A Generic Framework for the Cost Analysis of Java Bytecode*

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

Samir Genaim, Germán Puebla, Damiano Zanardini
CLIP, Technical University of Madrid
E-28660 Boadilla del Monte
{samir,german,damiano}@clip.dia.fi.upm.es

Abstract

Cost analysis of Java bytecode is complicated by its unstructured control flow, the use of an operand stack and its object-oriented programming features (like dynamic dispatching). This paper addresses these problems and develops a generic framework for the automatic cost analysis of sequential Java bytecode. Our method generates *cost relations* which define at compile-time the cost of programs as a function of their input data size. To the best of our knowledge, this is the first approach to the automatic cost analysis of Java bytecode.

1 Introduction

Cost analysis has been intensively studied in the context of declarative (see, e.g., [21, 20, 22, 16, 8] for functional programming and [14, 15] for logic programming) and *high-level* imperative programming languages (mainly focused on the estimation of worst case execution times and the design of cost models [27]). Traditionally, cost analysis has been formulated at the source level. However, there are situations where we do not have access to the source code, but only to compiled code. An example of this is *mobile code*, where the *code consumer* receives code to be executed. In this context, Java bytecode [17] is widely used, mainly due to its security features and the fact that it is *platform-independent*. Automatic cost analysis has interesting applications in this context. 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, and to accept code which meets the established requirements [12, 5, 6]. In fact, this is the main motivation for the *Mobile Resource Guarantees* (MRG) research project [6], which establishes a *Proof-Carrying Code* [19] framework for guaranteeing resource consumption. Furthermore, the *Mobility, Ubiquity and Security* (MOBIUS) research project [7], also considers resource consumption as one of the central properties of interest for proof-carrying code. Also, in parallel systems, knowledge about the cost of different procedures can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

The aim of this work is to develop an automatic approach to the cost analysis of Java bytecode which statically generates *cost relations*. These relations define the cost of a program as a function of its input data size. This approach was proposed by Debray and Lin [14] for logic programs, and by Rabhi and Manson [20] for functional programs. In these approaches, cost functions are expressed by means of *recurrence equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. A low-level object-oriented language such as Java bytecode introduces novel challenges, mainly due to: 1) its unstructured

*An extended version of this paper has appeared in [2]. This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* integrated project, and by the Spanish projects TIN2005-09207-C03 (*MERIT*) and CAM S-0505/TIC/0407 (*PROMESAS*). S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

control flow, e.g., the use of goto statements rather than recursive structures; 2) its object-oriented features, like virtual method invocation, which may influence the cost; and 3) its stack-based model, in which stack cells store intermediate values. This paper addresses these difficulties and develops a generic framework for the automatic cost analysis of Java bytecode programs. The process takes as input the bytecode corresponding to a method and yields a cost relation after performing these steps:

1. The input bytecode is first transformed into a *control flow graph* (CFG). This allows making the unstructured control flow of the bytecode explicit (challenge 1 above).

2. The CFG is then represented as a set of rules by using an *intermediate recursive representation* in which we *flatten* the local stack by converting its contents into a series of additional local variables (challenge 3).

3. In the third step, we infer *size relations* among the input variables for all calls in the rules by means of static analysis. These size relations are constraints on the possible values of variables (for integers) and constraints on the length of the longest reachable path (for references).

4. The fourth phase provides, for each rule of the recursive representation, a safe approximation of the set of input arguments which are "relevant" to the cost. This is performed using a simple static analysis.

5. From the recursive representation, its relevant arguments, and the size relations, the fifth step automatically yields as output the *cost relation* which expresses the cost of the method as a function of its input arguments.

In order to assess the practicality of our cost analysis framework, we have implemented a prototype analyzer in `Ciao` [9]. Some experimental results showing the run-times of the different phases of the cost analysis process can be found in [3]. We point out that computed cost relations, in many cases, can be simplified to the point of deriving statically an *upper and lower* bound. Such simplifications have been well-studied in the field of algorithmic complexity (see e.g. [26]).

2 The Java Bytecode Language

Java bytecode [17] is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory (the *heap*). A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* c, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature m which consists of its name and its type; its bytecode $bc_m = \langle pc_0 : b_0, \ldots, pc_{n_m} : b_{n_m} \rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table.

In this work we consider a subset of the JVM [17] language which is able to handle operations on integers, object creation and manipulation (by accessing fields and calling methods) and exceptions (either generated by abnormal execution or explicitly thrown by the program). We omit interfaces, static fields and methods and primitive types different from integers. Methods are assumed to return an integer value.

3 From Bytecode to Control Flow Graphs

This section describes the generation of a *control flow graph* (CFG) from the bytecode of a method. This will allow transforming the unstructured control flow of bytecode into recursion. The technique we use follows well-established ideas on compilers [1], already applied in Java bytecode analysis [23].

