *Cost Equations Systems* (CES) which define the cost of the program as a function of (some of) its data input size. CES are a general form of describing the resource consumption of programs and, in a particular application, they can be used to infer heap consumption [4], number of executed bytecode instructions [3], etc. Essentially, CES are generated by abstracting the structure of the program such that when the program contains an iteration the CES contains a corresponding recursion, which in addition includes information about how the sizes of the different variables change when the program goes through this recursion. The traditional approach is to infer upper bounds of the cost by solving the CES, for instance using *Computer Algebra Systems* (CAS) like Mathematica, Maple, etc.

The approach to cost analysis as described in [2] makes two initial steps in order to obtain a structured representation for the bytecode:

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

In the last step, *size relations analysis* is applied to the above intermediate representation and a CES is generated from such representation. Traditionally, the CES

```
class a {
  static int sum(int m, int n) {
    int res=0;

    for (int i=1; i<=m; i++) {
      for (int j=i; j<=n; j++) {
        res += i*j;
      }
    }
    return res;
  }
}
```

```
 0: iconst_0       18: iload_2
 1: istore_2       19: iload_3
 2: iconst_1       20: iload 4
 3: istore_3       22: imul
 4: iload_3        23: iadd
 5: iload_0        24: istore_2
 6: if_icmpgt 37   25: iinc 4, 1
 9: iload_3        28: goto 12
10: istore 4       31: iinc 3, 1
12: iload 4        34: goto 4
14: iload_1        37: iload_2
15: if_icmpgt 31   38: ireturn
```

Fig. 1. Java Program and its corresponding Java bytecode

is then solved using existing CAS in order to obtain a closed from upper bound on the cost. An important observation is that many of the rules arguments may not be relevant to the cost. For instance, typical *accumulating* parameters which merely keep the value of some temporary result do not affect the control flow nor the cost of the program. Such tools are more likely to fail in providing useful information if the corresponding CES includes information which originates from the program's parts that do not affect its cost. Therefore, eliminating those superfluous parts from the CES is crucial in practice.

Basically, given a rule, the arguments which can have an impact on the cost of the program are those which may affect directly or indirectly the program guards (i.e., they can affect the control flow of the program), or are used as input arguments to external methods whose cost, in turn, may depend on the input size. The problem of computing a safe approximation of these arguments can be formalized as a backwards slicing problem using the reachable guards and external methods as the slicing criterion [9].

In this abstract, we discuss several aspects of the role of *static slicing* to minimize the number of arguments which need to be taken into account in CES. We will focus on two benefits of eliminating the arguments which do not have an impact on the cost of the program. On one hand, analysis can be more efficient if we reduce the number of variables. And also CES are more likely to be solved by standard CAS.

2 Cost Analysis of Java Bytecode by Example

We illustrate the cost analysis of [2] on the example depicted in Fig. 1. The Java program (on the left) and its corresponding Java bytecode (on the right) define a method *sum* that, for a given two integer values n and m, computes the sum:

$$res = \sum_{i=1}^{m} \sum_{j=i}^{n} i * j \tag{1}$$

The Java program is provided here just for clarity, the analysis is performed directly in the Java bytecode program. Computing a *closed form* function which is an upper bound to the cost of *sum* (i.e., the number of bytecode instructions that may be executed) in term of its input arguments is done in several steps: (1) recovering the structure of the Java bytecode program by mean of a set of control

flow graphs (CFGs); (2) transforming the CFGs into a rule-based representation; (3) inferring *size relations* between the program variables and generate cost equations system (CES) from which we can compute a closed form upper bound using standard computer algebra systems (CAS).

