Experiments in Cost Analysis of Java Bytecode

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

[1] *DSIC, Complutense University of Madrid,* {elvira,puri}@sip.ucm.es

[2] *Technical University of Madrid,* {german,samir,damiano}@clip.dia.fi.upm.es

Abstract

Recently, we proposed a general framework for the cost analysis of Java bytecode which can be used for measuring resource usage. This analysis generates, at compile-time, *cost relations* which define the cost of programs as a function of their input data size. The purpose of this paper is to assess the practicality of such cost analysis by experimentally evaluating a prototype analyzer implemented in Ciao. With this aim, we approximate the computational complexity of a set of selected benchmarks, including both well-known algorithms which have been used to evaluate existing cost analyzers in other programming paradigms, and other benchmarks which illustrate object-oriented features. In our evaluation, we first study whether the generated cost relations can be automatically solved. Our experiments show that in some cases the inferred cost relations can be automatically solved by using the Mathematica system, whereas, in other cases, some prior manipulation is required for the equations to be solvable. Moreover, we experimentally evaluated the running time of the different phases of the analysis process. Overall, we believe our experiments show that the efficiency of our cost analysis is acceptable, and that the obtained cost relations are useful in practice since, at least in our experiments, it is possible to get a closed form solution.

Keywords: Cost analysis, Java bytecode, cost relations, recurrence equations, complexity.

1 Motivation

Having information about the execution cost [17,11] of a piece of code is quite useful; in many cases, this aspect is crucial in choosing among different implementations of the same specification. Moreover, this may allow certifying that the execution of an application meets the specified resource-consumption constraints [10]. Cost analysis is also (and especially) very useful in the context of mobile code, where resources are very limited and we may want to accept or reject code depending on its cost. In the limit, accepting mobile code without cost guarantees [6,12] can be a source of denial-of-service attacks, since execution can be very (or infinitely) costly. It is important to note that it is unlikely to have access to the source code in the above-mentioned situations; rather, we can only directly deal with the compiled code. Java bytecode [14] is becoming one of the most popular formats for mobile code. Having accurate cost analyzers for Java bytecode is hence desirable.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

In general, cost analysis is far from being trivial; it takes a good amount of expertise for programmers to have an intuition about which implementation techniques might lead to more efficient programs. This is particularly difficult when we are dealing with a low-level, object-oriented language such as Java bytecode. In some sense, it is to be expected that automating cost analysis of Java bytecode will not always succeed in giving meaningful results, especially for very intricate code. Thus, one of the main questions about the cost analysis framework we have recently proposed [5], and which we assess in practice in the present work, is whether the generated cost relations can be automatically simplified into a *closed form solution* [22] when considering the simple cost model traditionally used in complexity analysis, which counts the number of execution steps (bytecode instructions).

In order to find closed form solutions we use a state-of-the-art recurrence equations solver, the Mathematica system [1]. We study, for a series of representative benchmarks, whether the generated cost relations can be solved by using the provided RSolve query. When this is not directly possible, we propose transformations which make the generated equations solvable.

As regards the input language, and as in [5], we consider a subset of Java bytecode which does not include features such as dynamic class loading, reflection, or floating point arithmetic. Indeed, this subset basically corresponds to the CLDC, a variant of Java for the embedded industry which stands between JavaCard and the Java Standard Edition. We believe that CLDC is a good choice because it has all the characteristics of a real language: true memory management, object orientation, etc., while being at the same time much more manageable than the Java Standard Edition from the point of view of the analysis. Furthermore, CLDC is widely accepted by the industry as a runtime environment for downloadable code: on mobile phones (MIDP), set-top-boxes (JSR 242) and smart card terminal equipment (STIP).

Work on cost analysis by means of size inference has been mostly carried on in logic [11] and functional [16] programming. Debray and Lin's work [11] investigates key features of logic programming and generates cost information by abstracting the recursive structure of the program. Some recent work in functional languages [15,21] involves using type systems in order to study size relations and infer cost equations. In both cases, the issue of obtaining a closed form for cost relations is not discussed in depth, and no examples of solutions are provided (although references to the underlying mathematical theory are given). In [8], a static analysis approach for over-approximating the amount of memory allocated by source Java-like object-oriented programs is presented. Here, object size is represented as symbolic expressions. No cost equations are involved in such a method.

The rest of this paper is structured as follows. Sec. 2 provides an overview of our cost analysis framework for Java bytecode [5]. Afterwards, we study three classes of benchmarks. In Sec. 3, we analyze some well-known recursive procedures which, due to their structure, give rise to cost relations which can be easily handled by Mathematica. Sec. 4 deals with programs which use arrays, with both simple and nested loops, and require some simple transformations in order to solve the equations. In Sec. 5, we evaluate some programs with object-oriented features, like objects and dynamic dispatching; the obtained cost relations can be handled by