Given a method m, we denote by G_m its CFG, which is a directed graph whose nodes are referred to as *blocks*. Each block $Block_{id}$ is a tuple of the form $\langle id, G, B, D \rangle$ where: id is the block's unique identifier; G is the *guard* of the block which indicates under which conditions the block is executed; B is a sequence of contiguous bytecode instructions which are guaranteed to be executed unconditionally (i.e., if G succeeds then all instruc-

tions in B are executed before control moves to another block); and D is the *adjacency list* for $Block_{id}$, i.e., D contains the identifiers of all blocks which are possible successors of $Block_{id}$, i.e., $id' \in D$ iff there is an arc from $Block_{id}$ to $Block_{id'}$. Guards originate from bytecodes where the execution might take different paths depending on the runtime values. This is the case of bytecodes for conditional jumps, method invocation, and exceptions manipulation. In the CFG this will be expressed by *branching* from the corresponding block. The successive blocks will have mutually exclusive guards since only one of them will be executed. Guards take the form `guard(cond)`, where `cond` is a Boolean condition on the local variables and stack elements of the method. It is important to point out that guards in the successive blocks will not be taken into account when computing the cost of a program. A large part of the bytecode instruction set has only one successor. However, there are three types of branching statements:

Conditional jumps: of the form "pc_i:if$\diamond$ pc_j".[1] When this instruction is executed, depending on the truth value of the condition, the execution can jump to pc_j or continue, as usual, with pc_{i+1}. The graph describes this behavior by means of two arcs from the block containing the instruction of pc_i to those starting respectively with instructions of pc_j and pc_{i+1}. Each one of these new blocks begins by a `guard` expressing the condition under which such block is to be executed.

Dynamic dispatch: of the form "pc_i:invokevirtual $c.m$". The type of the object o whose method is being invoked is not known statically (it could be c or any subclass of c); therefore, we cannot determine statically which method is going to be invoked. Hence, we need to make all possible choices explicit in the graph. We deal with dynamic dispatching by using the function `resolve_virtual`(c, m), which returns the set *ResolvedMethods* of pairs $\langle d, \{c_1, \ldots, c_k\}\rangle$, where d is a class that defines a method with signature m and each c_i is either c or a subclass of c which inherits that specific method from d. For each $\langle d, \{c_1, \ldots, c_k\}\rangle \in$ *ResolvedMethods*, a new block $Block_d^{pc_i}$ is generated with a unique instruction `invoke`$(d{:}m)$ which stands for the *non-virtual* invocation of the method m that is defined in the class d In addition, the block has a guard of the form `instanceof`$(o, \{c_1, \ldots, c_k\})$ (o is a stack element) to indicate that the block is applicable only when o is an instance of one of the classes $c_1, \ldots, c_k$. An arc from the block containing pc_i to $Block_d^{pc_i}$ is added, together with an arc from $Block_d^{pc_i}$ to the block containing the next instruction at pc_{i+1} (which describes the rest of the execution after invoking m). Yet, in order to take into account the cost of dynamic dispatching, we replace the `invokevirtual` by a corresponding call to `resolve_virtual`. Fields are treated in a similar way.

Exceptions: As regards the structure of the CFG, exceptions are not dealt with in a special way. Instead, the possibility of an exception being raised while executing a bytecode statement b is simply treated as an additional branching after b. Let $Block_b$ be the block ending with b; arcs exiting from $Block_b$ are those originated by its *normal behavior* control flow, together with those reaching the sub-graphs which correspond to exception handlers.

Describing dynamic dispatching and exceptions as additional blocks simplifies program analysis. After building the CFG, we do not need to distinguish how and why blocks were generated. Instead, all blocks can be dealt with uniformly.

Example 3.1 (running example) *The execution of the method* `add(n,o)` *shown in Fig. 1 computes:* $\Sigma_{i=0}^{n} i$ *if* `o` *is an instance of A;* $\Sigma_{i=0}^{\lfloor n/2 \rfloor} 2i$ *if* `o` *is an instance of B; and* $\Sigma_{i=0}^{\lfloor n/3 \rfloor} 3i$ *if* `o` *is an instance of C. The CFG of the method* `add` *is depicted in Fig. 2. The fact that the successor of* 6: `if_icmpgt 16` *can be either the instruction at address* `7` *or* `16` *is expressed by means of two arcs from* Block_1*, one to* Block_2 *and another one to* Block_3*, and by adding the guards* icmpgt *and* icmple *to* Block_2 *and* Block_3*, respectively. The invocation* 13: `invokevirtual A.incr : (I)I` *is split into 3 possible runtime scenarios described in*

[1] $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.)

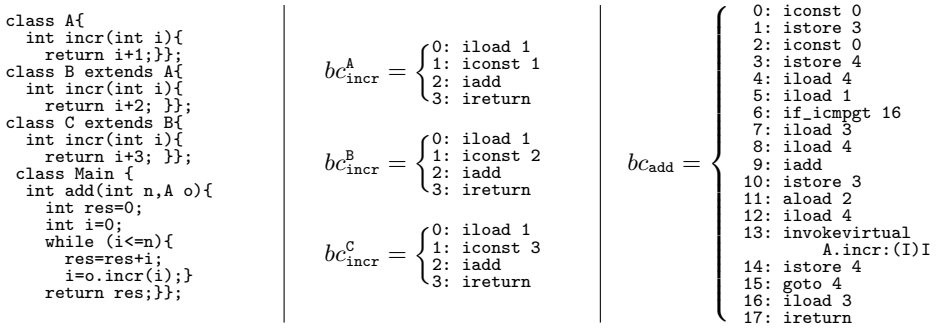

Figure 1: The running example in source code, and bytecode

blocks Block_4*,* Block_5 *and* Block_6*. Depending on the type of the object* `o` *(the second stack element from top, denoted* $\mathtt{s(top(1))}$ *in the guards), only one of these blocks will be executed and hence one of the definitions for* `incr` *will be invoked. Note that the* `invokevirtual` *bytecode is replaced by* `resolve_virtual`*. The exception behavior when* `o` *is a* `NULL` *object is described in blocks* Block_7 *and* Block_{exc}*.*

4 Recursive Representation with Flattened Stack

In this section, we present a method for obtaining a representation of the code of a method where iteration is transformed into recursion and the operand stack is *flattened* in the sense that its contents are represented as a series of local variables. The latter is possible because in valid bytecode the maximum stack height t can always be statically decided. For the sake of simplicity, exceptions possibly occurring in a method will be ignored. Handling them introduces more branching in the CFG and also requires additional arguments in the recursive representation. This could influence the performance of the cost analysis.