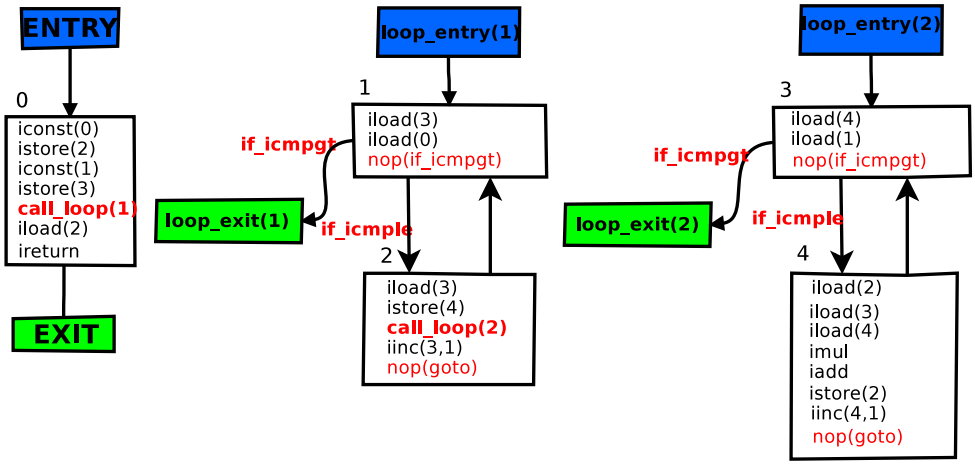

Fig. 2. CFG for the Java bytecode program in Fig. 1

Step I: Control Flow Graph.

In the first step, each JBC instructions sequence (which corresponds to a method) is transformed into a corresponding set of CFGs. This is done by dividing the sequences into maximal sub-sequences of non-branching instructions, which form the basic blocks (nodes) of the initial CFG. Then, the basic blocks are connected by guarded edges that describe the possible transitions. The guards and the edges are introduced by considering the last bytecode instruction of each block. Finally, a loop extraction step is applied on the initial CFG in order to separate those parts that correspond to loops, this is crucial when the program contains nested loop since it allow *compositional* reasoning.

The CFGs of the *sum* method are depicted in Fig. 2. In block 0, the variables i and *res* are initialized (the first 4 bytecode instructions) and then the control is transferred to the middle CFG (using the instruction call_loop) which corresponds to the outer-loop, and, upon return from that loop, the method returns the value *res* (the last two instructions). In block 1 (the entry of the outer-loop), the values of i and n are compared. If $i \leq n$ then the control is transferred to block 2 which corresponds to the loop's body, otherwise the control is transferred to a block which indicates that the loop has terminated and the control is transferred back to the caller. Note that the corresponding edges are annotated by conditions which correspond to $i \leq n$ and $i > n$. Block 2 corresponds to the body of the outer-loop, it first initializes j and then the control is transferred to block 3 which corresponds to the inner-loop, and, upon return it increases i by one. The inner-loop is defined similarly by the CFG on the right.

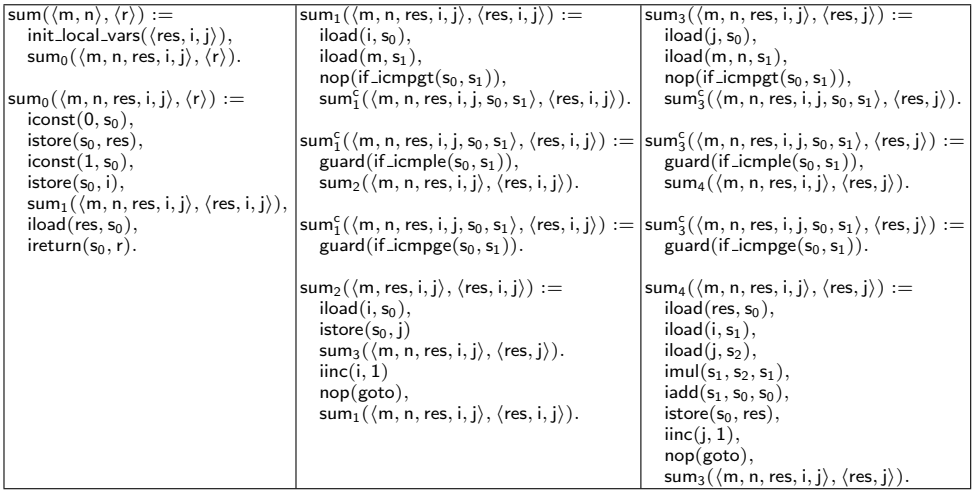

Fig. 3. The Intermediate Represenation for the CFGs of Fig. 2

Step II: Rule-Based Representation.