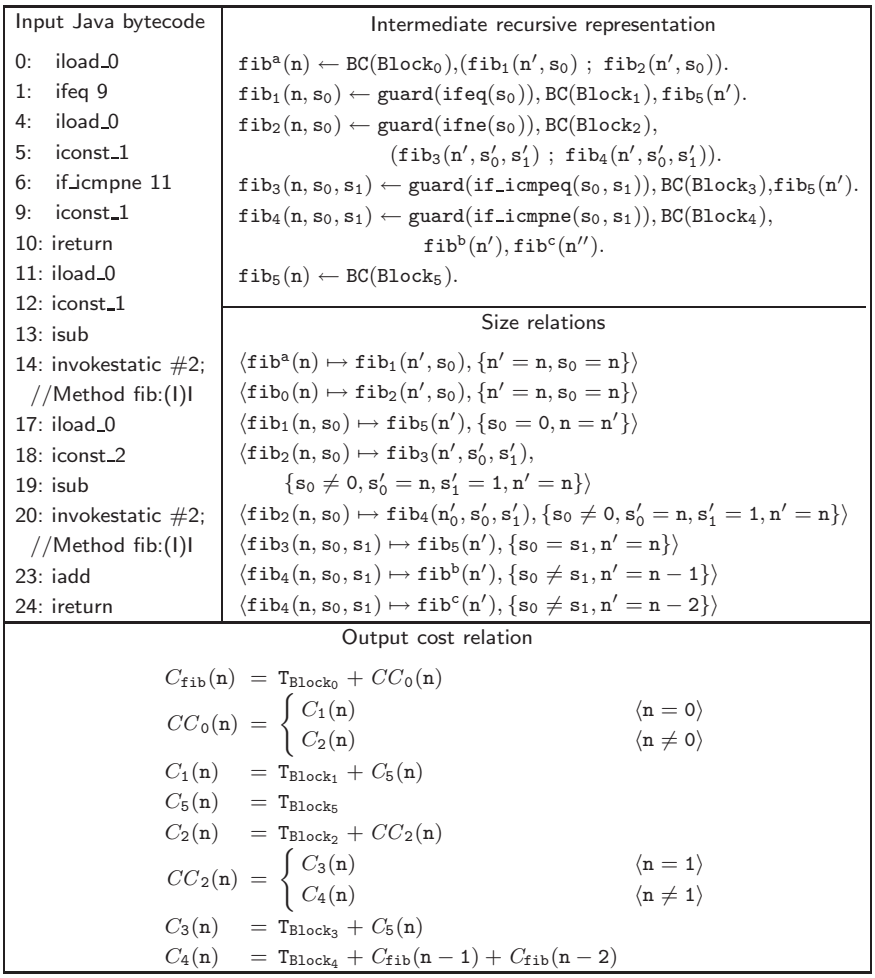

Fig. 1. Overview of the Cost Analysis Phases

Mathematica only after some transformations. Finally, Sec. 6 presents experimental results about the time required by our analysis, and concludes the paper.

2 An Overview of Cost Analysis of Java Bytecode

We briefly recall, by means of an example, the different phases of the cost analysis we recently proposed [5]. The running example is shown in Fig. 1; it corresponds to a naïve recursive implementation of the well-known Fibonacci number series. Given a natural number n, the call fib(n) computes the n-th term in the Fibonacci series. The input bytecode to the cost analysis can be seen at the top-left part of the figure. The input variable n is stored, at the bytecode level, in the local variable with index 0. This local variable is compared with the constants 0 and 1 (the base cases of the Fibonacci series) in lines 1 and 6. If either of the comparisons succeeds, execution jumps to bytecode instruction 9, where the constant 1 is pushed on the stack and returned as the result of the method. Otherwise, i.e., when both comparisons fail, control goes to bytecode instruction 11, where the method is called recursively twice (lines 14 and 20), with values $n-1$ and $n-2$, respectively. The obtained values are added, thus giving the return value of the method. Our cost analysis starts from

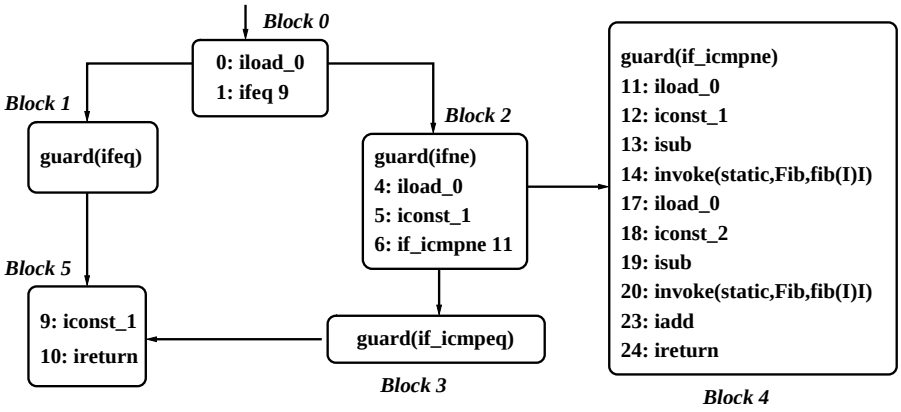

Fig. 2. Control flow graph

such input bytecode and carries out five main analysis steps which are described in the following subsections.

2.1 Control Flow Graph

First, the bytecode associated to a method is transformed into a *Control Flow Graph* (CFG) by using well-established ideas in compilers theory [2,3], already applied in Java Bytecode analysis [18]. This is instrumental to transform the unstructured control flow of the bytecode into recursion.

Consider the CFG in Figure 2. Each block contains an identifier (Block i), an optional *guard*, and a (possibly empty) sequence of contiguous bytecode instructions which are guaranteed to be executed sequentially. Block 0 is the initial block. Edges in the CFG show the different execution paths. *Branching* originated by exceptions, conditional jumps or dynamic dispatching is controlled by means of guards of the form guard(C), which indicate conditions under which blocks are executed. For example, Block 1 and Block 2 contain guard(ifeq) and guard(ifne), respectively: if the element which is on the top of the stack when executing bytecode instruction 1 (ifeq 9) is equal to 0, then execution moves to Block 1, whereas execution moves to Block 2 if such top element is different from 0. We point out that guards are not taken into account when computing the cost of a program, since they are not part of the original program; yet, they provide information which is very relevant for the generation of accurate recurrence cost equations.

2.2 Recursive Representation

From the CFG we obtain a *recursive representation* of the method, where iteration is transformed into recursion. In this representation, each block in the graph is represented as a *rule*. In addition, the operand stack is *flattened* by converting its content into a series of additional local variables. Note that this is possible since, in every valid bytecode program, the height of the local stack at each program point can be computed statically.

(A simplified version of) the recursive representation for the fib method is shown at the top-right corner of Fig. 1. As fib is indeed a recursive method, the relevance of this representation is not evident in this case (yet, see, e.g., the example in [5]).

