Removing Useless Variables in Cost Analysis of Java Bytecode

E. Albert P. Arenas
DSIC, Complutense University of Madrid
E-28040 Madrid, Spain
{elvira,puri}@sip.ucm.es

S. Genaim G. Puebla D. Zanardini
CLIP, Technical University of Madrid
E-28660 Boadilla del Monte, Madrid, Spain
{samir,german,damiano}@sip.ucm.es

ABSTRACT

Automatic cost analysis has interesting applications in the context of verification and certification of mobile code. For instance, the code receiver can use cost information in order to decide whether to reject mobile code which has too large cost requirements in terms of computing resources (in time and/or space) or billable events (SMSs sent, bandwidth required). Existing cost analyses for a variety of languages describe the resource consumption of programs by means of *Cost Equation Systems* (CESs), which are similar to, but more general than recurrence equations. CESs express the cost of a program in terms of the size of its input data. In a further step, a closed form (i.e., non-recursive) solution or upper bound can sometimes be found by using existing Computer Algebra Systems (CASs), such as Maple and Mathematica. In this work, we focus on cost analysis of Java bytecode, a language which is widely used in the context of mobile code and we study the problem of identifying variables which are *useless* in the sense that they do not affect the execution cost and therefore can be ignored by cost analysis. We identify two classes of useless variables and propose automatic analysis techniques to detect them. The first class corresponds to *stack variables* that can be replaced by program variables or constant values. The second class corresponds to variables whose value is *cost-irrelevant*, i.e., does not affect the cost of the program. We propose an algorithm, inspired in *static slicing* which safely identifies cost-irrelevant variables. The benefits of eliminating useless variables are two-fold: (1) cost analysis without useless variables can be more efficient and (2) resulting CESs are more likely to be solvable by existing CASs.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers, Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Symbolic execution*; F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; D.3.2 [**Programming Languages**]

General Terms

Complexity, Languages, Theory.

Keywords

Cost Analysis, Program Slicing, Information Flow,
Cost Equations Systems, Java Bytecode.

1. INTRODUCTION

Java bytecode [19] is a stack-based, low-level, object oriented language which is widely used in the context of mobile code, due to its security features and platform independence. Automatic cost analysis has interesting applications in the context of Java bytecode [3]. For instance, the code receiver may want to infer (or to check, in the spirit of Proof-Carrying Code [22]) 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) [6, 17, 12, 9], and to accept code meeting the established requirements. Moreover, 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 [13].

Several cost analyses exist for a wide variety of programming languages, including logic [21, 13, 14], functional [24, 23, 25, 16, 10] and imperative languages [3, 18, 10]. In general, given an input program, cost analysis infers a Cost Equation System (CES), which is a general form of describing the resource consumption of programs w.r.t. the cost model of interest. Different cost models can be used to capture different aspects of the computation, such as the number of bytecode instructions executed, [4], the memory (heap) consumption, [5], etc. Intuitively, CESs are systems of *recursive* equations which express the cost of a part of the program in terms of the cost of the parts which may follow it, according to the program structure. All kinds of iterations in the program are expressed in form of recursion in the corresponding CES. In addition, CESs include information about how the size of variables changes when the control moves between different parts of the program (e.g., the increase of a loop index). CESs are a generalization of recurrence equations in the sense that cost analysis often

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
Copyright 2008 ACM 978-1-59593-753-7/08/0003 ...$5.00.

infers results which are CESs but do not satisfy the syntactic requirements imposed by recurrence equations. CESs can sometimes be *solved* by inferring a *closed form* solution (or an *upper bound* of it), i.e., an expression without recurrences, by using *Computer Algebra Systems* (CASs) such as Mathematica and Maple.

Ideally, we are interested in obtaining CESs where only the program variables and arguments which affect the cost appear as arguments in the equations. The remaining ones can be ignored as far as cost analysis is concerned. The focus of this paper is on defining automatic techniques which can identify and remove from CESs generated by cost analysis of Java bytecode [3], variables which do not affect the cost, and are therefore *useless*. Importantly, removing useless arguments from CESs is crucial for the practical uptake of cost analysis for, at least, the following two reasons: (i) static (cost) analysis can be more efficient if we reduce the number of variables; and (ii) CESs become simpler, and more likely solvable with standard CASs. Essentially, we classify useless variables into two categories:

1. *Redundant stack variables.* A *stack variable* represents an operand stack location at a given program point. Stack variables are a component of our rule-based representation of programs (Sec. 3, Step II). They are created from the original program by means of a transformation step which eliminates the stack and uses additional variables instead. It is often the case that a stack variable is created by loading a program variable (or constant) on the stack, and it is immediately eliminated after storing the result of some computation (as in the typical *load*, *load*, *add*, *store* sequence). In this case, in the rule-based representation, it is safe to replace stack variables by the local variables or constants whose value was loaded on the corresponding stack location in the program. This can be done by first applying a *single static assignment* transformation to the corresponding sequence of bytecode instructions, to avoid name clashes, and then *unifying* (i.e., making identical) the local variable or constants with the corresponding stack variable. E.g., in $\mathsf{iload}(\mathsf{v}, \mathsf{s_i})$ we unify $\mathsf{s_i}$ with v, and in $\mathsf{iconst}(1, \mathsf{s_i})$ we unify $\mathsf{s_i}$ with 1.