In the second step, the CFGs are represented in a procedural way by means of *rule-based program*. A *rule-based program* defines a set of *procedures*, each of them defined by one or more rules. Each rule has the form $head(\bar{x}, \bar{y}) := guard, instr, cont$ where *head* is the name of the procedure the rule belongs to, $\bar{x}$ and $\bar{y}$ respectively indicate sequences of input and output arguments, *guard* is of the form $\mathsf{guard}(\phi)$, where ϕ is a Boolean condition on the variables in $\bar{x}$, *instr* is a sequence of bytecode instructions that include explicitly the local variables and stack elements on which they operate, and *cont* indicates a call to another procedure which represents the continuation of this procedure. In principle, $\bar{x}$ should include the local variables for the method and the stack elements at the beginning of the block. In most cases, $\bar{y}$ only includes the return value of the method, which we denote by r, but in rules that correspond to loops it usually includes more variables. The rule-based program(s) depicted in Fig. 3 correspond respectively to the CFGs in Fig. 2. In what follows we explain some of the rules.

The entry rule *sum* (the first one on the left column) is the method's entry. It takes the input variables m and n and returns the output variable r. It first initializes the local variables and then calls the rule sum_0 (which corresponds to block 0). The rule sum_0 takes all local variables (including the method's formal parameters) as input and returns r as output. The instructions $\mathsf{iconst}(\mathsf{0}, \mathsf{s_0})$ and $\mathsf{istore}(\mathsf{s_0}, \mathsf{res})$ initialize *res* to zero, note that s_0 corresponds to a stack position which is explicit in the rule-based representation. Similarly, $\mathsf{iconst}(\mathsf{1}, \mathsf{s_0})$ and $\mathsf{istore}(\mathsf{s_0}, \mathsf{i})$ initialize i to one. Then the outer-loop is called using $\mathsf{sum_1}(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j} \rangle, \langle \mathsf{res}, \mathsf{i}, \mathsf{j} \rangle)$, and upon return, the last two instructions $\mathsf{iload}(\mathsf{res}, \mathsf{s_0})$ and $\mathsf{ireturn}(\mathsf{s_0}, \mathsf{r})$ bind the output variable r to the return value of the method *sum*. Note that when calling the outer-loop sum_1, the list of output arguments includes only those that might be modified during the execution of sum_1. The rule sum_1 (which corresponds to block 1) is the entry rule to the outer-loop. The fact that block 1 has two successors, block 2 and the loop-exit block, is expressed in sum_1 by a call to a continuation rule sum_1^c (at the end of sum_1) which in turn is defined by two rules: the first one for to

$$
\begin{array}{lll}
(1)\ \mathsf{sum}(\mathsf{m},\mathsf{n}) & = \mathsf{sum}_0(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\underline{\mathbf{j}}) & \{res=0, i=0, j=0\} \\
(2)\ \mathsf{sum}_0(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\underline{\mathbf{j}}) & = 6+\mathsf{sum}_1(\mathsf{m},\mathsf{n},\underline{\mathbf{res}}',\mathsf{i}',\underline{\mathbf{j}}) & \{res'=0, i'=1\} \\
 & & \\
(3)\ \mathsf{sum}_1(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) & = 3+\mathsf{sum}_1^{\mathsf{c}}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & \{s_0=i, s_1=m\} \\
(4)\ \mathsf{sum}_1^{\mathsf{c}}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & = \mathsf{sum}_2(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) & \{s_0 \leq s_1\} \\
(5)\ \mathsf{sum}_1^{\mathsf{c}}(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & = 0 & \{s_0 > s_1\} \\
(6)\ \mathsf{sum}_2(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\mathsf{i},\underline{\mathbf{j}}) & = 4+\mathsf{sum}_3(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}') & \\
 & \quad +\mathsf{sum}_1(\mathsf{m},\mathsf{n},\underline{\mathbf{res}}',\mathsf{i}',\underline{\mathbf{j}''}) & \{j'=i, i'=i+1\} \\
 & & \\
(7)\ \mathsf{sum}_3(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) & = 3+\mathsf{sum}_3^{\mathsf{c}}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & \{s_0=j, s_1=n\} \\
(8)\ \mathsf{sum}_3^{\mathsf{c}}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & = \mathsf{sum}_4(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) & \{s_0 \leq s_1\} \\
(9)\ \mathsf{sum}_3^{\mathsf{c}}(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j},\boxed{\mathbf{s_0}},\boxed{\mathbf{s_1}}) & = 0 & \{s_0 > s_1\} \\
(10)\ \mathsf{sum}_4(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}) & = 9+\mathsf{sum}_3(\underline{\mathbf{m}},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\mathsf{j}') & \{j'=j+1\}
\end{array}
$$

Fig. 4. CES for the rule-based program of Fig. 3

the case where $i \leq n$ which continues to sum_2; and the second one for the $i > n$ which terminates the loop.