The notation $\mathtt{fib}_\mathsf{i}(\overline{\mathtt{x}}) \leftarrow \mathsf{B_i}$ means that the execution of the block Block i on the input variables $\overline{\mathtt{x}}$ consists of the actions contained in $\mathsf{B_i}$. Guards and block calls are explicitly represented but, for simplicity, the rest of the bytecode is written as $\mathtt{BC}(\mathtt{Block}_\mathsf{i})$. The ";" operator stands for *determinate disjunction*, meaning that exactly one of the terms (corresponding to exclusive paths on the CFG) will succeed for each combination of values of the local variables. The main rule is the one for $\mathtt{fib}^\mathtt{a}$; the superscript is only used to distinguish among different calls to the main rule occurring in the body of a same rule, as in the case of rule for $\mathtt{fib}_4$, which contains calls to $\mathtt{fib}^\mathtt{b}$ and $\mathtt{fib}^\mathtt{c}$.

In the example, apart from the parameter n, two more variables need to be taken into account: the elements $\mathtt{s}_0$ and $\mathtt{s}_1$ on the stack (whose maximum height is 2). The rule $\mathsf{fib}_2(\mathtt{n}, \mathtt{s}_0)$ models the behavior of Block 2, which is traversed if the top of the stack is not 0 (as stated by $\mathsf{guard}(\mathsf{ifne}(\mathtt{s}_0))$) and, after executing $\mathtt{BC}(\mathtt{Block}_2)$, can be followed by either Block 3 or Block 4 (this is made visible by the term $(\mathtt{fib}_3(\mathtt{n}', \mathtt{s}_0', \mathtt{s}_1') \ ; \ \mathtt{fib}_4(\mathtt{n}', \mathtt{s}_0', \mathtt{s}_1'))$), depending on the comparison in the guards at the beginning of the two blocks.

2.3 Size Analysis

The following step consists of inferring *size relations* between the states at different program points. Concretely, we infer size relations between the input variables occurring at the head of a rule and the those occurring in block or method calls in its body. This is done by performing a bottom-up fixpoint computation. In general, various measures can be used to determine the *size* of an input term, possibly affecting the precision of the result. Among the most used size measures, our system is able to handle (i) *integer_value* for numeric variables (i.e., the size of x is its value); and (ii) *path_length* [19] for pointers (i.e., the size of x is the length of the longest *pointer* chain starting from x). As we will see later, the *size* of a variable is a piece of information which is essential in estimating the cost of programs [13].

In the running example, *integer_value* is used as the size measure, and the inferred size relations are shown in the central part of Figure 1. As an example, the third size relation is derived from the second rule of the recursive representation: the relations $\{\mathtt{s}_0 = 0, \mathtt{n} = \mathtt{n}'\}$ mean that $\mathtt{s}_0$ is 0 when entering Block 1, as required by the guard, and n is not modified inside the block.

As for *path_length* analysis, our analyzer does not support the analysis of arbitrary programs yet; in particular, the program is supposed to satisfy some correctness conditions [19]: (1) data structures are not cyclic; and (2) whenever a reference is passed to a method, it is guaranteed that the corresponding structure (on the heap) is not updated by that method. In order to overcome these limitations, we should enrich our analyzer by *Sharing* and *Cyclicity* components [19]. This is the subject of ongoing work.

2.4 Relevant Variables

The information obtained by means of the previous analysis steps will be used in order to estimate the cost of programs. The problem of solving the cost equations is, in general, very difficult, and the existing tools we used for experimenting our

approach denote several limitations as regards the class of problems which can be dealt with. Therefore, our purpose is trying to make things simpler by applying additional transformations to the results obtained so far.

As a first step, we note that, in many cases, the stack is only used to load a piece of data contained in a non-stack variable and perform a comparison; afterwards, the stack location is emptied without any modifications to the loaded data. In this case, the stack variable is only used as a temporary data-keeper, and its use ends just after the comparison; detecting such situations is often possible, and leads to unify the non-stack variable with the stack variable in order to eliminate the latter from the relations [5]. At line 1 of the example, a comparison to 0 is executed just after loading `n` on the stack. It is clear, therefore, that the variable $\mathtt{s}_0$ in the recursive representation can be replaced by `n` in the guard, thus leading to `guard(ifeq(n))`. In `fib`, this optimization allows to get rid of $\mathtt{s}_0$ and $\mathtt{s}_1$ in all relations; `n` comes to be the only variable which needs to be taken into account.

Moreover, it is important to identify the set of variables which are *relevant* to the cost, i.e., whose value may influence the execution time of the program. As an example, the index of a *for*-like loop is usually relevant since it affects the number of iterations; on the other hand, a variable which is used to store partial results has no effects in the cost, unless its value takes part in computations whose execution time is not fixed. Relevant variables turn out to be those which are involved in guards or method calls, since (i) a guard affects the control flow of a program and, therefore, its execution time; and (ii) the cost of executing external methods can be clearly relevant to the overall cost. This analysis is similar, in its purpose, to *program slicing* [20]; it is performed by propagating backwards through the control flow graph variables which are found to be relevant. In the end, when a fixpoint is reached, every block is labeled with the sets of input and output relevant variables which will be used to produce cost relations. The use of slicing in `fib` does not lead to the elimination of any variables from the relations, since, after the optimization described above, there is only one variable, `n`, which is clearly relevant to the cost. However, in the other examples (see Section 3.1), several variables can be eliminated, thus leading to a simpler form for cost relations which could not be solved otherwise.

2.5 Cost Relations

From the recursive representation, the size relations and the relevant variables, we automatically yield as output the Cost relation which defines the cost of the procedure by means of a set of *cost equations*. Intuitively, for each rule $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G}, \mathtt{B}, (\mathtt{q}_1; \ldots; \mathtt{q}_n)$ associated to the block id, where `G` is the guard and `B` the bytecode instructions, 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).