2. *Cost-Irrelevant program variables.* The program variables which may have an impact on the cost of a program are those that may affect directly or indirectly the conditional statements (i.e., they can affect the control flow of the program), and those that may affect the values which are used as input to operations which do not have a constant cost. Calls to methods are a typical example of non-constant operations w.r.t. cost. Also bytecode instructions can be non-constant, as in the case of array manipulation (see Sec. 5). We refer to both kinds of variables as *cost-relevant*. The rest of variables are called *cost-irrelevant*, and can be safely removed. For instance, typically, *accumulating* variables which merely keep the temporary value of some result do not affect the control flow, and, if they are not used in operations with a non-constant cost, they are cost-irrelevant. We formalize the problem of computing a safe approximation of the set of cost-relevant arguments as a *backward slicing* [27] problem, where variables in reachable conditional statements and operations with non-constant cost are the information we want to preserve, i.e., they are used to build the *slicing criterion.*

While redundant stack variables only occur in stack-based programming languages (such as Java bytecode), their removal is beneficial in principle for any static analysis, not only cost analysis. Also, though cost-irrelevant variable elimination is particular to cost analysis (and also termination) it is of interest for cost analysis of any language, not only Java bytecode. Therefore, we claim that our contributions are both useful for other analyses (category 1) and applicable to other programming languages and paradigms (category 2).

2. JAVA BYTECODE

A (sequential) Java bytecode (JBC) program consists of a set of *class files*, one for each class. A class file contains information about its name, the class it extends, and the fields and methods it defines. Each method has a unique signature m from which we can obtain the class where the method is defined, the name of the method, and its type. The bytecode associated to m is a sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n\rangle$, where each b_i is a *bytecode instruction* and pc_i is its address. In addition, the method's local variables are denoted by $\langle l_0, \ldots, l_{n-1}\rangle$, where l_0 corresponds to the *this* reference (unlike Java, in Java bytecode, the *this* reference is explicit), and $l_1, \ldots, l_k$ with $k{<}n$ are the formal parameters of the method. Similarly, each field f has a unique signature from which we can obtain its name and the name of the class the field belongs to. The bytecode instructions we consider include:

$$\begin{array}{lcl} \mathit{bcInst} & ::= & \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ n \\ & \mid & \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{iinc}\ v\ n \mid \mathsf{imul} \mid \mathsf{idiv} \\ & \mid & \mathsf{if\_cond}\ pc \mid \mathsf{goto}\ pc \mid \mathsf{ireturn} \\ & \mid & \mathsf{new}\ c \mid \mathsf{invokevirtual}\ c.m \mid \mathsf{invokespecial}\ c.m \\ & \mid & \mathsf{getfield}\ c.f \mid \mathsf{putfield}\ c.f \end{array}$$

where c is a class, cond is a comparison condition on numbers (ne, le, icmpgt, etc.) or references (null, nonnull), v is a local variable, n is an integer and pc is an instruction address. The instructions in the first row move information from and to the stack. The second row includes arithmetic instructions. The third row contains conditional and unconditional jumps and the return from a method call. The fourth and fifth rows contain object-oriented instructions. All instructions in the first two rows are executed *sequentially*, i.e., one after the other (*non-branching*). The ones in the last three rows, except for new c, are *non-sequential* (or *branching*): execution may continue with an instruction which is not the next one.

3. COST ANALYSIS OF JAVA BYTECODE BY EXAMPLE

We briefly illustrate the cost analysis we refer to [3] by using the example depicted in Fig. 1. The Java program (on the top, provided just for clarity since the analysis is directly performed on the Java bytecode program) and its corresponding Java bytecode (on the bottom) define a method *sum* that, given the integer values n and m, computes the sum

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

Computing a *closed form* function which is an exact solution or an upper bound of the cost of *sum* (e.g., we may choose

```
class Sum {
  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
```

Figure 1: A Java Program and its corresponding Java bytecode

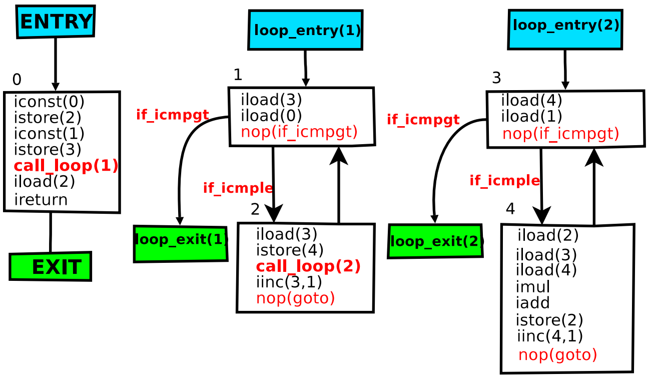

Figure 2: CFGs for the Java bytecode program in Fig. 1

a cost model where the cost is the number of bytecode instructions which may be executed) in terms of its input arguments consists of several steps which are explained below: (1) recovering the structure of the program by means 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 generating a CES from which we can obtain a closed form solution or upper bound using standard CASs.