Let m be a method defined in class c, with *local* variables $\overline{l_k} = l_0, \ldots, l_k$; of them, l_0 contains a reference to the *this* object, $l_1, \ldots, l_n$ are the n input arguments to the method, and $l_{n+1}, \ldots, l_k$ correspond to the $k-n$ local variables declared in m. In addition to these arguments, we add the variables $\overline{s_t} = s_0, \ldots, s_{t-1}$, which correspond to the stack elements, with s_0 and s_{t-1} being the bottom-most and top-most positions respectively. Moreover, let h_{id} be the height of the stack at the entry of $Block_{id}$, and $\overline{s_t}|_{h_{id}}$ be the restriction of $\overline{s_t}$ to the corresponding stack variables. The *recursive representation* of m is defined as a set of rules *head* $\leftarrow$ *body* obtained from its control flow graph G_m as follows:

(1) the *method entry* rule is $\mathtt{c{:}m}(\overline{\mathtt{l_n}}, \mathtt{ret}) \leftarrow \mathtt{c{:}m}^0(\overline{\mathtt{l_k}}, \mathtt{ret})$, where `ret` is a variable for storing the return value,

(2) for each $Block_{id} = \langle id, G, \overline{B}_p, \{id_1, \ldots, id_j\}\rangle \in G_m$, there is a rule $\mathtt{c{:}m}^{id}(\overline{\mathtt{l_k}}, \overline{\mathtt{s_t}}|_{\mathtt{h_{id}}}, \mathtt{ret}) \leftarrow \mathtt{G'}, \overline{\mathtt{B}}'_{\mathtt{p}}(\mathtt{call}_{\mathtt{id_1}}\ ;\ \ldots\ ;\ \mathtt{call}_{\mathtt{id_j}})$, where $\{\mathtt{G'}\} \cup \overline{\mathtt{B}}'_{\mathtt{p}}$ is obtained from $\{\mathtt{G}\} \cup \overline{\mathtt{B}}_{\mathtt{p}}$, and $\mathtt{call}_{\mathtt{id_1}}\ ; \ldots ;\ \mathtt{call}_{\mathtt{id_j}}$ are calls to blocks (";" means disjunction), as explained below.

Each $b_i \in \{\mathtt{G}\} \cup \overline{\mathtt{B}}_{\mathtt{p}}$ is *translated* into b'_i by explicitly adding the variables (local variables or stack variables) used by b_i as arguments. For example, `iadd` is translated to $\mathtt{iadd}(s_{j-1}, s_j, s'_{j-1})$, where j is the index of the top of the stack just before executing `iadd`. Here, we refer to the $j{-}1^{th}$ stack variable twice by different names: s_{j-1} refers to the input value and s'_{j-1} refers to the output value. The use of new names for output variables, in the spirit of *Static Single Assignment (SSA)* (see [13] and its references), is crucial in order to obtain simple, yet efficient, denotational program analyses. In Fig. 3 we give the trans-

