Verification of Java Bytecode using Analysis and Transformation of Logic Programs

Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, German Puebla

Verification of Java Bytecode using Analysis and Transformation of Logic Programs

E. Albert[1], M. Gómez-Zamalloa[1], L. Hubert[2], and G. Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain
{elvira,mzamalloa,laurent,german}@clip.dia.fi.upm.es

Abstract. State of the art analyzers in the Logic Programming (LP) paradigm are nowadays mature and sophisticated. They allow inferring a wide variety of global properties including termination, bounds on resource consumption, etc. The aim of this work is to automatically transfer the power of such analysis tools for LP to the analysis and verification of Java bytecode (JVML). In order to achieve our goal, we rely on well-known techniques for meta-programming and program specialization. More precisely, we propose to partially evaluate a JVML interpreter implemented in LP together with (an LP representation of) a JVML program and then analyze the residual program. Interestingly, at least for the examples we have studied, our approach produces very simple LP representations of the original JVML programs. This can be seen as a decompilation from JVML to high-level LP source. By reasoning about such residual programs, we can automatically prove in the `CiaoPP` system some non-trivial properties of JVML programs such as termination, run-time error freeness and infer bounds on its resource consumption. We are not aware of any other system which is able to verify such advanced properties of Java bytecode.

1 Introduction

Verifying programs in the (Constraint) Logic Programming paradigm —(C)LP— offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. The work presented in this paper is motivated by the existence of *abstract interpretation*-based analyzers [3] which infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus, obtaining safe approximations of programs behavior. These analyzers are parametric w.r.t. the so-called abstract domain, which provides a finite representation of possibly infinite sets of values. Different domains capture different properties of the program with different levels of precision and at different computational costs. This includes error freeness, data structure shape (like pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as procedure-level properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost), etc. `CiaoPP` [9] is the *abstract interpretation*-based preprocessor of the `Ciao` (C)LP system, where analysis results have been applied to perform high- and low-level optimizations and *program verification*.

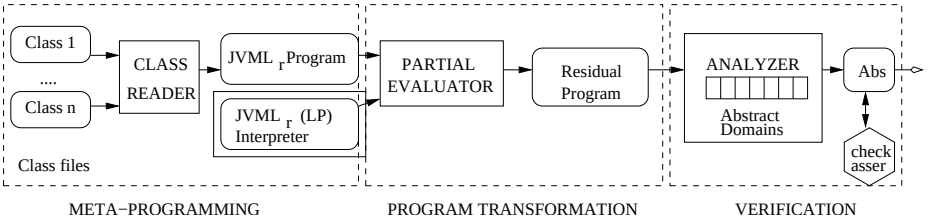

Fig. 1. Verification of Java Bytecode using Logic Programming Tools

A principal advantage of verifying programs on the (LP) *source* code level is that we can infer complex global properties (like the aforementioned ones) for them. However, in certain applications like within the context of mobile code, one may only have the *object* code available. In general, analysis tools for such low-level languages are unavoidably more complicated than for high-level languages because they have to cope with complicated and unstructured control flow. Furthermore, as the JVML (Java Virtual Machine Language, i.e., Java bytecode) is a stack-based language, stacks cells are used to store intermediate values, and therefore their type can change from one assignment to another, and they can also be used to store 32 bits of a 64 bit value, which make the inference of stack information much more difficult. Besides, it is a non trivial task to specify/infer global properties for the bytecode by using pre- and post-conditions (as it is usually done in existing tools for high-level languages).

The aim of this work is to provide a practical framework for the verification of JVML which exploits the expressiveness, automation and genericity of the advanced analysis tools for LP source. In order to achieve this goal, we will focus on the techniques of meta-programming, program specialization and static analysis that together support the use of LP tools to analyze JVML programs. Interpretative approaches which rely on CLP tools have been applied to analyze rather restricted versions of high-level imperative languages [13] and also assembly code for PIC [8], an 8-bit microprocessor. However, to the best of our knowledge, this is the first time the interpretative approach has been successfully applied to a general purpose, realistic, imperative programming language.