Step I: Control Flow Graph

In the first step, each sequence of Java bytecode instructions (which corresponds to a method) is transformed into a corresponding set of CFGs by using techniques from compiler theory [1, 2]. This is done by splitting the instruction sequence into maximal sub-sequences of *non-branching* instructions, which form the basic blocks (nodes) of the initial graph. The basic blocks are connected by *guarded* edges which describe the possible transitions. Guards and edges are introduced by considering the last bytecode instruction of each block, and represent the condition (guard) for the control going from one block to another one. Finally, a *loop extraction* transformation is applied on the initial CFG in order to separate those sub-graphs corresponding to loops. This transformation has been well studied in the area of program decompilation [7]. It is crucial when the program contains *nested loops*, since it allows analyses which are *compositional*, in the sense that they can reason on just one loop at a time.

$$\mathsf{sum}(\langle \mathsf{m},\mathsf{n}\rangle,\langle \mathsf{r}\rangle) := \mathsf{init\_local\_vars}(\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle), \mathsf{sum}_0(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{r}\rangle).$$

$$\mathsf{sum}_0(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{r}\rangle) := \mathsf{iconst}(0,\mathsf{s}_0), \mathsf{istore}(\mathsf{s}_0,\mathsf{res}), \mathsf{iconst}(1,\mathsf{s}_0), \mathsf{istore}(\mathsf{s}_0,\mathsf{i}), \mathsf{sum}_1(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle), \mathsf{iload}(\mathsf{res},\mathsf{s}_0), \mathsf{ireturn}(\mathsf{s}_0,\mathsf{r}).$$

$$\mathsf{sum}_1(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{i},\mathsf{s}_0), \mathsf{iload}(\mathsf{m},\mathsf{s}_1), \mathsf{nop}(\mathsf{if\_icmpgt}(\mathsf{s}_0,\mathsf{s}_1)), \mathsf{sum}_1^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).$$

$$\mathsf{sum}_1^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s}_0,\mathsf{s}_1)), \mathsf{sum}_2(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).$$

$$\mathsf{sum}_1^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s}_0,\mathsf{s}_1)).$$

$$\mathsf{sum}_2(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{i},\mathsf{s}_0), \mathsf{istore}(\mathsf{s}_0,\mathsf{j}), \mathsf{sum}_3(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle), \mathsf{iinc}(\mathsf{i},1), \mathsf{nop}(\mathsf{goto}), \mathsf{sum}_1(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{i},\mathsf{j}\rangle).$$

$$\mathsf{sum}_3(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{j},\mathsf{s}_0), \mathsf{iload}(\mathsf{n},\mathsf{s}_1), \mathsf{nop}(\mathsf{if\_icmpgt}(\mathsf{s}_0,\mathsf{s}_1)), \mathsf{sum}_3^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{j}\rangle).$$

$$\mathsf{sum}_3^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s}_0,\mathsf{s}_1)), \mathsf{sum}_4(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle).$$

$$\mathsf{sum}_3^{\mathsf{c}}(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j},\mathsf{s}_0,\mathsf{s}_1\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s}_0,\mathsf{s}_1)).$$

$$\mathsf{sum}_4(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle) := \mathsf{iload}(\mathsf{res},\mathsf{s}_0), \mathsf{iload}(\mathsf{i},\mathsf{s}_1), \mathsf{iload}(\mathsf{j},\mathsf{s}_2), \mathsf{imul}(\mathsf{s}_1,\mathsf{s}_2,\mathsf{s}_1), \mathsf{iadd}(\mathsf{s}_1,\mathsf{s}_0,\mathsf{s}_0), \mathsf{istore}(\mathsf{s}_0,\mathsf{res}), \mathsf{iinc}(\mathsf{j},1), \mathsf{nop}(\mathsf{goto}), \mathsf{sum}_3(\langle \mathsf{m},\mathsf{n},\mathsf{res},\mathsf{i},\mathsf{j}\rangle,\langle \mathsf{res},\mathsf{j}\rangle).$$

Figure 3: The Intermediate Representation for the CFGs of Fig. 2

The CFGs of the *sum* method are depicted in Fig. 2. In block 0, the variables i and res are initialized (first four instructions), and the control is transferred to the middle CFG (using the instruction call_loop), which corresponds to the outer-loop. Upon return from that loop, the method returns the value res (last two instructions). In block 1 (the entry of the outer-loop), the values of i and m are compared. If $i \le m$ 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 control goes back to the caller). Note that the corresponding edges are annotated by conditions (guards) corresponding to $i \le m$ and $i > m$. Block 2 corresponds to the body of the outer-loop: it initializes j, transfers the control 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.

Step II: Rule-Based Representation

In the second step, CFGs are represented procedurally by means of rule-based programs. A *rule-based program* defines a set of *procedures*, each of them defined by one or more rules. We use $\bar{x}$ to denote a sequence of variables $\langle x_1, \ldots, x_n\rangle$. Each rule takes the form $head(\bar{x}, \bar{y})$:=[$guard$], $instr$,[$cont$] where (1) $head$ is a unique identifier for the procedure the rule belongs to; (2) $\bar{x}$ and $\bar{y}$ respectively indicate the sequences of input and output arguments; (3) $guard$ takes the form guard(ϕ), where ϕ is a Boolean condition on the variables in $\bar{x}$; (4) $instr$ is a sequence of bytecode instructions (where all input and output arguments to the instructions, including the local variables and stack elements they work on, are made explicit) and calls to other rules; and (5) $cont$ indicates a call to another procedure which represents the continuation of this procedure, if it exists. We use [_] to denote that an element is optional.

The most relevant issue in our study is related to the variables which become the arguments of the rules. Regarding the input arguments, $\bar{x}$ should include the local variables for the method, and the stack elements at the beginning of the block. As for the output arguments, $\bar{y}$ usually contains only the return value of the method, denoted by r. Moreover, in the case of rules which correspond to loops, output arguments also include the variables which are modified during the execution of the loop.