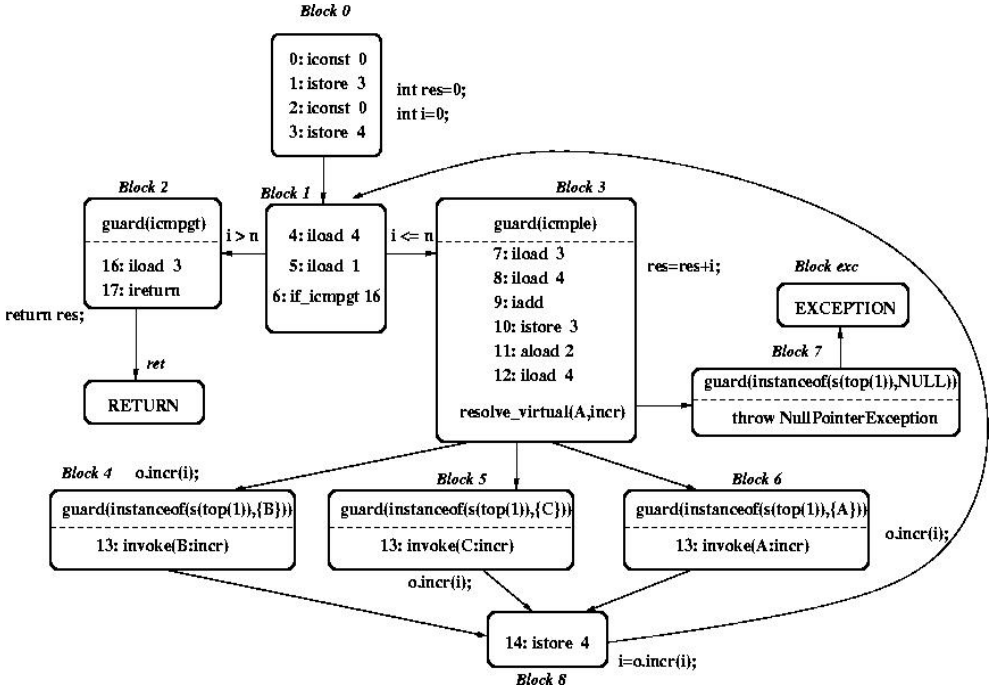

Figure 2: The control flow graph for the running example

lation function for selected bytecodes; among them, the one for `iadd` works as follows. Function `translate` takes as input the name of the current method m, the program counter pc of the bytecode, the bytecode (in this case `iadd`), the current local variable *names* $\overline{\mathtt{l_k}}$, and the current stack variable *names* $\overline{\mathtt{s_t}}$. In line 1, we retrieve the index of the top stack element before executing the current bytecode. In line 2, we generate new stack variable *names* $\overline{\mathtt{s'}}_\mathtt{t}$ by renaming the output variable of `iadd` in $\overline{\mathtt{s}}_\mathtt{t}$. As notation, given a sequence $\overline{a}_n$ of elements, $\overline{a}_n[i \mapsto b]$ denotes the replacement in $\overline{a}_n$ of the element a_i by b. In line 3, we return $(\mathtt{ret}\langle\_\rangle)$ the translated bytecode together with the new stack variable *names*. Assume that $\mathtt{G}{=}pc_0{:}b_0$ and $\overline{\mathtt{B}}_\mathtt{p}{=}\langle pc_1{:}b_1, \ldots, pc_\mathtt{p}{:}b_\mathtt{p}\rangle$. The translation of all bytecodes is done iteratively as follows:

```
for i = 0 to p
  {⟨b'_i, l_k^{i+1}, s_t^{i+1}⟩ =translate(m, pc_i, b_i, l_k^i, s_t^i)}
```

We start from an initial set of local and stack variables, $\overline{\mathtt{l}}^0_\mathtt{k}{=}\overline{\mathtt{l}}_\mathtt{k}$ and $\overline{\mathtt{s}}^0_\mathtt{t}{=}\overline{\mathtt{s}}_\mathtt{t}$; in each step, `translate` takes as input the local and stack variable names which were generated by translating the previous bytecode. At the end of this loop, we can define each $\mathtt{call}_{\mathtt{id}_i}$, $1 \leq i \leq j$, as $\mathtt{c{:}m}^{\mathtt{id}_i}(\overline{\mathtt{l}}^{\mathtt{p}+1}_\mathtt{k}, \overline{\mathtt{s}}^{\mathtt{p}+1}_\mathtt{t}|_{\mathtt{h}_{\mathtt{id}_i}}, \mathtt{ret})$, meaning that we call the next block with the last local and (restricted) stack variable *names*.

Several optimizations are applied to the above translation. An important one is to replace (redundant) stack variables corresponding to intermediate states by local variables whenever possible. This can be done by tracking dependencies between variables, which stem from instructions like `iload` and `istore`. The fact that the program is in SSA form makes this transformation relatively straightforward. However, note that, in order to eliminate stack variables from the head of a block, we need to consider all calling patterns to the

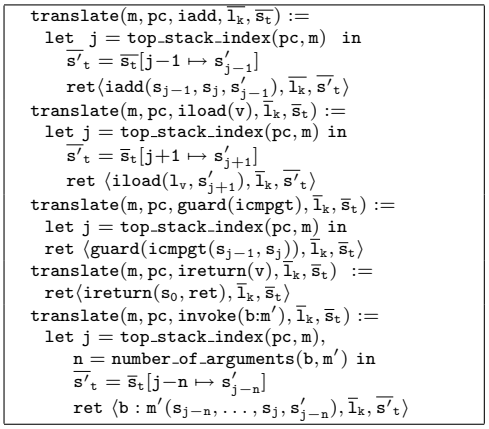

Figure 3: Translation of selected bytecode instructions

block.

Example 4.1 *Consider the CFG in Fig. 2. The translation (after eliminating redundant variables as commented above) of* Block_3 *and* Block_4 *works as shown below. For clarity, in the block identifiers we have not included the class name for the* add *method. Also, we ignore the exception branch from* Block_3 *to* Block_7*.*

$\texttt{add}^3(\overline{l_4}, \texttt{ret}) \leftarrow$
$\quad \mathbf{guard}(\mathbf{icmple}(l_4, l_1)),$
$\quad \underline{\texttt{iload}(l_3, s'_0), \texttt{iload}(l_4, s'_1), \texttt{iadd}(l_3, l_4, l'_3),}$
$\quad \underline{\texttt{istore}(s''_0, l'_3), \texttt{aload}(l_2, s'''_0), \texttt{iload}(l_4, s''_1),}$
$\quad \underline{\texttt{resolve\_virtual}(\texttt{A}, \texttt{incr}),}$
$\quad (\ \texttt{add}^4(l_0, l_1, l_2, l'_3, l_4, \texttt{ret})\ ;$
$\quad\ \ \texttt{add}^5(l_0, l_1, l_2, l'_3, l_4, \texttt{ret})\ ;$
$\quad\ \ \texttt{add}^6(l_0, l_1, l_2, l'_3, l_4, \texttt{ret})\ )$
$\texttt{add}^4(\overline{l_4}, \texttt{ret}) \leftarrow$
$\quad \mathbf{guard}(\mathbf{instanceof}(l_2, \{\mathbf{B}\})),$
$\quad \texttt{B:incr}(l_2, l_4, s'_0),$
$\quad \texttt{add}^8(\overline{l_4}, s'_0, \texttt{ret}).$

In the $\texttt{add}^3$ *rule, dynamic dispatch is represented as a disjunction of calls to* $\texttt{add}^4$, $\texttt{add}^5$ *or* $\texttt{add}^6$*. Thus, in the rule for* $\texttt{add}^4$*, we find a call to (the translation of)* incr *from class* B *which corresponds to the translation of* invoke(B:incr)*; arguments passed to* incr *are the two top-most stack elements; the return value (the last argument) goes also to the stack. Note the change in the superscript when a variable is updated.*