Overview. Fig. 1 presents a general overview of our approach. We depict an element within a straight box to denote its use as a program and a rounded box for data. The whole verification process is split in three main parts:

1. *Meta-programming.* We use LP as a language for representing and manipulating JVML programs. We have implemented an automatic translator, called CLASS_READER, which given a set of `.class` files {`Class 1`,..., `Class n`} returns P, an LP representation of them in JVML$_r$ (a representative subset of JVML presented in Sect. 2). Furthermore, we also describe in Sect. 3 an interpreter in LP, called JVML$_r$_INT, which captures the JVM semantics. The interpreter has been extended in order to compute *execution traces*, which will be very useful for reasoning about certain properties.

2. *Partial evaluation.* The development of partial evaluation techniques [10] has allowed the so-called "interpretative approach" to compilation which consists in specializing an interpreter w.r.t. a fixed object code. We have used an existing PARTIAL_EVALUATOR for LP in order to specialize the JVML$_r$_INT w.r.t. P. As a result, we obtain I_P, an LP residual program which can be seen as a decompiled and translated version of P into LP (see Sect. 4).
3. *Verification of Java bytecode.* The final goal is that the JVML program can be verified by analyzing the residual program I_P obtained in Step 2) above by using state-of-the-art ANALYZERs developed for LP, as we will see in Sect. 5.

The resulting scheme has been implemented and incorporated in the `CiaoPP` preprocessor. Our preliminary experiments show that it is possible to infer global properties of the computation of the residual LP programs. We believe our proposed approach is very promising in order to bring the analysis power of declarative languages to low-level, imperative code such as Java bytecode.

2 The Class Reader (JVML to JVML$_r$ in LP)

As notation, we use *Prog* to denote LP programs and *Class* to denote `.class` files (i.e., JVML classes). The input of our verification process is a set of `.class` files, denoted as $C_1 \ldots C_n \in Class$, as specified by the Java Virtual Machine Specification [12]. Then, the CLASS_READER takes $C_1 \ldots C_n$ and returns an LP file which contains all the information in $C_1 \ldots C_n$ represented in our JVML$_r$ language. JVML$_r$ is a representative subset of the JVML language which is able to handle: classes, interfaces, arrays, objects, constructors, exceptions, method call to class and instance methods, etc. For simplicity, some other features such as packages, concurrency and types as float, double, long and string are left out of the chosen subset. For conciseness, we use JVML$_r$_*Prog* to make it explicit that an LP program contains a JVML$_r$ representation. The differences between JVML and JVML$_r$ are essentially the following:

1. *Bytecode factorization.* Some instructions in JVML have a similar behavior and have been factorized in JVML$_r$ in order to have fewer instructions[3]. This makes the JVML$_r$ code easier to read (as well as the traces which will be discussed in Sect. 3) and the JVML$_r$_INT easier to program and maintain.
2. *References resolution.* The original JVML instructions contain indexes onto the *constant-pool* table [12], a structure present in the `.class` file which stores different kinds of data (constants, field and method names, descriptors, class names, etc.) and which is used in order to make bytecode programs as compact as possible. The CLASS_READER removes all references to the constant-pool table in the bytecode instructions by replacing them with the complete information to facilitate the task of the tools which need to handle the bytecode later.

[3] This allows covering over 200 instructions of JVML in 54 instructions in JVML$_r$.

```
1 class(
2   className(packageName(''),shortClassName('Rational')),final(false),public(true),
3   abstract(false),className(packageName('java/lang/'),shortClassName('Object')),[],
4   [field(
5      fieldSignature(
6        fieldName(
7          className(packageName(''),shortClassName('Rational')),shortFieldName(num)),
8        primitiveType(int)),
9      final(false),static(false),public,initialValue(undef)),
10   field(
11     fieldSignature(
12       fieldName(
13         className(packageName(''),shortClassName('Rational')),shortFieldName(den)),
14         primitiveType(int)),
15     final(false),static(false),public,initialValue(undef))],
16  [method(
17     methodSignature(
18       methodName(
19         className(packageName(''),shortClassName('Rational')),shortMethodName('<init>')),
20       [primitiveType(int),primitiveType(int)],none),
21     bytecodeMethod(3,2,0,methodId('Rational_class',1),[]),
22     final(false),static(false),public),
23   method(
24     methodSignature(
25       methodName(
26         className(packageName(''),shortClassName('Rational')),shortMethodName(exp)),
27       [primitiveType(int)],
28       refType(classType(className(packageName(''),shortClassName('Rational'))))),
29      bytecodeMethod(4,4,0,methodId('Rational_class',2),[]),
30      final(false),static(false),public),
31   method(
32     methodSignature(
33       methodName(
34         className(packageName(''),shortClassName('Rational')),shortMethodName(expMain)),
35       [primitiveType(int),primitiveType(int),primitiveType(int)],
36       refType(classType(className(packageName(''),shortClassName('Rational'))))),
37     bytecodeMethod(3,4,0,methodId('Rational_class',3),[]),
38     final(false),static(true),public)]).
```