Therefore, each rule in the recursive representation is associated at least to one cost equation (when there is no disjunction). For instance, the rule defining $\mathtt{fib}^{\mathtt{a}}$ is used to generate $C_{\mathtt{fib}}$. Since `fib` contains a disjunction in its body, then a

continuation is generated. This continuation CC_0 has as many alternatives as calls in the disjunction, i.e., two. Each of them is labeled by the corresponding guard, which determines its applicability. Thus, equation C_1 (resp. C_2) only may be applicable if `n` is equal to 0 (resp. different from 0). Note that the guards in the equations are extracted from the guards in the rules in the recursive representation. Size relations are also taken into account in each one of the cost equations not corresponding to continuations. For instance, equation C_4, associated to the rule $\texttt{fib}_4$, makes use of the last two size relations in Figure 1, which relate $\texttt{fib}_4$ with $\texttt{fib}^{\texttt{b}}$ and $\texttt{fib}^{\texttt{c}}$ respectively. The application of such size relations allows the generation of the corresponding calls $C_{\texttt{fib}}(\texttt{n}-1)$ and $C_{\texttt{fib}}(\texttt{n}-2)$, respectively. Note that the cost relations are parametric w.r.t. the *cost model* (in the figure, we use T_b to denote the cost of the bytecode block b).

3 Cost Analysis for Recursive procedures

In this section, we infer the cost of two classical recursive procedures. In both cases, and in general for recursive procedures whose base case depends on constant values, the cost relations obtained by our analysis are directly solvable by Mathematica. For simplicity, in the following the cost of all bytecode instructions is assumed to be 1; using a more refined cost model which assigns different costs to different bytecodes would not introduce further complications. For readability, we present only the original Java code, instead of the bytecode.

3.1 The Classical Hanoi Towers

The first example corresponds to the classical algorithm of the Hanoi Towers, which is depicted in the table below; the call hanoi(7, 1, 2, 3) moves 7 disks from tower 1 to tower 3 using the auxiliary tower 2. The recurrence equations obtained by the analyzer are depicted in the same table. The equation hanoi[n] corresponds to the total cost of a call to hanoi, where n is the first argument of the method. The other equations correspond to the cost of the different blocks in the control flow graph; they are obtained directly from the corresponding recursive representation. For example, the equation m0[n] corresponds to verifying the condition n>0; here, 2 is the cost of the corresponding bytecodes used in the comparison. The equation m3[0] corresponds to the base-case (when n≤0), and m3[n] corresponds to executing the *then* branch; the constant 15 is the cost of the corresponding bytecodes, and the two occurrences of hanoi[n−1] are the cost of the recursive calls. The fact that n decreases by 1 in the recursive calls was detected by size analysis of the bytecode program. Note that the local variables, and stack elements, which do not appear in the equations were removed by the slicing algorithm (Section 2.4), since they do not affect the base-case condition; therefore, they are not relevant for the cost.

Once the equations have been generated, we solve them in Mathematica by calling its recurrence equation solver RSolve. The query RSolve[{eqns}, {a[n], . . . , z[y]}, {n, . . . , y}] solves a set of recurrence equations {eqns} for a[n], . . . , z[y], where n, . . . , y are the only variables, by giving solutions for a, . . . , z as pure functions. The full Mathematica query is shown in the table. We are able to solve the above equations without any preprocessing, and, as expected, the obtained answer predicts an

Cost relations and Mathematica solution for Hanoi

```
static void hanoi(int n,int s,int a,int t) {
 if (n > 0) {
   hanoi(n-1, s, t, a);
   System.out.println(n+":"+s+"→"+t);
   hanoi(n-1, a, t, s); } }
```

$$\mathcal{E}_{hanoi} = \begin{cases} \text{hanoi[n]} == \text{m0[n]}, \\ \text{m0[n]} == 2 + \text{m3[n]}, \\ \text{m3[0]} == 1, \\ \text{m3[n]} == \text{m4[n]}, \\ \text{m4[n]} == 15 + \text{hanoi[n-1]} + \text{hanoi[n-1]} \end{cases}$$

Mathematica **query:** RSolve[$\{\mathcal{E}_{hanoi}\}$, {hanoi[n],m0[n],m3[n],m4[n]},n]

Mathematica **answer (complexity):** hanoi[n] → (-17) + 5 2^{2+n}

Fig. 3. The Hanoi Problem

Cost relations and Mathematica solution for Fibonacci

```
static int fib(int n){
  if ((n==0) || (n==1))   return 1;
   else return (fib(n-1)+fib(n-2));
}
```

$$\mathcal{E}_{fib} = \begin{cases} \text{fib[n]} == \text{m0[n]}, \\ \text{m0[n]} == 2 + \text{m4[n]}, \\ \text{m4[0]} == 2, \\ \text{m4[n]} == \text{m5[n]}, \\ \text{m5[n]} == 3 + \text{m6[n]}, \\ \text{m6[1]} == 2, \\ \text{m6[n]} == \text{m7[n]}, \\ \text{m7[n]} == 10 + \text{fib[n-1]} + \text{fib[n-2]} \end{cases}$$

Mathematica **query:** RSolve[$\{\mathcal{E}_{fib}\}$, {fib[n],m0[n],m4[n],m6[n],m7[n]},n]

Mathematica **answer (complexity):** fib[n] →-(2^{3-n} (15 2^{1+n} - 19 (1 - $\sqrt{5}$)^n + 5 $\sqrt{5}$ (1 - $\sqrt{5}$)^n - 19 (1 + $\sqrt{5}$)^n - 5 $\sqrt{5}$ (1 + $\sqrt{5}$)^n)) / ((-1 + $\sqrt{5}$)^2 (1 + $\sqrt{5}$)^2)

Fig. 4. The Fibonacci Problem

exponential complexity for hanoi[n].