The underlined instructions have been used to discover equivalences among stack elements and local variables. For example, all the arguments of iadd *have been replaced by local variables. However, eliminating stack variables is not always possible. This is the case of* s'_0 *in the rule* $\texttt{add}^4$*, as it corresponds to the return value of* B:incr*. After these optimizations, the underlined instructions become redundant and could be removed. However, we do not remove them in order to take their cost into account in the next sections.*

5 Size Relations for Cost Analysis

Obtaining *size-relations* between the states at different program points is indispensable for setting up cost relations. In particular, they are essential for defining the cost of one block in terms of the cost of its successors. In general, various *measures* can be used to determine the *size* of an input. For instance, in symbolic languages (see, e.g., [14]), term-depth, list-length, etc. are used as term sizes. In Java bytecode, we consider two cases: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths.

Inferring *size-relations* is not straightforward: such relations might be the result of executing several statements, calling methods or loops. For instance, in our running example, the size relation for variable i is the result of executing the method incr and is propagated through the loop in the procedure add. Fixpoint computation is often required. Fortunately, there are several abstract interpretation based approaches for inferring *size-relations* between integer variables [11], as well as between reference variables (in terms of longest path length) [24].

5.1 The notion of *Size Relation*

In order to set up cost relations, we need, for each rule in the recursive representation, the *calls-to size-relations* between the variables in the head of the rule and the variables used

in the calls (to rules) which occur in the body. Note that, given a rule $\mathtt{p}(\bar{\mathtt{x}})\leftarrow \mathtt{G}, \overline{\mathtt{B}}_\mathtt{k}, (\mathtt{q}_1; \ldots ; \mathtt{q}_\mathtt{n})$, each $\mathtt{b}_\mathtt{i}\in\overline{\mathtt{B}}_\mathtt{k}$ is either a bytecode or a call to another rule (which stems from the translation of a method invocation). We denote by $\mathtt{calls}(\overline{\mathtt{B}}_\mathtt{k})$ the set of all $\mathtt{b}_\mathtt{i}$ corresponding to a method call, and by $\mathtt{bytecode}(\overline{\mathtt{B}}_\mathtt{k})$ the set of all $\mathtt{b}_\mathtt{i}$ corresponding to other bytecodes.

Definition 5.1 (calls-to size-relations) *Let $\mathcal{R}_m$ be the recursive representation of a method m, where each rule takes the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G}, \overline{\mathtt{B}}_\mathtt{k}, (\mathtt{q}_1(\bar{\mathtt{y}}); \cdots ; \mathtt{q}_\mathtt{n}(\bar{\mathtt{y}}))$. The calls-to* size-relations *of $\mathcal{R}_m$ are triples of the form $\langle p(\bar{x}),\ \mathtt{p}'(\bar{\mathtt{z}}), \varphi\rangle$ where $\mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls}(\overline{\mathtt{B}}_\mathtt{k}) \cup \{\mathtt{p\_cont}(\bar{\mathtt{y}})\}$ describing, for all rules, the* size-relation *between $\bar{\mathtt{x}}$ and $\bar{\mathtt{z}}$ when $\mathtt{p}'(\bar{\mathtt{z}})$ is called, where $\mathtt{p\_cont}(\bar{\mathtt{y}})$ refers to the program point immediately after $\overline{\mathtt{B}}_\mathtt{k}$. The* size-relation *$\varphi$ is given as a conjunction of linear constraints $\mathtt{a}_0{+}\mathtt{a}_1\mathtt{v}_1{+}\cdots{+}\mathtt{a}_\mathtt{n}\mathtt{v}_\mathtt{n}\ \mathtt{op}\ 0$, where $\mathtt{op}\in\{=,\leq,<\}$, each $\mathtt{a}_\mathtt{i}$ is a constant and $\mathtt{v}_\mathtt{k}\in\bar{\mathtt{x}}\cup\bar{\mathtt{z}}$.*

Note that in the definition above there is no need to have separate relations for each $\mathtt{q}_\mathtt{i}(\bar{\mathtt{y}})$ as, in the absence of exceptions, size relations are exactly the same for all of them, since they correspond to the same program point.

5.2 Inferring Size Relations

A simple, yet quite precise and efficient, *size-relation* analysis for the recursive representation of methods can be done in two steps: 1) compiling the bytecodes into the linear constraints they impose on variables; and 2) computing a bottom-up fixpoint on the compiled rules using standard bottom-up fixpoint algorithms. Compilation into linear constraints is done by an abstraction function α_{size} which basically replaces guards and bytecodes by the constraints they impose on the corresponding variables. In general, each bytecode performing (linear) arithmetic operations is replaced by a corresponding linear constraint, and each bytecode which manipulates objects is compiled to linear constraints on the length of the longest reachable path from the corresponding variable [24]. Next, we show examples of abstracting guards and bytecodes into linear constraints:

$\alpha_{\mathtt{size}}(\mathtt{iload}(\mathtt{l}_1, \mathtt{s}_0)) := (\mathtt{l}_1 = \mathtt{s}_0)$
$\alpha_{\mathtt{size}}(\mathtt{iadd}(\mathtt{s}_1, \mathtt{s}_0, \mathtt{s}'_0)) := (\mathtt{s}'_0 = \mathtt{s}_0 + \mathtt{s}_1)$
$\alpha_{\mathtt{size}}(\mathtt{guard}(\mathtt{icmpgt}(\mathtt{s}_1, \mathtt{s}_0))) := (\mathtt{s}_1 > \mathtt{s}_0)$
$\alpha_{\mathtt{size}}(\mathtt{getfield}(\mathtt{s}_1, \mathtt{f}, \mathtt{s}'_1)) := (\mathtt{s}'_1 < \mathtt{s}_1)$

It is important to note that $\alpha_{\mathtt{size}}$ uses the same name for the original variables in order to refer to their sizes.