The rule-based program(s) depicted in Fig. 3 correspond respectively to the CFGs in Fig. 2. The rule *sum* corresponds to the method entry. It takes the input local variables m and n, and returns the output variable r. It first calls $\mathsf{init\_local\_vars}(\langle \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle)$ which initializes the local variables to the default value of their type as stipulated by Java, and then calls the rule sum_0 (corresponding to block 0). The rule sum_0 takes all local variables (including the formal parameters) as input and returns r as output. The instructions $\mathsf{iconst}(0, \mathsf{s}_0)$ 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, $\mathsf{iconst}(1, \mathsf{s}_0)$ and $\mathsf{istore}(\mathsf{s}_0, \mathsf{i})$ initialize i to one. Afterwards, 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 r to the return value of *sum*. Note that, when calling the outer-loop sum_1, the list of output arguments only includes 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 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 accounts for to the case where $i{\leq}m$ ($\mathsf{guard}(\mathsf{if\_icmple}(\mathsf{s}_0, \mathsf{s}_1))$), which continues to sum_2. The second one accounts for $i{>}m$ ($\mathsf{guard}(\mathsf{if\_icmpgt}(\mathsf{s}_0, \mathsf{s}_1))$), which terminates the loop.

Instructions labeled with nop are not considered when computing the size relations in the following step, since they have been replaced either by guards or by calls to some other rule. For instance, nop(if_icmpgt) in sum_1 is replaced by the corresponding guards in sum_1^c. Similarly, nop(goto) in sum_2 is replaced by the call to sum_1.

Step III: Generating a Cost Equations System

In the last step, size relations analysis is applied to the rule-based program and a CES, 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 terms of its input arguments $\bar{x}$ to be: (1) c is the direct cost of the bytecode instructions which appear 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 are inferred by the size analysis) describe the size relations between the variables $\bar{x}\cup\bar{x}_1\cdots\cup\bar{x}_k$.

$$
\begin{array}{rl}
(1) & \mathsf{sum}(\mathsf{m},\mathsf{n}) = \mathsf{sum}_0(\mathsf{m},\mathsf{n},\underline{\mathbf{res}},\underline{\mathbf{i}},\underline{\mathbf{j}}) \\
 & \qquad\qquad \{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}}) \\
 & \qquad\qquad \{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}}) \\
 & \qquad\qquad \{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}}) \\
 & \qquad\qquad \{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 \\
 & \qquad\qquad \{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}') \\
 & \qquad + \mathsf{sum}_1(\mathsf{m},\mathsf{n},\underline{\mathbf{res'}},\mathsf{i}',\underline{\mathbf{j''}}) \\
 & \qquad\qquad \{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}}) \\
 & \qquad\qquad \{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}) \\
 & \qquad\qquad \{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 \\
 & \qquad\qquad \{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}') \\
 & \qquad\qquad \{j' = j + 1\}
\end{array}
$$

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

We refer to the set of all generated equations as CES.

Assuming that we are interested in knowing the number of bytecode instructions executed, from the rule-based program in Fig. 3 we obtain the CES depicted in Fig. 4. Equation 1 corresponds to the entry cost equation and its size relations reflect the initialization of the local variables (init_local_vars in the recursive representation). Let us consider now, 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 $\mathsf{sum}_1^{\mathsf{c}}(\mathsf{m}, n, res, i, \mathsf{j}, \mathsf{s}_0, \mathsf{s}_1)$ to be equal to the cost of $\mathsf{sum}_2(\mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j})$ if $s_0{\leq}s_1$, and 0 if $s_0{>}s_1$. 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).

Automatic cost analysis usually aims at providing a closed form upper bound from the CES, e.g., $sum(m, n){=}O(m{*}n)$. Sometimes, this can be done by using standard CASs, such as Mathematica and Maple. A problem that most automatic cost analyzers face is that, without a further processing, the generated CES are not even considered as a valid input for such systems. This is often the case when equations in the CES contain *irrelevant* variables which do not affect the cost of the corresponding rules. As a consequence, human interaction is usually required to remove them before giving the CES to the CAS. The problem can be often alleviated by further simplifying the equations by automatically removing such irrelevant variables. As we will explain in the next sections, in the CES of Fig. 4, the underlined variables are irrelevant to the cost and therefore can be safely eliminated. Furthermore, all *stack* variables, which appear in frames, are redundant and can be replaced by the corresponding local variables. In the next two sections, we provide automatic techniques to identify and eliminate both classes of irrelevant variables in the context of cost analysis of Java bytecode.

4. REMOVING REDUNDANT STACK VARIABLES

Stack variables can often be removed from the rule-base representation of a program by replacing their occurrences with local variables or constants. Our method to remove redundant stack variables is based on transforming the rule to be in *static single assignment* form and then *unifying* stack variables to local variables (or constants) in the statements that relate them. This is a simple process that is done locally to the rules. We sketch the three steps which are performed by our system in each corresponding subsection.