Step III: Generating a Cost Equations System

In the last step, size relations analysis is applied to the rule-based program and a cost equations system, which defines the cost of each rule as a function of its input arguments, is generated. The aim of the size analysis is to infer (linear) relations between the values (or sizes of data structures) of the different variables at different program points. For example, it infers that the value of i when calling sum_1 (in the rule sum_2) is greater than the input value of i by one. Using these size relations, for each rule in the corresponding rule-based program we generate an equation of the form

$$p(\bar{x}) = c + \sum_{i=1}^{k} p_i(\bar{x}_i), \quad \varphi$$

which defines the cost of the rule p in term of its input arguments $\bar{x}$ to be: (1) the number of bytecode instructions c in the rule; plus (2) the cost of all calls to other rules, namely $p_1(\bar{x}_1) \ldots, p_k(\bar{x}_k)$. The linear constraints φ (which is inferred by the size analysis) describe the size relations between the variables $\bar{x} \cup \bar{x}_i \cdots \cup \bar{x}_k$. We refer to the set of all generated equation as *cost equations system* (CES). The CES of the rule-based program of Fig. 3 is depicted in Fig. 4.

Let us consider, for example, the equations 3-6 which correspond to the outer-loop. Equation 3 defines the cost of $\mathsf{sum}_1(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j})$ to be the number of its bytecode instructions, namely 3, plus the cost of executing $\mathsf{sum}_1^{\mathsf{c}}(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1)$ where $s_0 = i$ and $s_1 = m$. Equations 4 and 5 define the cost of $\mathsf{sum}_1^{\mathsf{c}}(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1)$ to be like the cost of $\mathsf{sum}_2(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j})$ in case that $s_0 \leq s_1$, and 0 in case that $s_0 > s_1$. Equation 6 defines the cost of $\mathsf{sum}_2(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j})$ to be the number of its bytecode instructions, namely 4, plus the cost of $\mathsf{sum}_3(\mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}')$ (the inner-loop) and $\mathsf{sum}_1(\mathsf{m},\mathsf{n},\mathsf{res}',\mathsf{i}',\mathsf{j}'')$ where $j' = i$ (the initial value for j in the inner-loop) and $i' = i + 1$ (the outer-loop counter is increased).

Usually, the CES are used to obtain a closed form upper bounds on the cost, e.g., $sum(m, n) = O(m * n)$, and this is usually done by using standard computer algebra systems, such as Mathematica and Maple. The problem is that in its current

$$
\begin{array}{lll}
(1)\ \mathsf{sum}(\mathsf{m},\mathsf{n}) & = \mathsf{sum}_0(\mathsf{m},\mathsf{n}) & \{\} \\
(2)\ \mathsf{sum}_0(\mathsf{m},\mathsf{n}) & = 6 + \mathsf{sum}_1(\mathsf{m},\mathsf{n},\mathsf{i}') & \{i' = 1\} \\
 & & \\
(3)\ \mathsf{sum}_1(\mathsf{m},\mathsf{n},\mathsf{i}) & = 3 + \mathsf{sum}_1^c(\mathsf{m},\mathsf{n},\mathsf{i}) & \{\} \\
(4)\ \mathsf{sum}_1^c(\mathsf{m},\mathsf{n},\mathsf{i}) & = \mathsf{sum}_2(\mathsf{m},\mathsf{n},\mathsf{i}) & \{i \leq m\} \\
(5)\ \mathsf{sum}_1^c(\mathsf{m},\mathsf{n},\mathsf{i}) & = 0 & \{i > m\} \\
(6)\ \mathsf{sum}_2(\mathsf{m},\mathsf{n},\mathsf{i}) & = 4 + \mathsf{sum}_3(\mathsf{n},\mathsf{j}') + \mathsf{sum}_1(\mathsf{m},\mathsf{n},\mathsf{i}') & \{j' = i, i' = i + 1\} \\
 & & \\
(7)\ \mathsf{sum}_3(\mathsf{n},\mathsf{j}) & = 3 + \mathsf{sum}_3^c(\mathsf{n},\mathsf{j}) & \{\} \\
(8)\ \mathsf{sum}_3^c(\mathsf{n},\mathsf{j}) & = \mathsf{sum}_4(\mathsf{n},\mathsf{j}) & \{j \leq n\} \\
(9)\ \mathsf{sum}_3^c(\mathsf{n},\mathsf{j}) & = 0 & \{j > n\} \\
(10)\ \mathsf{sum}_4(\mathsf{n},\mathsf{j}) & = 9 + \mathsf{sum}_3(\mathsf{n},\mathsf{j}') & \{j' = j + 1\}
\end{array}
$$