Example 5.2 *Compiling all the rules corresponding to the program in Fig. 1 and computing a bottom-up fixpoint over an appropriate abstract domain [11] would result in the following calls-to* size-relations *for rules from Ex. 4.1:*

$\langle \mathtt{add}^3(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}),$
$\quad \mathtt{add}^3\mathtt{\_cont}(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}'_3, \mathtt{l}_4, \mathtt{ret}),$
$\quad \{\mathtt{l}_4 \leq \mathtt{l}_1, \mathtt{l}'_3 = \mathtt{l}_3 + \mathtt{l}_4\}\rangle$
$\langle \mathtt{add}^4(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}), \mathtt{B}{:}\mathtt{incr}(\mathtt{l}_2, \mathtt{l}_4, \mathtt{ret}), \{\}\rangle$
$\langle \mathtt{add}^4(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{ret}),$
$\quad \mathtt{add}^4\mathtt{\_cont}(\mathtt{l}_0, \mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4, \mathtt{s}'_0, \mathtt{ret}),$
$\quad \{\mathtt{s}'_0 = \mathtt{l}_4 + 2\}\rangle$

6 Cost Relations for Java Bytecode

We now present our approach to the automatic generation of *cost relations* which define the computational cost of the execution of a bytecode method. They are generated from the recursive representation of the method (Sec. 4) and by using the information inferred by the size analysis (Sec. 5). An important issue in order to obtain optimal cost relations is to find out the arguments which can be safely ignored in cost relations.

6.1 Restricting Cost Relations to (Subsets of) Input Arguments

Let us consider $Block_{id}$ in a CFG, represented by the rule $\mathtt{c}{:}\mathtt{m}^{id}(\overline{\mathtt{l}_\mathtt{k}}, \mathtt{ret}) \leftarrow \mathtt{G}, \overline{\mathtt{B}}_\mathtt{h}, (\mathtt{call}_{\mathtt{id}_1}\ ;\ \ldots\ ;\ \mathtt{call}_{\mathtt{id}_\mathtt{j}})$ in which local and stack variables are no longer distinguishable. The cost function for $Block_{id}$ takes the form $C_{id}{:}(\mathbb{Z})^n \rightarrow \mathbb{N}_\infty$, with $n \leq k$ argument positions, and where $\mathbb{Z}$ is the set of integers and $\mathbb{N}_\infty$ is the set of natural numbers augmented with a special symbol ∞, denoting *unbounded*.

Our aim here is to minimize the number n of arguments which need to be taken into account in cost functions. As usual in cost analysis, we

consider that the output argument `ret` cannot influence the cost of any block, so that it can be ignored in cost functions. Furthermore, it is sometimes possible to disregard some input arguments. For instance, in our running example, $\mathtt{l}_3$ is an *accumulating* parameter whose value does not affect the control flow nor the cost of the program: it merely keeps the value of the temporary result.

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. Computing a safe approximation of the set of variables affecting a series of statements is a well studied problem in static analysis. To do this, we need to follow data dependencies against the control flow, and this involves computing a fixpoint. Our problem is slightly simpler than *program slicing* [25], since we do not need to delete redundant program statements; instead, we only need to detect relevant arguments. Given a rule $p(\overline{\mathtt{x}})\leftarrow body$ (p for short), $\hat{\mathtt{l}}_p \subseteq \overline{\mathtt{x}}$ is the sub-sequence of *relevant variables* for p. The sequence $\hat{\mathtt{l}}_P$, obtained by union of sequences $\{\hat{\mathtt{l}}_p\}_{p\in P}$ for a set P of rules, keeps the ordering on variables.

Example 6.1 *Given p_i, corresponding to* Block_i *in the graph of the running example, we are interested in computing which variables in this rule are relevant to program guards or external methods. For example, 1) when the execution flow reaches p_2, we execute the unconditional bytecode instructions in p_2 and move to the final block. As a result, there are no relevant variables for p_2, since none can have any impact on its cost, and p_2 does not* reach *any guards nor methods. 2) On the other hand, p_3 can reach the guards in p_4, p_5 and p_6, which take the form* `instanceof(_)` *and involve* $\mathtt{l}_2$. *Also, the guard in p_3 itself, involving* $\mathtt{l}_1$ *and* $\mathtt{l}_4$, *can be recursively reached via the loop. Moreover, the call to the external method* ***incr*** *involves* $\mathtt{l}_2$ *and* $\mathtt{l}_4$. *After computing a fixpoint, we conclude that* $\hat{\mathtt{l}}_{p_3}=\{\mathtt{l}_1,\mathtt{l}_2,\mathtt{l}_4\}$. *3) We have* $\hat{\mathtt{l}}_{p_8}=\{\mathtt{l}_1,\mathtt{l}_2,\mathtt{s}_0\}$; *here,* $\mathtt{s}_0$ *is also relevant since it affects* $\mathtt{l}_4$ *(which in turn is involved in the guard of p_3, reachable from p_8).*

6.2 The Cost Relation

We define the cost function $C_{id}{:}(\mathbb{Z})^n{\rightarrow}\mathbb{N}_\infty$ for a $Block_{id}$ by means of a *cost relation* which consists of a set of *cost equations*. The idea is that, given the rule $\mathtt{p}(\overline{\mathtt{x}})\leftarrow \mathtt{G},\mathtt{B},(\mathtt{q}_1;\ldots;\mathtt{q}_n)$ for $Block_{id}$, we generate:

- one cost equation which defines the cost of `p` as the cost of the statements in `B`, plus the cost of its *continuation*, denoted `p_cont`;
- another cost equation which defines the cost of `p_cont` as either the cost of $\mathtt{q}_1$ (if its guard is satisfied), ..., or the cost of $\mathtt{q}_n$ (if its guard is satisfied).