4.1 Static Single Assignment

In the first step, we perform a Static Single Assignment (SSA) transformation on the bytecode instructions of the rule-based representation. This is necessary since we want to apply unification to its variables (and they cannot be assigned to more than one value). It should be noted that this step is also needed –regardless whether we perform useless variable elimination– for the size analysis because it infers input-output denotations for each program point. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be transformed to $\mathsf{iadd}(s_0, s_1, s'_0)$ where s'_0 refers to the value of s_0 after executing the instruction. To implement the SSA transformation, we maintain a mapping ρ, for each rule, of variable names (as they appear in the rule) to new variable names (constraint variables). Such a mapping is referred to as a *renaming*. We let $\rho[x \mapsto y]$ denote the modification of the renaming ρ such that it maps x to the new variable y. We write $\rho[\bar{x} \mapsto \bar{y}]$ to denote the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

For each rule $p(\bar{x}, \bar{y}){:=}b_1, \ldots, b_n$, the SSA transformation generates a new rule $p(\bar{x}, \rho_{n+1}(\overline{y})){:=}b'_1, \ldots, b'_n$ by translating each b_i to b'_i as illustrated in the following table for a few bytecode instructions.

i$-$*th element*	*SSA*	ρ_{i+1}
$\mathsf{iload}(v, s_j)$	$\mathsf{iload}(\rho_i(v), s')$	$\rho_i[s_j \mapsto s']$
$\mathsf{istore}(s_j, v)$	$\mathsf{istore}(\rho_i(s_j), v')$	$\rho_i[v \mapsto v']$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$\mathsf{iadd}(\rho_i(s_j), \rho_i(s_{j+1}), s')$	$\rho_i[s_j \mapsto s']$
$q(\overline{x}, \overline{y})$	$q(\rho_i(\overline{x}), \overline{y}')$	$\rho_i[\overline{y} \mapsto \overline{y}']$
$\mathsf{guard}(\phi)$	$\mathsf{guard}(\rho_i(\phi))$	ρ_i

where (1) ρ_1 is the identity renaming; (2) ρ_i $(2{\leq}i{\leq}n+1)$ is the mapping available before traversing b_i; (3) ρ_{i+1} is the result of updating ρ_i; and (4) $\overline{y}'$, s' and v' are fresh variables. As an example, we show in Fig. 5 the SSA transformation for rules *sum₀* and *sum₁* together with the associated renaming for each bytecode.

4.2 Propagating Dependencies by Unification

After applying the SSA transformation, we can *unify* the stack elements, local variables and constants that occur as arguments of instructions like iload, iconst, istore and ireturn. These unifications will automatically reduce the number of (distinct) variables which occur in the rule. In addition, the corresponding instruction can be removed from the program as its effect is already accounted for in the unifications. This can be implemented using standard unification as, for instance, done in logic programming [20]. In our example, the rules sum_0 and sum_1 in Fig. 5 allow us to apply respectively the following sets of unifications:

$$sum_0 : \quad \{0{=}s'_0, s'_0{=}res', 1{=}s''_0, s''_0{=}i, res''{=}s'''_0, s'''_0{=}r\}$$
$$sum_1 : \quad \{i{=}s'_0, m{=}s'_1\}$$

Removing the corresponding instructions[1] (after applying the unifications) results in the following simplified rule:

$$\begin{array}{l} \mathsf{sum}_0(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{r}\rangle) := \\ \quad \mathsf{sum}_1(\langle \mathsf{m}, \mathsf{n}, 0, 1, \mathsf{j}\rangle, \langle \mathsf{r}, \mathsf{i}'', \mathsf{j}'\rangle). \\ \mathsf{sum}_1(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle) := \\ \quad \mathsf{sum}^{\mathsf{c}}_1(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}, \mathsf{i}, \mathsf{m}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle). \end{array}$$

A main advantage of implementing the dependency tracking by relying on unification is that we implicitly have *backwards* propagation of dependencies.

4.3 Argument Filtering

In addition to removing the bytecode instructions that manipulate stack elements, after unifying stack variables to local variables and constants, we can often filter out some arguments from rules. In our example for the rule sum_1, after the unification of s'_0 and s'_1 to the local variables, respectively, i and m as described in Section 4.2, calls to the rule sum^c_1 become of the form $\mathsf{sum}^{\mathsf{c}}_1(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}, \underline{\mathsf{i}}, \underline{\mathsf{m}}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle)$. We can observe that the same local variables i and m are unnecessarily passed twice in all calls to sum^c_1. Therefore, in the last step, we remove repeated arguments and constant arguments from rules as long as *all* calls to such rules are performed with the same repeated variables. Therefore, in our example, without the irrelevant stack variables, the head of the rule becomes $\mathsf{sum}^{\mathsf{c}}_1(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{res}', \mathsf{i}', \mathsf{j}'\rangle)$. Similarly, we can reason that the head of the rule sum^c_3 can be $\mathsf{sum}^{\mathsf{c}}_3(\langle \mathsf{m}, \mathsf{n}, \mathsf{res}, \mathsf{i}, \mathsf{j}\rangle, \langle \mathsf{res}', \mathsf{j}'\rangle)$.

It could happen that there exists one (or several) call which does not have the same repeated arguments. In such case, the argument filtering process described above cannot be performed. It would be possible, though, to filter those arguments which are actually repeated in all possible call patterns to the corresponding rule. In practice, by using this simple analysis described in the three phases above, all stack variables can often be eliminated, except some of them that correspond to the return value of methods. Clearly, eliminating the stack variables from the rule-based representation can improve the performance of any analysis of Java bytecode, as the number of total variables is significantly reduced. In addition, it can also be beneficial for obtaining a clearer source code in decompilation of bytecode.