Fig. 2. Extract of the Program Fact Describing the Rational Class of Running Example

The `Ciao` file generated by the CLASS_READER contains the bytecode instructions for all methods in $C_1 \dots C_n$, represented as a set of facts; and also, a single fact obtained by putting together all the other information available in the `.class` files (class name, methods and fields signatures, etc.).

Example 1 (running example). Our running example considers a main Java class named `Rational` which represents rational numbers using two attributes: `num` and `den`. The class has a constructor, an instance method `exp` for computing the exponential of rational numbers w.r.t. a given exponent (the result is returned on a new rational object), and a static method `expMain` which given three integers, creates a new rational object using the first two ones as numerator and denominator, respectively, and invokes its `exp` method using the third argument as parameter. Finally, it returns the corresponding rational object. This example features arithmetic operations, object creation, field access, and invocation of both class and instance methods. It also shows that our approach is not restricted to intra-procedural analysis.

In Fig. 2, we show the extract of the program fact corresponding to class `Rational`. Line numbers are provided for convenience but they are not part of the

```
bytecode(0,2,'Rational',const(primitiveType(int),1),1).
bytecode(1,2,'Rational',istore(2),1).
bytecode(2,2,'Rational',const(primitiveType(int),1),1).
bytecode(3,2,'Rational',istore(3),1).
bytecode(4,2,'Rational',iload(1),1).
bytecode(5,2,'Rational',if0(leInt,23),3).
bytecode(8,2,'Rational',iload(2),1).
bytecode(9,2,'Rational',aload(0),1).
bytecode(10,2,'Rational',getfield(fieldSignature(
        fieldName(className(packageName(''),shortClassName('Rational')),shortFieldName(num)),
        primitiveType(int))),3).
bytecode(13,2,'Rational',ibinop(mulInt),1).
bytecode(14,2,'Rational',istore(2),1).
bytecode(15,2,'Rational',iload(3),1).
bytecode(16,2,'Rational',aload(0),1).
bytecode(17,2,'Rational',getfield(fieldSignature(
        fieldName(className(packageName(''),shortClassName('Rational')),shortFieldName(den)),
        primitiveType(int))),3).
bytecode(20,2,'Rational',ibinop(mulInt),1).
bytecode(21,2,'Rational',istore(3),1).
bytecode(22,2,'Rational',iinc(1,-1),3).
bytecode(25,2,'Rational',goto(-21),3).
bytecode(28,2,'Rational',new(className(packageName(''),shortClassName('Rational'))),3).
bytecode(31,2,'Rational',dup,1).
bytecode(32,2,'Rational',iload(2),1).
bytecode(33,2,'Rational',iload(3),1).
bytecode(34,2,'Rational',invokespecial(methodSignature(
        methodName(
           className(packageName(''),shortClassName('Rational')),
           shortMethodName('<init>')),
        [primitiveType(int),primitiveType(int)],none)),3).
bytecode(37,2,'Rational',areturn,1).

bytecode(0,3,'Rational',new(className(packageName(''),shortClassName('Rational'))),3).
bytecode(3,3,'Rational',dup,1).
bytecode(4,3,'Rational',iload(0),1).
bytecode(5,3,'Rational',iload(1),1).
bytecode(6,3,'Rational',invokespecial(methodSignature(
        methodName(
           className(packageName(''),shortClassName('Rational')),
           shortMethodName('<init>')),
        [primitiveType(int),primitiveType(int)],none)),3).
bytecode(9,3,'Rational',iload(2),1).
bytecode(10,3,'Rational',invokevirtual(methodSignature(
        methodName(
           className(packageName(''),shortClassName('Rational')),
           shortMethodName(exp)),
        [primitiveType(int)],
        refType(classType(className(packageName(''),shortClassName('Rational')))))),3).
bytecode(13,3,'Rational',areturn,1).
```