3.2 Recursive Fibonacci

The next example (Fig. 4) is a recursive implementation of the Fibonacci number series, already studied in Sec. 2. The recurrence equations obtained by the analyzer are depicted in the same table. The equation fib[n] corresponds to the total cost of a call fib(n). The other equations correspond to the different blocks in the control flow graph. For example, m4[0] and m4[n] correspond to the success and failure of the condition n == 0, respectively. Similarly, m6[1] and m6[n] corresponds to n == 1. The equation m7[n] corresponds to the cost of the recursive calls and their corresponding bytecodes; the decreasing by 1 and 2 in the calls was detected by size analysis on the bytecode. Moreover, irrelevant stack elements were removed from the equations by means of slicing. Solving the above equations in Mathematica gives the expected exponential complexity.

4 Analyzing Programs with Arrays and (Nested) Loops

In this section, we assess the practicality of the cost analysis for several procedures dealing with arrays and loops. We start by an example for array reversal, whose cost relations are solvable in Mathematica. Then, we study array concatenation, which requires some transformations over the cost relation in order to make it solvable. Finally, we analyze a method for matrix multiplication with several nested loops, which can be solved by means of a different query for each loop.

Cost relations and Mathematica **solution for Array Reversal**	

static int[ ] reverse(int[ ] a){
int la = a.length;
int[ ] r = new int[la];
for (int i=la ; i > 0 ; i--) r[la-i]=a[i-1];
return r;
}
	reverse[a] == m0[a],
	m0[a] == 8 + m1[a],
	m1[i] == 2 + m2[i],
	m2[0] == 2,
	m2[i] == m4[i],
	m4[i] == 12 + m1[i-1]

Mathematica **Query:** RSolve[{ rev[a] == m0[a], m0[a] == 8 + m1[a-1], m1[a] == 2 + m2[a], m2[0] == 2, m2[a] == m4[a], m4[a] == 12 + m1[a-1]}, {rev[a],m0[a],m1[a],m2[a],m4[a]}]	
Mathematica **Answer:** reverse[a] − > 12 (1 + 2 a)	

Fig. 5. Array Reversal

4.1 Reverse of an Array

We want to infer the cost of a simple reverse method which reverses the elements of an array. The recursive representation of reverse in our system takes the form reverse(a, i, r), where a represents the input array, i is the local variable and r is the resulting array. Basically, the execution time depends on the number of loop iterations; therefore, relevant variables are those appearing in the guard of the recurrence relation for m2 (which denotes the termination condition of the loop). Only a and i appear in the cost relation yielded by our system, while r is removed. The size analysis abstracts the array a to its *length* and infers that the variable i decreases by one unit in each iteration.

In order to solve the recurrence equations in Mathematica, we need to use the same variable name in all equations, i.e., we cannot have both a and i. This is because, otherwise, Mathematica requires all variables to be passed from the initial equation on (see also Sec. 4.2). Note that this renaming can be easily done in an automatic way (the result can be seen in the RSolve query).

4.2 Concatenation of Two Arrays

Consider the method concat in Fig. 6: it concatenates two input arrays a and b and returns the result in c. The equation concat[a, b] corresponds to the cost of calling concat with two arrays with length a and b, and m0[a, b] corresponds to the initialization of the local variables. The loops correspond respectively to the equations: (1) m1[a, b, i], m2[a, b, i] and m4[a, b, i]; and (2) m3[a, b, i] m5[a, b, i], m7[a, b, i] and m8[a, b, i].

The size analysis was able to infer the increase in the loops' counters and their corresponding initial values; slicing removed the variable r, which is irrelevant to the cost. The major limitations we found in Mathematica are:

1) it is impossible to include guards in the recurrence equations;

2) variables cannot be repeated in the equation head;

3) all equations must have at least one variable argument;

4) variables in the equation head must appear in the body.

Regarding limitation 1), we can notice in the equations for m2 that recursion ends when i = a. Therefore, we could write the two equations for m2 as fol-

Cost relations and Mathematica solution for Array Concatenation

```
static int[ ] concat(int a[ ], int b[ ]) {
  int l1 = a.length;
  int l2 = b.length;
  int[ ] r = new int[l1+l2];
  int i = 0;
  for (i=0;i<l1;i++) r[i]=a[i];
  for (i=l1;i<l1+l2;i++) r[i]=b[i];
  return r;
}
```

```
concat[a,b] == m0[a,b],
m0[a,b] == 15 + m1[a,b,0],
m1[a,b,i] == 3 + m2[a,b,i],
m2[a,b,i] == m3[a,b,i],          i ≥ a
m2[a,b,i] == m4[a,b,i],          i < a
m3[a,b,i] == 2 + m5[a,b,b],
m4[a,b,i] == 8 + m1[a,b,i+1],
m5[a,b,i] == 5 + m7[a,b,i],
m7[a,b,i] == 2,                  i ≥ a+b
m7[a,b,i] == m8[a,b,i],          i < a+b
m8[a,b,i] == 8 + m5[a,b,i+1],
```

Mathematica **queries:**

```
RSolve[{ m1[i] == 3 + m2[i], m2[a] == 2 + k, m2[i] == m4[i],
         m4[i] == 8 + m1[i+1] }, {m1[i],m2[i],m4[i]},i]
RSolve[{ m5[i] == 5 + m7[i], m7[a+b] == 2, m7[i] == m8[i],
         m8[i] == 8 + m5[i+1]}, {m5[i],m7[i],m8[i]},i]
```

Mathematica **answers:** m1[i] -> 5 + 11 a - 11 i + k (k ->m5[b]) m5[i] -> 7 + 13 a + 13 b - 13 i