5. ELIMINATING COST-IRRELEVANT PROGRAM VARIABLES

This section describes a technique, based on program slicing and dependence calculus, whose purpose is to remove information which is not needed by cost analysis. Program slicing [28, 27] has been usually applied to source code, since its main interest is to help humans in debugging, maintaining and understanding software. This is also the case of Java [8, ?], where little or no attention has been paid to slicing bytecode presumably for this reason. In this sense, our slicing algorithm is the first one developed at the level of bytecode. However, our approach does not properly belong to standard slicing, since the focus is on removing variables instead of statements. Also, the executability of the slice is not an issue here.

The process of eliminating variables which are irrelevant for computing the cost is based on the basic observation that the cost is affected by (1) variables appearing in *guards*, since

[1] Even if these instructions have no effect in size analysis, depending on the cost model they have to be taken into account to generate the cost equation system.

```
$\mathsf{sum}_0(\langle \mathsf{m, n, res, i, j}\rangle, \langle \mathsf{r}\rangle) :=$
   $\mathsf{iconst}(0, \mathsf{s}'_0),$                                            $\%\rho_2{=}\rho_1[s_0 \mapsto s'_0]$
   $\mathsf{istore}(\mathsf{s}'_0, \mathsf{res}'),$                                $\%\rho_3{=}\rho_2[res \mapsto res']$
   $\mathsf{iconst}(1, \mathsf{s}''_0),$                                           $\%\rho_4{=}\rho_3[s_0 \mapsto s''_0]$
   $\mathsf{istore}(\mathsf{s}''_0, \mathsf{i}'),$                                 $\%\rho_5{=}\rho_4[i \mapsto i']$
   $\mathsf{sum}_1(\langle \mathsf{m, n, res', i', j}\rangle, \langle \mathsf{res'', i'', j'}\rangle),$   $\%\rho_6{=}\rho_5[\langle res, i, j\rangle \mapsto \langle res'', i'', j'\rangle]$
   $\mathsf{iload}(\mathsf{res}'', \mathsf{s}'''_0),$                              $\%\rho_7{=}\rho_6[s_0 \mapsto s'''_0]$
   $\mathsf{ireturn}(\mathsf{s}'''_0, \mathsf{r}).$                                $\%\rho_8{=}\rho_7$
```

```
$\mathsf{sum}_1(\langle \mathsf{m, n, res, i, j}\rangle, \langle \mathsf{res', i', j'}\rangle) :=$
   $\mathsf{iload}(\mathsf{i}, \mathsf{s}'_0),$                                    $\%\rho_2{=}\rho_1[s_0 \mapsto s'_0]$
   $\mathsf{iload}(\mathsf{m}, \mathsf{s}'_1),$                                    $\%\rho_3{=}\rho_2[s_1 \mapsto s'_1]$
   $\mathsf{nop}(\mathsf{if\_icmpgt}(\mathsf{s}'_0, \mathsf{s}'_1)),$              $\%\rho_4{=}\rho_3$
   $\mathsf{sum}^c_1(\langle \mathsf{m, n, res, i, j, s'_0, s'_1}\rangle, \langle \mathsf{res', i', j'}\rangle).$   $\%\rho_5{=}\rho_4[\langle res, i, j\rangle$
                                                                                     $\mapsto \langle res', i', j'\rangle]$
```

Figure 5: The SSA transformations for some rules in Fig. 3

those are the variables corresponding to loop conditions, recursion base-case conditions, etc.; and (2) arguments of bytecode instructions which are required by the cost model in order to compute its corresponding cost, e.g., when creating an array the size of the array is important if we count memory consumption. Therefore, variables which are guaranteed not to affect, directly or indirectly, any variable of the above two categories can be safely removed from the analysis. This can be done by computing a superset (i.e., a safe approximation) of the set of the variables which might affect variables of the above two categories. Thus, the problem can be formalized as a (static) *backward slicing* problem, where the slicing criterion is all variables of the above two categories. Due to this choice of criterion, the slicing algorithm does not need to track *implicit dependencies* (e.g., an assignment in the branches of a conditional, where an implicit dependence exists between the guard variables and the modified variables), since all variables in guards will be included in the relevant set of variables. Standard backward slicing algorithms [27] can be adapted and applied directly to the rule-based representation. Algorithm 1 is a slicing algorithm that computes a set of relevant input variables for each rule p. The algorithm returns for each rule p a set slice(p) of relevant input arguments. It has a fixpoint nature with an abstract interpretation flavor [11], where at each iteration the set slice(p) is refined to include more variables. As we discuss below, our algorithm assumes information flow [15] and sharing information [26] is precomputed and available at slicing time. The procedure slicing is the fixpoint driver, and the procedure slice_procedure is the one applied iteratively by the driver until a fixpoint is reached. We start by explaining slice_procedure.