Fig. 3. Extract of the Bytecode facts of our Running Example

code. The description of the field `num` appears in Lines 4-9, `den` in L.10-15 and the methods in L.16-38. For conciseness, only methods actually used are shown. The first method (L.16-22) is a constructor that takes two integers (L.20) as arguments. The second method (L.23-30) is named `exp` (L.26), it is an instance method (cf. `static(false)` L.30)) and takes an integer (L.27) as a parameter and returns an instance of `Rational` (L.28). Finally, the last method (L.31-38), `expMain`, is a class method (cf. `static(true)` L.38), that takes as parameters three integers (L.35) and returns an instance of `Rational` (L.36).

Fig. 3 presents the bytecode facts corresponding to the methods `exp` and `expMain`. Each fact is of the form `bytecode(PC,MethodID,Class,Inst,Size)`, where `Class` and `MethodID`, respectively, identify the class and the method to which the instruction `Inst` belongs. `PC` corresponds to the program counter and `Size` to the number of bytes of the instruction in order to be able to compute the next value of the program counter. The class method number 3 (i.e., `expMain`) creates first an instance of `Rational` (Instructions 0-6) and then invokes the instance method `exp` (I.9-10). The bytecode of the method number 2 (i.e., `exp`), can be divided in 3 parts. First, the initialization (I.0-3) of two local variables, say x_2 and x_3, to 1. Then, the loop body (I.4-25) first compares the exponent to 0 and, if it is less or equal to 0, exits the loop by jumping 23 bytes ahead (I.4-5). Then, the current value of x_2 (`iload`) and the denominator (`aload` and `getfield`) are retrieved (I.8-10), multiplied and stored in x_2 (I.13-14). The same is done for x_3 with the numerator in I.15-21. Finally, the value of the exponent is decreased by one (I.22) and PC is decreased by 21 (I.25) i.e., we jump back to the beginning of the loop. After the loop, the method creates an instance of `Rational`, stores the result (I.28-34), and returns this object (I.37).

3 Specification of the Dynamic Semantics

(C)LP programs have been used traditionally for expressing the semantics of both high- and low-level languages [13,17]. In our approach, we express the JVML semantics in `Ciao`. The formal JVML specification chosen for our work is Bicolano [14], which is written with the Coq Proof Assistant [1]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.

In the specification, a state is modeled by a 3-tuple[4] $\langle\ Heap, Frame, StackFrame\ \rangle$ which represents the machine's state where $Heap$ represents the contents of the heap, $Frame$ represents the execution state of the current *Method* and, $StackFrame$ is a list of frames corresponding to the call stack. Each frame is of the form $\langle\ Method, PC, OperandStack, LocalVar\ \rangle$ and contains the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point PC of the method $Method$. The definition of the dynamic semantics is based on the notion of *step*.

Definition 1 (*step* $\xrightarrow{L}_P$)**.** *The dynamic semantics of each instruction is specified as a partial function step* : JVML$_r$_*Prog* $\times$ $State_{JVM} \rightarrow State_{JVM} \times Step\_Name$ *that, given a program* $P \in$ JVML$_r$_*Prog and a state* $S \in State_{JVM}$*, computes the next state* $S' \in State_{JVM}$ *and returns the name of the step* $L \in Step\_Name$*. For convenience, we write* $S \xrightarrow{L}_P S'$ *to denote* $step(P, S) = (S', L)$*.*

In order to formally define our interpreter, we need to define the following function which iterates over the steps of the program until obtaining a final state.

[4] Both in Bicolano and in our implementation there is another kind of state for exceptions, but we have omitted it from this formalization for the sake of simplicity.

Definition 2 ($\xrightarrow{T}{}^{*}_{P}$)**.** *Let* $\xrightarrow{T}{}^{*}_{P}$ *be a relation on* $State_{JVM}$ *with* $S \xrightarrow{T}{}^{*}_{P} S'$ *iff:*

- *there exists a sequence of steps* L_1 *to* L_n *such that* $S \xrightarrow{L_1}_P \ldots \xrightarrow{L_n}_P S'$,
- *there is no state* $S'' \in State_{JVM}$ *such that* $S' \xrightarrow{L}_P S''$, *and*
- $T \in Traces$ *such that* $T = [L_1, \ldots, L_n]$ *is the list of the names of the steps.*

We can now define a general interpreter which takes as parameters a program and a *method invocation specification* (MIS in the following) that indicates: 1) the method the execution should start from, 2) the corresponding effective parameters of the method which will often contain logical variables or partially instantiated terms (and should be interpreted as the set of all their instances) and 3) an initial heap. The interpreter relies on an EXECUTE function that takes as parameters a program $P \in \text{JVML}_r\_Prog$ and a state $S \in State_{JVM}$ and returns (S', T) where $S \xrightarrow{T}{}^{*}_{P} S'$.