Solution (composition of the answers):

concat[a,b] -> 15 + m1[0] ≡ 15 + 5 + 11 a + m5[b] ≡ 27 + 24 a

Fig. 6. Array Concatenation

lows: $\mathsf{m2}[\mathsf{a},\mathsf{b},\mathsf{a}] == \mathsf{m3}[\mathsf{a},\mathsf{b},\mathsf{a}], \mathsf{m2}[\mathsf{a},\mathsf{b},\mathsf{i}] == \mathsf{m3}[\mathsf{a},\mathsf{b},\mathsf{i}]$. The same process can be applied to the equations for m7, which can be transformed to $\mathsf{m7}[\mathsf{a},\mathsf{b},\mathsf{a}+\mathsf{b}] == 2, \mathsf{m7}[\mathsf{a},\mathsf{b},\mathsf{i}] == \mathsf{m8}[\mathsf{a},\mathsf{b},\mathsf{i}]$. This reformulation is still not acceptable by Mathematica, because there are repeated variables in the head of the rules (point 2). Yet, we observe that the first two arguments of the relation, a and b (i.e., the array lengths), remain constant through the relation. Therefore, we can safely (and automatically) remove them from *all* the equations. However, this transformation incurs problems 3) and 4). Problem 3 appears because the first two equations do not have variables anymore; this prevents us from including them in the Mathematica query (rather, we can use them only at the end, to compose the final solution). Furthermore, when i is initialized to the length of the array b in the equation m3, i.e., we have $\mathsf{m3}[\mathsf{i}] == \mathsf{m5}[\mathsf{b}]$, problem 4) occurs. In order to overcome problem 4) (which will indeed appear frequently), we treat $\mathsf{m5}[\mathsf{b}]$ as a constant (k is used in the table) and replace it in all the equations. This involves the execution of two different queries in Mathematica, as it can be seen above: one for $\mathsf{m1}[\mathsf{i}]$, and one for $\mathsf{m5}[\mathsf{i}]$. The final complexity is obtained by composing the results (taking into account that $\mathsf{k} = \mathsf{m5}[\mathsf{b}]$) with the initial equations, which have no variables.

We want to point out that, although the above transformations could be done automatically (and we could produce recurrence relations which are directly solvable in Mathematica), we have not implemented them in our system because we are still studying which solver is more appropriate for our needs. Indeed, Mathematica is a rather complex software which offers much more than is needed in order to solve recurrence equations; therefore, we might want to process the output of our system with a simpler software, like PURRS [7], which is indeed dedicated to solve recurrence equations.

Cost relations and Mathematica solution for Matrix Multiplication

```
static int[ ][ ] mult(int[ ][ ] a,int[ ][ ] b,
                      int r, int c) {
  int[ ][ ] c1 = new int[r][c];
  for(int i=0; i < r;i++)
    for( int j =0; j < c; j++)
      for (int k=0; k < c; k++)
        c1[i][j] = c1[i][j] + (a[i][k] *a[k][j]);
  return c1;
}
```

```
mult[r,c] == 16 + m0[r,c,0],
m0[r,c,i] == 3 + m1[r,c,i],
m1[r,c,i] == 0                    i ≥ r
m1[r,c,i] == m2[r,c,i]            i < r
m2[r,c,i] == 4 + m3[r,c,0] + m0[r,c,i+1]
m3[r,c,j] == 3 + m4[r,c,j],
m4[r,c,j] == 0,                   j ≥ c
m4[r,c,j] == m5[r,c,j],           j < c
m5[r,c,j] == 4 + m6[r,c,0] + m3[r,c,j+1]
m6[r,c,k] == 3 + m7[r,c,k],
m7[r,c,k] == 0,                   k ≥ c
m7[r,c,k] == m8[r,c,k],           k < c
m8[r,c,k] == 24 + m6[r,c,k+1]
```

Mathematica **queries:**

```
RSolve[{m0[i] == 3 + m1[i], m1[r] == 0, m1[i] == m2[i],
        m2[i] == 4 + k + m0[i+1]}, {m0[i],m1[i],m2[i]},i]

RSolve[{m3[j] == 3 + m4[j], m4[c] == 0, m4[j] == m5[j],
        m5[j] == 4 + z + m3[j+1]},{m3[j],m4[j],m5[j]},j]

RSolve[{m6[k] == 3 + m7[k], m7[c] == 0, m7[k] == m8[k],
        m8[k] == 24 + m6[k+1]}{m6[k],m7[k],m8[k]},k]
```

Mathematica **answers:**

```
m1[i] -> 3 - 7 i - i k + 7 r + k r        (k = m3[0])
m3[j] -> 3 + 7 c - 7 j + c z - j z        (z = m6[0])
m6[k] -> 3 (1 + 9 c - 9 k)
```

Solution: mul-> $16+m1[0] \equiv 19+7r+rm3[0] \equiv 19+7r+r(3+7c+cm6[0]) \equiv 19+10r+10rc+27c^2r$

Fig. 7. Matrix multiplication

4.3 Matrix Multiplication

Consider the method mult in Fig. 7, which implements the multiplication of (a subset of) two matrices. The first two arguments are the matrices to be multiplied, and r and c are the number of rows and columns to be taken into account. As a novel feature, mult presents nested loops. This requires a special processing of the CFG (see [4] for more details), which detects and extracts loops.

The equations $m0[r,c,i]$, $m1[r,c,i]$ and $m2[r,c,i]$ correspond to the outermost loop; $m3[r,c,j]$, $m4[r,c,j]$ and $m5[r,c,j]$ corresponds to the middle loop; and $m6[r,c,k]$, $m7[r,c,k]$ and $m8[r,c,k]$ correspond to the innermost loop. Note that size analysis was able to infer the increase of the loops' counters, and that slicing was able to remove variables which are irrelevant to the cost.

The inferred recurrence equations are not solvable by Mathematica. We basically need to apply the same transformations explained in Sect. 4.2 to make the equations solvable (and overcome the previously mentioned limitations). Very briefly, we first simplify all guards by applying them to the equation heads. Then, we remove parameters f and c from the equations, since they are constant in all of them. Finally, we input three separate queries to Mathematica, one for each loop. In the end, the results obtained for the three loops are composed in the initial equation (we could not include it in the query as it has no arguments).

5 Dealing with Object-Oriented Features

In this section, we study several object-oriented features. First, we see how we deal with dynamic dispatching in the context of cost analysis. Then, we analyze the cost of reversing a list implemented as a class with field attributes. Finally, we infer the cost of a linear search algorithm over the list. To the best of our knowledge, these examples illustrate novel object-oriented features which are not studied in existing cost analyses for other languages and paradigms.