Fig. 5. A Simplified version of the CES of Fig. 4

form, the CES are not even considered a valid input for such systems, and therefore they should be further simplified. The main problem in the generated CES is that some equations may contain variables that do not affect the cost of the corresponding rules. For example, in the CES of Fig. 4, the underlined variables are irrelevant to the cost and therefore should be eliminated; and moreover all *stack* variables (appear in frames) can be replaced by corresponding local variables. The next section describe how this can be automatically done.

3 Eliminating Irrelevant Variables in CES

As described in the previous section, we are interested in eliminating two kind of variables from the CES: (1) stack variables which can be replaced by corresponding local variables or constant values; and (2) variables that do not affect the cost of their corresponding rules. Eliminating the first kind of variables can be done by a simple dependencies analysis which is applied *locally* to the rules, and eliminating the second kind of variables can be done using program slicing.

Eliminating Stack Variables

Replacing stack variables by their corresponding local variables (or constants) can be done by tracking dependencies that are introduced by instructions that link between stack elements and local variables (or constants), such as $\mathsf{iload}(\mathsf{v},\mathsf{s_i})$ which pushes the variable v on the top stack s_i, $\mathsf{iconst}(1,\mathsf{s_i})$ which pushes the constants 1 on the top of the stack s_i, and $\mathsf{istore}(\mathsf{s_i},\mathsf{v})$ which assigns the variable v to the value of the top of the stack s_i. This is a simple process that can be done locally to the rules by keeping dependencies list, and it does not require any fixpoint computation. For example, in the rule sum_1 (Fig. 3), we can keep the dependencies $s_0 \mapsto i$ and $s_1 \mapsto m$ which are introduced by the first two bytecode instructions, and then using these dependencies to eliminates s_0 and s_1 are from sum_1 and sum_1^c. In practice, using this simple analysis all stack variables can be eliminated except some of those that correspond to return value of a method. In addition, eliminating the stack variables from the rule-based representation before applying the size analysis improves significantly its performance as the number of total variables is significantly reduced.

Eliminating Irrelevant Variables

The process of eliminating variables that do not affect the cost of their corresponding rules is based on the basic observation that the cost is affected only by variables that appear in guards of the different rules, since only those variables correspond (directly) to loop conditions, recursion base-case conditions, etc. Therefore variables that are guaranteed not to affect, directly or indirectly, any variable that appear in a guard are irrelevant to the cost and therefore can be safely removed. This can be done by computing a superset (approximation) for the set of variables that might affect guards' variables, which in turn can be formalized as a backward slicing problem where the slicing criterion includes *all guards and their variables*. This is a simple instance of backwards slicing, mainly due to the fact that the slicing criterion includes all conditions and their variables. Therefore, the slicing algorithm does not need to track implicit dependencies that stem from conditional assignments (e.g., assignment in the *then* or *else* branches) since, by the definition of the slicing criterion, all variables in guards will be included in the relevant set of variables. Standard backward slicing algorithms [9] can be adapted and applied directly to the rule-based representation or to the CES. For example, applying backward slicing on the rule based program of Fig.3 (after eliminating the stack variables) results in the following set of relevant variables for the different rules:

$\mathsf{sum} = \{m, n\}$	$\mathsf{sum}_1 = \{m, n, i\}$	$\mathsf{sum}_3 = \{n, j\}$
$\mathsf{sum}_0 = \{m, n\}$	$\mathsf{sum}_1^c = \{m, n, i\}$	$\mathsf{sum}_3^c = \{n, j\}$
	$\mathsf{sum}_2 = \{m, n, i\}$	$\mathsf{sum}_4 = \{n, j\}$

which can be used to simplify the CES in Fig. 4 to the one in Fig. 5.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *9th International Workshop on Termination, WST'07*, June 2007.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 67–83. Elsevier - North Holland, July 2007.
4. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.
5. E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

6. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.
7. K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
9. Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.