The following definition of JVML_r_INT computes, in addition to the return value of the method called, also the trace which captures the computation history. Traces represent the semantic steps used and therefore do not only represent instructions, as the context has also some importance. They allow us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. E.g., `invokevirtual_step_ok` and `invokevirtual_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception.

Definition 3 (JVML_r_INT)**.** *Let* M *be a* MIS *that contains a method signature, the parameters for the method and a heap, written as* $M \in$ MIS*. We define a general interpreter* $\text{JVML}_r\text{\_INT}(P, M) = (R, T)$ *with*

- $S = initialState(P, M)$*, where function initialState builds, from the program* P *and the* MIS M*, a state* $S \in State_{JVM}$*,*
- $\text{EXECUTE}(P, S) = (S', T)$ *and*
- $R = result\_of(S')$ *is the result of the execution of the method specified by* M *(the value on top of the stack of the current frame of* S'*).*

This definition of JVML_r_INT returns the trace and the result of the method but it is straightforward to modify the definitions of JVML_r_INT and EXECUTE to return less information or to add more. This gives more flexibility to our interpretative approach when compared to direct compilation: for example, if needed, we can return in an additional argument a list containing the information about each state which we would like to *observe* in order to prove properties which may require a deeper inspection of execution states.

4 Automatic Generation of Residual Programs

Partial evaluation (PE) [10] is a semantics-based program optimization technique which has been deeply investigated within different programming paradigms. The main purpose of PE is to specialize a given program w.r.t. the *static data*,

i.e., the part of its input data which is known—hence it is also known as *program specialization*. The partially evaluated (or residual) program will be (hopefully) executed more efficiently since those computations that depend only on the static data are performed once and for all at PE time. We use the partial evaluator for LP programs of [15] which is part of `CiaoPP`. Here, we represent it as a function PARTIAL_EVALUATOR: $Prog \times Data \rightarrow Prog$ which, for a given program $P \in Prog$ and static data $S \in Data$, returns a residual program $P_S \in Prog$ which is a *specialization* [10] of P w.r.t. S.

The development of PE, program specialization and related techniques [6,10,7] has led to an alternative approach to compilation (known as the first Futamura projection) based on specializing an interpreter with respect to a fixed object program. The success of the application of the technique involves eliminating the overhead of parsing the program, fetching instructions, etc., and leading to a residual program whose operations mimic those of the object program. This can also be seen as a translation of the object program into another programming language, in our case `Ciao`. The *residual* program is ready now to be, for instance, efficiently executed in such language or, as in our case, accurately analyzed by tools for the language in which it has been translated. The application of this interpretative approach to compilation within our framework consists in partially evaluating the JVML_r_INT w.r.t. $P = \text{CLASS\_READER}(C_1, \ldots, C_n)$ and a MIS.

Definition 4 (LP residual program). *Let* JVML_r_INT $\in Prog$ *be a* JVML_r *interpreter,* $M \in$ MIS *and* $C_1, \ldots, C_n \in Class$ *be a set of classes. The* LP residual program*,* I_P*, for* JVML_r_INT *w.r.t.* $C_1, \ldots, C_n$ *and* M *is defined as* $I_P = \text{PARTIAL\_EVALUATOR}(\text{JVML}_r\text{\_INT}, (\text{CLASS\_READER}(C_1, \ldots, C_n), M))$.

Note that, instead of using the interpretative approach, we could have implemented a compiler from Java bytecode to LP. However, we believe that the interpretative approach has at least the following advantages: 1) more flexible, in the sense that it is easy to modify the interpreter in order to observe new properties of interest, see Sect. 3, 2) easier to trust, in the sense that it is rather difficult to prove (or trust) that the compiler preserves the program semantics and, it is also complicated to explicitly specify what the semantics used is, 3) easier to maintain, new changes in the JVM semantics can be easily reflected in the interpreter by modifying (or adding) a proper "step" definition, and 4) easier to implement, provided a powerful partial evaluator for LP is available.