We specify the cost of the *continuation* in a separate equation since the conditions for determining the alternative path $\mathtt{q}_i$ that the execution will take (with $i{=}1,\ldots,n$) are only known at the end of the execution of `B`; thus, they cannot be evaluated before `B` is executed. In the definition below, we use the function α_{guard} to replace those guards which indicate the type of an object by the appropriate test (e.g., $\alpha_{guard}(\mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_0,\{\mathtt{B}\}))){:=}\mathtt{s}_0{\in}\mathtt{B}$). For size relations, it is equivalent to α_{size}.

Definition 6.2 (cost relation) *Let $\mathcal{R}_m$ be the recursive representation of a method m where each block takes the form* $\mathtt{p}(\overline{\mathtt{x}}) \leftarrow \mathtt{G_p},\mathtt{B},(\mathtt{q}_1(\overline{\mathtt{y}});\cdots;\mathtt{q}_n(\overline{\mathtt{y}}))$ *and* $\hat{\mathtt{l}}_\mathtt{p}$ *be its sequence of relevant variables. Let φ be the calls-to size relation for $\mathcal{R}_m$ where each size relation is of the form* $\langle \mathtt{p}(\overline{\mathtt{x}}),\mathtt{p}'(\overline{\mathtt{z}}),\varphi^{\mathtt{p}(\overline{\mathtt{x}})}_{\mathtt{p}'(\overline{\mathtt{z}})}\rangle$ *for all* $\mathtt{p}'(\overline{\mathtt{z}}) \in \mathtt{calls(B)} \cup\{\mathtt{q}(\overline{\mathtt{y}})\}$ *such that* $\mathtt{q}(\overline{\mathtt{y}})$ *refers to the program point immediately after* `B`. *Then, we generate the cost equations for each block of the above form in $\mathcal{R}_m$ as follows:*

$$C_p(\hat{\mathtt{l}}_p) = \sum_{b\in bytecode(B)} T_b \;+\; \sum_{r(\bar{z})\in calls(B)} C_r(\hat{\mathtt{l}}_r) + C_{p\_cont}(\cup_{i=1}^n \hat{\mathtt{l}}_{q_i}) \quad \textbf{if} \bigwedge_{r(\bar{z})\in calls(B)} (\varphi^{p(\overline{x})}_{r(\bar{z})}) \wedge \varphi^{p(\overline{x})}_{q(\bar{y})}$$

$$C_{p\_cont}(\cup_{i=1}^n \hat{\mathtt{l}}_{q_i}) = \begin{cases} C_{q_1}(\hat{\mathtt{l}}_{q_1}) \textbf{ if } \alpha_{guard}(G_{q_1}) \\ \cdots \\ C_{q_n}(\hat{\mathtt{l}}_{q_n}) \textbf{ if } \alpha_{guard}(G_{q_n}) \end{cases}$$

where T_b is the cost unit associated to the bytecode b. The cost relation associated to $\mathcal{R}_m$ and φ is defined as the set of cost equations of its blocks.

Let us notice four points about the above definition. 1) The size relationships between the input variables provided by the size analysis are *attached* to the cost equation for p. 2) Guards do not affect the cost: they are simply used to define the applicability conditions of the equations. 3) Arguments of the cost equations are only the relevant arguments to the block. In the equation for the continuation, we need to include the union of all relevant arguments to each of the subsequent blocks $\mathtt{q_i}$.

The cost T_b of an instruction b depends on the chosen *cost model*. If our interest is merely on finding out the complexity or on approximating the number of bytecode statements which will be executed, then T_b can be the same for all instructions. On the other hand, we may use more refined cost models in order to estimate the execution time of methods. Such models may assign different costs to different instructions. One approach might be based on the use of a *profiling* tool which estimates the value of each T_b on a particular platform. (see, e.g., an application [18] for Prolog). It should be noted that, since we are not dealing with the problem of choosing a realistic cost model, a direct comparison between the result of our analysis and the actual measured run time (e.g., in milliseconds) cannot be done; instead, in this paper we focus only on the number of instructions to be executed.

Example 6.3 *Consider the recursive representation in Ex. 4.1 (without irrelevant variables, as explained in Ex. 6.1). Consider the size relations derived in Ex. 5.2; by applying Def. 6.2, we obtain the following cost relations:*