Algorithm 1 A naïve algorithm for backward slicing of the rule-based representation

```
 1: procedure slicing
 2:     for all procedure $p$ defined in $Program$ do
 3:         slice($p$) = $\emptyset$    // initialization
 4:         last_invoke($p$) = $\emptyset$
 5:         add_to_queue($\langle p, \emptyset\rangle, \mathcal{Q}$)    // initial calls
 6:     while $\langle p, \mathcal{P}\rangle$ = get_from_queue($\mathcal{Q}$) do    // non-empty queue
 7:         slice_procedure($p, \mathcal{P}$)
 8: procedure slice_procedure($p$,$\mathcal{P}$)
 9:     for all $p(\bar{x}, \bar{y}) := G, b_1, \ldots, b_n \in Program$ do
10:         $\mathcal{V} =_{sh}$ pos_to_vars($\mathcal{P}, \bar{y}$)    // converting from positions
11:         for i=n to 1 do
12:             if $b_i$ takes the form $q(\bar{x}_i, \bar{y}_i)$ then    // proc. call
13:                 $\mathcal{V}' = \{w \mid w \in \bar{x}_i, z \in \mathcal{V} \cap \bar{y}_i,$
        $w$ might affect $z\}$
14:                 $\mathcal{V} =_{sh} (\mathcal{V} \setminus \bar{y}_i) \cup$ pos_to_vars(slice($q$)$, \bar{x}_i$)$\cup \mathcal{V}'$
15:                 $\mathcal{P}_0$ = vars_to_pos($\mathcal{V} \cap \bar{y}_i, \bar{y}_i$)
16:                 if $\mathcal{P}_0 \not\subseteq$ last_invoke($q$) then    // re-analyze $q$
17:                     last_invoke($q$) = last_invoke($q$) $\cup \mathcal{P}_0$
18:                     add_to_queue($\langle q$, last_invoke($q$)$\rangle, \mathcal{Q}$)
19:             else    // $b_i$ is a bytecode
20:                 if $\mathcal{V} \cap$ output_vars($b_i$) $\neq \emptyset$ then
21:                     $\mathcal{V}'$ = input_vars($b_i$)
22:                 else
23:                     $\mathcal{V}' = \emptyset$
24:                 $\mathcal{V} =_{sh} (\mathcal{V} \setminus$ output_vars($b_i$)$) \cup \mathcal{V}' \cup$ cm_vars($b_i$)
25:         $\mathcal{V} =_{sh} \mathcal{V} \cup vars(G)$
26:         $\mathcal{P}$ = vars_to_pos($\mathcal{V} \cap \bar{x}, \bar{x}$)
27:         if $\mathcal{P} \not\subseteq$ slice($p$) then
28:             slice($p$) = slice($p$) $\cup \mathcal{P}$
29:             for all $q$ which call $p$ do
30:                 add_to_queue($\langle q$, last_invoke($q$)$\rangle, \mathcal{Q}$)
```

Assume, for a given rule $p(\bar{x}, \bar{y}) := G, b_1, \ldots, b_n$, that we already know that a subset of the output variables $\mathcal{V} \subseteq \bar{y}$ is required for the cost of some other rules, for example, it can be because these output variables affect guards of other rules. The aim is to compute a subset of the input variables $\bar{x}$ that might affect (1) the given subset of output variables $\mathcal{V}$; (2) the guard of p; (3) variables that are relevant to the cost of other rules that are called from p; and (4) arguments of bytecodes that are required by the cost model. We do this by going backwards from b_n to b_1 (lines 11-24) such that at each step the set $\mathcal{V}$ is refined by using some dependency information that we obtain from the corresponding b_i. We distinguish two cases depending on whether b_i is a bytecode or a call to another rule:

- If $b_i = q(\bar{x}_i, \bar{y}_i)$ (line 12) then we first compute a set of variables $\mathcal{V}' \subseteq \bar{x}_i$ (line 13) that might affect the value of any variable in $\mathcal{V} \cap \bar{y}_i$ (we assume this information is available using information flow analysis [15]). Then (line 14) we refine $\mathcal{V}$ by removing all variables that are in $\mathcal{V} \cap \bar{y}_i$ and adding the relevant variables for q (namely slice(q)) and $\mathcal{V}'$. Afterwards (lines 15-18), q is scheduled for re-analyses if the set of output variables of interest $\mathcal{V} \cap \bar{y}_i$ is not included in the one with respect to which it was analyzed the last time. Note that the role of pos_to_vars and vars_to_pos is to convert from variable positions to their names and vice versa since it is sometimes convenient to use the names and sometimes the positions.

- If b_i is a bytecode (line 19) then if any of the output variables of b_i is included in $\mathcal{V}$ we will include its set of input variables in $\mathcal{V}$, this is reflected by computing $\mathcal{V}'$ in lines 20-23. Then $\mathcal{V}$ is refined (line 24) by removing the output variables of b_i and adding $\mathcal{V}'$ and those which are required by the cost model, namely

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

Figure 6: A Simplified version of the CES of Fig. 4

$\mathsf{cm\_vars}(b_i)$.

Once the backwards propagation is done, we add the guard variables to $\mathcal{V}$ (line 25) and project $\mathcal{V}$ on the head variables (line 26). At the end, if the computed set of relevant arguments is not included in the previous one (line 27), then the new set is stored (line 28), and each rule q that calls p is scheduled for re-analyses (lines 29-30). Note in particular the use of $=_{sh}$ which means that the computed set is closed under sharing information, i.e., if a variable v is in the set, then all variables that might share with v a data structure on the heap are also included in the set. This information can be computed using sharing analysis [26]. The main procedure slicing performs the fixpoint by means of a queue $\mathcal{Q}$. It initializes the values of each slice(p) to empty set, and schedules each p to be analyzed with respect to an empty set of output variables.

As an 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:

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

which can be used to simplify the CES in Fig. 4 to the one in Fig. 6.

Note that an interesting feature of our slicing problem, is that any set of relevant variables can be safely used to refine the CES, even if it is not a superset (i.e., safe approximation) of the actual relevant variables. This is due to the fact that removing arguments from the CES can only increase the cost, and therefore we are approximating the CES in the safe direction since we are interested in computing upper bounds. Indeed, in our current implementation we ignore the information flow (line 13) and the sharing information in $=_{sh}$ which in turn might result in unsafe approximation of the relevant variables, but in practice we still get very useful sets of relevant variables.