Example 2 (residual programs). We now want to partially evaluate our implementation of the interpreter which does not output the trace (see Sect. 3) w.r.t. the bytecode method `expMain` in Ex. 1, an empty heap and three free variables as parameters. The size of the program to be partially evaluated (i.e., interpreter) is 86,326 bytes (2,240 lines) while the size of the data (i.e., bytecode representation) is 16,677 bytes (101 lines) of JVML_r. The partial evaluator has different options for tuning the level of specialization. For this example, we have used local and global control strategies based on *homeomorphic embedding* (see [11]).

```
expMain(A,B,C,ref(loc(2)),heap([[num(int(A)),num(int(B))],
            [num(int(1)),num(int(1))]])) :- C=<0 .
expMain(A,B,C,ref(loc(2)),heap([[num(int(A)),num(int(B))],
            [num(int(A)),num(int(B))]])) :- C>0, F is C-1, F=<0 .
expMain(A,B,C,D,E) :- C>0, H is C-1, H>0, I is A*A,
            J is B*B, K is H-1, execute(A,B,K,I,J,E,D) .

execute(A,B,C,D,E,heap([[num(int(A)),num(int(B))],
            [num(int(D)),num(int(E))]]),ref(loc(2))) :- C=<0 .
execute(A,B,C,D,E,G,L) :- C>0, N is D*A, O is E*B, P is C-1,
            execute(A,B,P,N,O,G,L) .
```

Fig. 4. Residual Exponential Program without Trace

We show in Fig. 4 the residual program resulting of such automatic PE. The parameters `A`, `B` and `C` of `expMain/5` represent the numerator, denominator and exponent, respectively. The fourth and fifth parameters represent, respectively, the top of the stack and the heap where the method result (i.e., an object of type `Rational` in the bytecode) will be returned. In particular, the result corresponds to the second element, `ref(loc(2))`, in the heap. Note that this object is represented in our LP program as a list of two atoms, the first one corresponds to attribute `num` and the second one to `den`. The first two rules for `expMain/5` are the base cases for exponents $\texttt{C} = 0$ and $\texttt{C} = 1$, respectively. The third rule, for $\texttt{C} > 1$, uses an auxiliary recursive predicate `execute/6` which computes $\texttt{A}^{\texttt{C}+1}$ and $\texttt{B}^{\texttt{C}+1}$ and returns the result in the second element of the heap. It should be noted that our PE tool has done a very good job by transforming a rather large interpreter into a small residual program (where all the interpretation overhead has been removed). The most relevant point to notice about the residual program is that we have converted low level jumps into a recursive behavior and achieved a very satisfactory translation from the Java bytecode method `expMain`. Indeed, it is not very different from the `Ciao` version one could have written by hand, provided that we need to store the result in the fifth argument of predicate `expMain/5` as an object in the heap, using the corresponding syntax.

While the above LP program can be of a lot of interest when reasoning about functional properties of the code, it is also of great importance to augment the interpreter with an additional argument which computes a trace (see Def. 3) in order to capture the computation history. The residual program which computes execution traces is `expMain/4`, which on success contains in the fourth argument the execution trace at the level of Java bytecode (rather than the top of the stack and the heap). Below, we show the recursive rule of predicate `execute/8` whose last argument represents the trace (and corresponds to the second rule of `execute/7` without trace in Fig. 4):

```
execute(B,C,D,E,F,G,I,[goto_step_ok,iload_step,if0_step_continue,
        iload_step,aload_step_ok,getfield_step_ok,ibinop_step_ok,
        istore_step_ok,iload_step,aload_step_ok,getfield_step_ok,
        ibinop_step_ok,istore_step_ok,iinc_step|H]) :-
    D>0, I is E*B, J is F*C, K is D-1, execute(B,C,K,I,J,G,I,H) .
```

As we will see in the next section, this trace will allow observing a good number of interesting properties about the program.