$$C_{\mathtt{add}}(\mathtt{l_1},\mathtt{l_2}) = C_{\mathtt{add^0}}(\mathtt{l_1},\mathtt{l_2})$$
$$C_{\mathtt{add^0}}(\mathtt{l_1},\mathtt{l_2}) = \mathtt{T_0} + C_{\mathtt{add^1}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4'}) \quad \mathtt{l_4'} = \mathtt{0}$$
$$C_{\mathtt{add^1}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \mathtt{T_1} + C_{\mathtt{add^1\_cont}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4})$$
$$C_{\mathtt{add^1\_cont}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \begin{cases} C_{\mathtt{add^2}}() & \mathtt{l_4} > \mathtt{l_1} \\ C_{\mathtt{add^3}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) & \mathtt{l_4} \leq \mathtt{l_1} \end{cases}$$
$$C_{\mathtt{add^2}}() = \mathtt{T_2}$$
$$C_{\mathtt{add^3}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \mathtt{T_3} + C_{\mathtt{add^3\_cont}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4})$$
$$C_{\mathtt{add^3\_cont}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \begin{cases} C_{\mathtt{add^4}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) & \mathtt{l_2} \in \mathtt{B} \\ C_{\mathtt{add^5}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) & \mathtt{l_2} \in \mathtt{C} \\ C_{\mathtt{add^6}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) & \mathtt{l_2} \in \mathtt{A} \end{cases}$$
$$C_{\mathtt{add^4}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \mathtt{T_4} + C_{\mathtt{B:incr}}(\mathtt{l_2},\mathtt{l_4}) + C_{\mathtt{add^8}}(\mathtt{l_1},\mathtt{l_2},\mathtt{s_0}) \quad \mathtt{s_0} = \mathtt{l_4} + \mathtt{2}$$
$$C_{\mathtt{add^5}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \mathtt{T_5} + C_{\mathtt{C:incr}}(\mathtt{l_2},\mathtt{l_4}) + C_{\mathtt{add^8}}(\mathtt{l_1},\mathtt{l_2},\mathtt{s_0}) \quad \mathtt{s_0} = \mathtt{l_4} + \mathtt{3}$$
$$C_{\mathtt{add^6}}(\mathtt{l_1},\mathtt{l_2},\mathtt{l_4}) = \mathtt{T_6} + C_{\mathtt{A:incr}}(\mathtt{l_2},\mathtt{l_4}) + C_{\mathtt{add^8}}(\mathtt{l_1},\mathtt{l_2},\mathtt{s_0}) \quad \mathtt{s_0} = \mathtt{l_4} + \mathtt{1}$$
$$C_{\mathtt{add^8}}(\mathtt{l_1},\mathtt{l_2},\mathtt{s_0}) = \mathtt{T_8} + C_{\mathtt{add^1}}(\mathtt{l_1},\mathtt{l_2},\mathtt{s_0})$$

$\mathtt{T_{B_i}}$ denotes the sum of the costs of all bytecode instructions contained in Block_i*. For brevity, as the blocks* `0`*,* `2`*,* `4`*,* `5`*,* `6`*, and* `8` *have a single-branched continuation, we merge their two equations. The cost relation for the external method* `incr` *does not include the third argument since it is an output argument.*

7 Conclusions

We have presented an automatic approach to the cost analysis of Java bytecode, based on generating at compile-time cost relations for an input bytecode program. Such relations are functions of input data which are informative by themselves about the computational cost, provided an accurate size analysis is used to establish relationships between the input arguments. Essentially, the sources of inaccuracy in size analysis are: 1) guards depending (directly of indirectly) on values which are not handled in the abstraction, e.g., non-integer values, numeric fields or multidimensional arrays, cyclic data-structures; 2) loss of precision due to the abstraction of (non-linear) arithmetic instructions and domain operations like widening. In such cases, we can still set up cost relations; however, they might not be useful if the size relationships are not precise enough.

To the best of our knowledge, our work presents the first approach to the automatic cost analysis of Java bytecode. We have implemented a prototype analyzer and shown some experimental results in [3]. In [4], our generic framework has been used to infer upper bounds on the heap space usage of Java bytecode programs.

Related work in the context of Java bytecode includes the work in the MRG project [6], which can be considered complementary to ours. MRG focuses on building a proof-carrying code [19] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds.Another related work is [10], where a resource usage analysis is presented. Again, this work focuses on 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. The analysis has been certified by proving its correctness using the Coq proof assistant.

References

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

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP'07*, LNCS 4421, Springer, 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *BYTECODE'07*, ENTCS, 2007. To appear.

[4] E. Albert, S. Genaim, M. Gómez-Zamalloa. Heap Space Analysis of Java Bytecode. In *ISMM'07*, ACM Press, 2007. To appear.

[5] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *LPAR'04*, LNAI 3452, 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, Springer, 2005.

[7] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. Mobius: Mobility, Ubiquity, Security: Objectives and progress report. In *TGC'06*, LNCS, 2007. To appear.

[8] R. Benzinger. Automated Higher-Order Complexity Analysis. *TCS*, 318(1-2), 2004.

[9] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). TR, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[10] D. Cachera, D. Pichardie T. Jensen, and G. Schneider. Certified Memory Usage Analysis. In *FM'05*, LNCS 3582, Springer, 2005.

[11] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL'78*. ACM, 1978.

[12] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*. ACM Press, 2000.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In *TOPLAS*, 13(4), 1991.

[14] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *TOPLAS*, 15(5), 1993.

[15] 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.

[16] G. Gomez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *PEPM'02*. ACM Press, 2002.

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

[18] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *PADL'07*, LNCS 4354, Springer, 2007.

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

[20] 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.

[21] M. Rosendhal. Automatic Complexity Analysis. In *FPCA'89*. ACM, 1989.

[22] D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *J. Log. Comput.*, 5(4), 1995.

[23] F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *FTfJP'2005*, 2005.

[24] F. Spoto, P. M. Hill, and E. Payet. Path-Length Analysis for Object-Oriented Programs. In *EAAI'06*, 2006.

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

[26] H. S. Wilf. Algorithms and Complexity. A.K. Peters Ltd, 2002.

[27] R. Wilhelm. Timing Analysis and Timing Predictability. In *FMCO'04*, LNCS, Springer, 2004.