5.1 Dynamic dispatching

The Incr example in Fig. 8, taken from [5], presents interesting object-oriented features, such as the use of objects and the invocation of methods with dynamic dispatching. In particular, as it is not known at compile time which of the three methods (A.inc, B.inc or C.inc) will be executed, we need to consider the different costs obtained for each case. Therefore, the *object* o which determines which method will be executed becomes part of the guards in the cost relation. It can be seen in the equation for m4 that, depending on whether the object o belongs to class A, B, or C, we have a different cost. We can apply all the transformations discussed in Sect. 4 in order to make the equations solvable in Mathematica (i.e., apply the guards for i, eliminate variable n from all equations, etc). However, we cannot apply the guards which distinguish the type of the object to the equation head. Our proposal consists of generating three different sets of recurrence equations (one corresponding to each method invocation). We can now get rid of variable o in all sets of equations. This leads to the three Mathematica queries written in the table. We named the result for each one as add_X, where X is the type of object for which the cost was computed. As the Mathematica answer is rather large for add_B and add_C, we did not written the constant parts in the table. Then, depending on whether one is interested in upper or lower bounds of the computational cost, we compute the maximum or the minimum of the three solutions: clearly, add_A provides an upper bound and add_C a lower bound of the computational cost.

5.2 List Processing Algorithms

The class List (Fig. 9) contains a procedure which computes the reverse of a list implemented as a class with two fields: next, which points to the next element in the list, and data, which contains the information stored in the list. The equations inferred by the analyzer are depicted in the table. Recall that, in the recurrence equations, x stands for the length of paths reachable from x, as explained in Section 2.3. The size analysis was able to infer that the path length of x is decreasing by one in every two consecutive visits of the loop, and that slicing was able to remove all variables that do not affect the loop condition. The output recurrence equations can be directly solved in Mathematica. We obtained linear complexity as it is shown in the table.

Finally, the last example Search (Fig. 10) implements the linear search of an element e in an input list x. It uses the List class, and returns the element of x whose data field is equal to e. The novel feature of this example is that we have two

Cost relations and Mathematica solution for Dynamic Dispatching

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

```
add[n,o]   == m0[n,o],
m0[n,o]    == 4 + m1[n,o,0],
m1[n,o,i]  == 3 + m2[n,o,i],
m2[n,o,i]  == 2,                    i > n
m2[n,o,i]  == m3[n,o,i],            i ≤ n
m3[n,o,i]  == 7 + m4[n,o,i],
m4[n,o,i]  == A:incr[i] + m5[n,o,i+1], o ∈ A
m4[n,o,i]  == B:incr[i] + m5[n,o,i+2], o ∈ B
m4[n,o,i]  == C:incr[i] + m5[n,o,i+3], o ∈ C
m5[n,o,i]  == 2 + m1[n,o,i],
A:incr[i]  == 3,
B:incr[i]  == 3,
C:incr[i]  == 3,
```

Mathematica **query:**

```
RSolve[{m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == A + m5[i+1], m5[i] == 2 + m1[i], A[i] == 3 },
        {m1[i],m2[i],m3[i],m4[i],m5[i],A[i]},i]

RSolve[{m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == B[i] + m5[i+2], m5[i] == 2 + m1[i], B[i] == 3},
        {m1[i],m2[i],m3[i],m4[i],m5[i],B[i]},i]

RSolve[{ m1[i] == 3 + m2[i], m2[n] == 2, m2[i] == m3[i], m3[i] == 7 + m4[i],
        m4[i] == C[i] + m5[i+3], m5[i] == 2 + m1[i], C[i] == 3},
        {m1[i],m2[i],m3[i],m4[i],m5[i],C[i]},i]
```

Appr. of Mathematica answers: $\mathsf{add}_A \approx 15n + K$ $\mathsf{add}_B \approx 7.5n + K$ $\mathsf{add}_C \approx 5n + K$

Fig. 8. The `Incr` program

Cost relations and Mathematica solution for List Reversal

```
class List {
  List next;  int data;
  public List reverse(List x) {
   List result = null;  List tmp = null;
   while ( x != null ) {
    tmp = x.next;  x.next = result;
    result = x;  x = tmp;
  }
   return result; }}
```

```
reverse[x] == m0[x],
m0[x]      == 4 + m1[x],
m1[x]      == 2 + m2[x],
m2[0]      == 2,
m2[x]      == m4[x],
m4[x]      == 11 + m1[x-1],
```

Mathematica **query:**

```
RSolve[{rev[x] == m0[x], m0[x] == 4 + m1[x], m1[x] == 2 + m2[x],
        m2[0] == 2, m2[x] == m4[x], m4[x] == 11 + m1[x-1]},
        {rev[x],m0[x],m1[x],m2[x],m4[x]},x]
```

Mathematica **answer (complexity):** rev[x] -> 8 + 19 x

Fig. 9. List reversal

conditions on the loop, and the second one depends on the content of the list. From the recurrence equations, we observe that the equations m8 correspond to the first guard in the loop condition. In particular, the first one is the exit condition of the loop when the list is null, i.e., $x = 0$. The second one, $x \neq 0$, leads to the equations n1, where the second condition is evaluated. Variable d in this guard represents x.data. Exiting form the loop depends on whether d is equal to e. Mathematica cannot handle these recurrence equations, due to the fact that they involve two guards (and one should consider the best and the worst case). Besides, it is not

Cost relations and Mathematica solution for List Manipulation

```
class Search {
  public List search(List x, int e) {
  int index=1;
   while ( x != null && x.data != e ) {
      index++;
      x = x.next;
   }
    return x;
}
```

```
search[x,e] == m5[x,e],
m5[x,e] == 7 + m6[x,e] + m7[c],  c ≤ x
m6[x,e] == 2 + m8[x,e],
m8[0,e] == 0,
m8[x,e] == m9[x,e],
m9[x,e] == 4 + n1[x,e,d],
n1[x,e,d] == 0,                  d = e
n1[x,e,d] == n0[x,e,d],          d ≠ e
n0[x,e,d] == 5 + m6[x-1,e],
m7[c] == 2
```