6. EXPERIMENTS AND DISCUSSION

We have incorporated the analysis techniques for the elimination of useless stack and program variables as described in the paper in a cost analyzer for JBC programs.

Table 1 shows the effect of eliminating redundant stack and irrelevant local variables on a series of benchmarks for which our system can infer automatically CESs. We include classical recursive programs such as *Factorial*, *Hanoi*, *Fibonacci*, *MergeSort* or *QuickSort*. Iterative programs *DivByTwo* and *Concat* contain a single loop, while *Sum*, *MatMult* and *BubbleSort* are implemented with nested loops. We also include programs written in object-oriented style, like *Polynomial* or *Incr*, which contain object creation, virtual invocation, etc. The remaining benchmarks are implemented using data structures: *ArrayReverse*, *MatMultVector*, *Search* use arrays and *BubbleSort* and *DoSum* traverse linked lists.

Benchmark	**#R**	**#V**	**SVE**	**Slic**	**Bth**	**Rt**
Polynomial	19	89	61	64	41	2.17
BinarySearch	15	115	78	83	52	2.21
DivByTwo	12	43	31	22	15	2.87
Indexes	29	232	169	155	105	2.21
EvenDigits	19	93	65	52	34	2.74
Factorial	11	31	21	20	13	2.38
ArrayReverse	12	65	46	31	17	3.82
Concat	15	108	71	64	34	3.18
Incr	40	177	136	104	75	2.36
ListReverse	12	63	46	32	20	3.15
Power	11	38	27	20	13	2.92
Search	26	137	105	73	52	2.63
MergeSort	26	212	156	134	87	2.44
QuickSort	26	226	166	159	106	2.13
Sum	15	101	75	55	37	2.73
ListInter	38	247	182	164	113	2.19
SelectSort	18	135	101	90	60	2.25
OrdSort	18	109	79	69	46	2.37
DoSum	29	155	111	87	57	2.72
BubbleSort	18	143	108	95	64	2.23
MatMult	18	191	144	94	56	3.41
Hanoi	12	49	35	13	7	7.00
Fibonacci	11	37	23	26	15	2.47

Table 1: Effect of removing useless variables

Column **#R** shows the number of rules in the rule-based representation of each program. Note that, for the case of our running example *Sum*, column **#V** contains 15 instead of 10, as shown in Fig. 3. This is because in Fig. 3 we have omitted some calls to continuations in order to simplify the presentation. Column **#V** shows the total number of different variables (including both stack and local variables) in the rule-based representation after applying SSA. We show this figure because, for efficiency, our prototype always generates the rule-based representation with SSA, since SSA is required in order to perform size analysis. The next three columns, namely **SVE**, **Slic**, **Bth**, show the number of remaining variables after performing different removal techniques. More precisely, in column **SVE**, we have performed stack variable elimination but not removal of cost-irrelevant variables (slicing). In the column **Slic**, slicing is done but not stack variable elimination. It should be noted that slicing can sometimes remove redundant stack variables, which explains the significantly high variable removal in this case as well. Finally, **Bth** combines the result of applying simultaneously both techniques. Clearly, the effect of applying both techniques is not, in terms of the number of removed variables, the sum of the two analyses, since some variables can be removed by both **SVE** and **Slic**. The last column **Rt** shows **#V / Bth**. It can be observed that the number of variables is importantly reduced. The overall reduction ranges from 2.13 in the case of *Quicksort*, to 7 in the case of *Hanoi*. This is explained in part by the fact that local variables are always pushed on the stack by means of a load instruction such that the corresponding stack variable can be removed by unifying it with the local variable, as explained in Sec. 4.

Finally, although not experimentally evaluated yet, reducing the number of variables will improve analysis performance. This is especially important for the case of cost analysis, where size analysis is a costly phase which has to track calls-to size relations between variables. Removing irrelevant variables from the recursive representation will thus bring us an efficiency improvement in size analysis and consequently in the overall analysis time. Moreover, the elimination of redundant stack variables is potentially beneficial for any analysis of Java bytecode.

7. 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 Regional Government of Madrid under the S-0505/TIC/0407 *PROMESAS* project.

8. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.

[4] 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.

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

[6] 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.

[7] F. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[8] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *ACM PEPM*, 2003.

[9] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS, 2005.

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

[11] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.

[12] K. Crary and S. Weirich. Resource bound certification. In *POPL*. ACM, 2000.

[13] S. K. Debray and N. W. Lin. Cost analysis of logic programs. *TOPLAS*, 15(5), 1993.

[14] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *ILPS'97*. MIT Press, 1997.

[15] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. of VMCAI*, LNCS. Springer-Verlag, 2005.

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

[17] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *Proc. of PPDP'05*. ACM Press, July 2005.

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

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

[20] J. W. Lloyd. *Logic Programming.* Springer-Verlag, 1987.

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

[22] G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

[23] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. TR. CS-90-1, Dept. of C.S., Univ. of Sheffield, UK, 1990.

[24] M. Rosendahl. Automatic Complexity Analysis. In *FPCA*. ACM, 1989.

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

[26] S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.

[27] F. Tip. A Survey of Program Slicing Techniques. *J. of Prog. Lang.*, 3, 1995.

[28] M. Weiser. Program slicing. In *Proc. Int. Conf. on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.