Mathematica **query:**

```
RSolve[{ search[x] == m5[x], m5[x] == 9 + m6[x], m6[x] == 2 + m8[x], m8[0] == 0,
    m8[x] == m9[x], m9[x] == 4 + n1[x], n1[x] == n0[x], n0[x] == 5 + m6[x-1]},
        {search[x],m5[x],m6[x],m8[x],m9[x],n0[x],n1[x]},{x}]
```

Mathematica **answer (upper bound complexity):** search[x] -> 11 (1+x)

Fig. 10. List Manipulation

Benchmark	BC	CFG	RR	Size An.	Slicing	Cost	Total
Hanoi	289	15	5	150	15	3	187
Fibonacci	298	19	6	265	39	2	331
Reverse	296	21	5	207	21	2	256
Concat	351	64	7	648	43	4	766
MatMult	388	182	12	2152	115	5	2465
Incr	320	38	13	956	371	7	1383
List	355	27	4	123	58	3	216
Search	351	51	12	462	220	4	750
Diff	377	167	14	3804	595	10	4590
Intersec	390	181	18	4575	869	15	5657
Sum	295	62	8	1415	287	5	1776

Table 1
Measured time (in ms) of the different phases of cost analysis

possible to express the second guard in a way which is understandable to the solver. The approach we propose consists of *approximating* the solution by disregarding the second guard of the loop. This implies that we delete the first equation for n1 from the set of equations, and the remaining guard d ≠ e. As a consequence, variables e and d become now irrelevant and are sliced away. Note that we will obtain an upper bound solution for the computational cost, rather than the exact solution. This reasoning is not easy to automate, and our system still cannot deal with it automatically. Besides, it should be noted that, in order to solve the equations in Mathematica, we need to unfold m7 in order to eliminate the guard of m5. After all these (non trivial) simplifications, Mathematica provides a linear complexity as the upper bound.

6 Experiments and Discussion

In order to assess the practicality of our cost analysis framework, we have implemented a prototype analyzer in Ciao [9]. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9.

Table 1 shows the run-times of the different phases of the cost analysis process. The first column, **Benchmark**, indicates the name of the class and method of the benchmark to be analyzed. The second column, **BC**, contains the size in bytes of the corresponding .class. All other columns show execution times in milliseconds and have been obtained using the statistics/2 procedure of `Ciao` with the parameter runtime. They are computed as the arithmetic mean of five runs. For each benchmark, **CFG** represents the time taken to build the control flow graph of the corresponding method; **RR** is the time taken for obtaining the recursive representation from the CFG (this includes translating bytecode operations for converting stack positions into local variables and performing the transformation outlined in Sec. 2.4); **Size An.** is the time taken by the abstract-interpretation based size analysis for computing size relations; **Slicing** shows the time required for detecting the set of variables which are relevant in each block of the CFG; finally, **Cost** stands for the time taken to build the cost relations for the different blocks.

The benchmarks are divided into four categories, as it can be seen from the structure of the table: (i) recursive procedures (Sec. 3) solving Hanoi and Fibonacci problems; (ii) methods involving (possibly nested) loops, as array reverse and concatenation, and matrix multiplication (Sec. 4); (iii) procedures manipulating objects and fields (Sec. 5), as the add method involving dynamic dispatching, and list reversal and search; (iv) further examples: computing the difference (diff) and the intersection (intersec) of two arrays, and the function sum computing $\Sigma_{i=1}^{n} \Sigma_{j=1}^{i} i+j$.

As the figure shows, the total times obtained using our prototype implementation range from 187 ms in the case of Hanoi, to 5657 ms in the case of Intersec. As it can be seen, computing size relations is the most expensive step. This comes from the fact that this step requires a global analysis of the program, whereas **CFG**, **RR**, and **Cost** basically involve a single pass on the code. **Slicing** also requires a global, though much simpler, analysis. Thus, the time it requires is the biggest after the size analysis.

Our experimental results are very preliminary, and there is still plenty of room for optimization (mainly in the size analysis phase). The main planned optimization is the use of abstract compilation techniques in order to avoid re-computation of abstract operations which are related to the bytecodes. This can be done since the analysis is denotational, so that those bytecodes will always have the same abstract approximations.

As regards the accuracy of the analysis, our approach was able to obtain accurate cost relations for all the considered benchmarks. Note that this is an important observation, since we are confident that, by further transformations on the cost relations, or by using a more powerful system for solving recurrence equations, we will be able to obtain closed form solutions for a broader class of programs.

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

awarded by MEC.

References

[1] Mathematica: the Way the World Calculates.
`http://www.wolfram.com/products/mathematica/index.html`.

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

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

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Automatic Cost Analysis of Java Bytecode. Technical Report CLIP10/2006.0, Technical University of Madrid, School of Computer Science, UPM, December 2006.

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007. To appear.

[6] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[7] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver.
`http://www.cs.unipr.it/purrs/`.

[8] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

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

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

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

[12] 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.

[13] C.S. Lee, N.D. Jones, and A.M. Ben-Ammar. The size-change principle for program termination. In *Proc. POPL*. ACM, 2001.

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

[15] Á. Rebón Portillo, K. Hammond, H-W. Loidl, and P. Vasconcelos. Cost Analysis Using Automatic Size and Time Inference. In *IFL*, volume 2670 of *LNCS*. Springer, 2002.

[16] 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.

[17] M. Rosendhal. Automatic Complexity Analysis. In *Proc. FPCA*. ACM, 1989.

[18] F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005. Available at `www.sci.univr.it/~spoto/papers.html`.

[19] F. Spoto, P. M. Hill, and E. Payet. Path-length analysis for object-oriented programs. In *Proc. EAAI*, 2006.

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

[21] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

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