5 Verification of Java Bytecode Using LP Analysis Tools

Having obtained an LP representation of a Java bytecode program, the next task is to use existing analysis tools for LP in order to infer and verify properties about the original bytecode program. We now recall some basic notions on *abstract interpretation* [3]. *Abstract interpretation* provides a general formal framework for computing safe approximations of program behaviour. In this framework, programs are interpreted using *abstract values* instead of *concrete values*. An abstract value is a finite representation of a, possibly infinite, set of concrete values in the concrete domain D. The set of all possible abstract values constitutes the *abstract domain*, denoted D_α, which is usually a complete lattice or cpo which is ascending chain finite. Abstract values and sets of concrete values are related by an *abstraction* function $\alpha : 2^D \to D_\alpha$, and a *concretization* function $\gamma : D_\alpha \to 2^D$. The concrete and abstract domains must be related in such a way that the following condition holds [3]: $\forall x \in 2^D : \ \gamma(\alpha(x)) \supseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. In general, the comparison in D_α, written $\sqsubseteq$, is induced by $\subseteq$ and α.

We rely on a generic analysis algorithm (in the style of [9]) defined as a function ANALYZER: $Prog \times AAtom \times ADom \to AApprox$ which takes a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and a set of abstract atoms $S_\alpha \in AAtom$ which are descriptions of the entries (or calling modes) into the program and returns $Approx_\alpha \in AApprox$. Correctness of analysis ensures that $Approx_\alpha$ safely approximates the semantics of P. We denote that S_α and $Approx_\alpha$ are abstract semantic values in D_α by using the same subscript α.

In order to verify the program, the user has to provide the intended semantics $Assert_\alpha$ (or specification) as a semantic value in D_α in terms of *assertions* (these are linguistic constructions which allow expressing properties of programs) [16]. This intended semantics embodies the requirements as an expression of the user's expectations. The *verifier* has to compare the (actual) inferred semantics $Approx_\alpha$ w.r.t. $Assert_\alpha$. We use the *abstract interpretation*-based verifier integrated in `CiaoPP`. It is dealt here as a function AI_VERIFIER: $Prog \times AAtom \times ADom \times AAssert \to boolean$ which for a given program $P \in Prog$, a set of abstract atoms $S_\alpha \in AAtom$, an abstract domain $D_\alpha \in ADom$ and an intended semantics $Assert_\alpha$ in D_α succeeds if the approximation computed by ANALYZER(P, S_α, D_α)=$Approx_\alpha$ entails that P satisfies $Assert_\alpha$, i.e., $Approx_\alpha \sqsubseteq Assert_\alpha$.

Definition 5 (verified bytecode). *Let $I_P \in Prog$ be an LP residual program for* JVML$_r$_INT *w.r.t. $C_1, \ldots, C_n \in Class$ and $M \in$* MIS *(see Def. 3). Let $D_\alpha \in ADom$ be an abstract domain, $S_\alpha \in AAtom$ be a set of abstract atoms and $Assert_\alpha \in D_\alpha$ be the abstract intended semantics of I_P. We say that $(C_1, \ldots, C_n, M)$ is verified w.r.t. $Assert_\alpha$ in $ADom$ if* AI_VERIFIER$(I_P, S_\alpha, D_\alpha, Assert_\alpha)$ *succeeds.*

In principle, any of the considerable number of abstract domains developed for *abstract interpretation* of logic programs can be applied to residual programs, as

well as to any other program. In addition, arguably, analysis of logic programs is inherently simpler than that of Java bytecode since the bytecode programs decompiled into logic programs no longer contain an operand stack for arithmetic and execution flow is transformed from jumps (since loops in the Java program are compiled into conditional and unconditional jumps) into recursion.

5.1 Run-Time Error Freeness Analysis

The use of objects in Ex. 1 could in principle issue exceptions of type `NullPointerException`. Clearly, the execution of the `expMain` method will not produce any exception, as the unique object used is created within the method. However, the JVM is unaware of this and has to perform the corresponding run-time test. We illustrate that by using our approach we can statically verify that the previous code cannot issue such an exception (nor any other kind of run-time error).

First, we proceed to specify in `Ciao` the property "`goodtrace`" which encodes the fact that a bytecode program is run-time error free in the sense that its execution does not issue `NullPointerException` nor any other kind of run-time error (e.g., `ArrayIndexOutOfBoundsException`, etc). As this property is not predefined in `Ciao`, we declare it as a regular type using the `regtype` declarations in `CiaoPP`. Formally, we define this property as a *regular unary logic* program, see [5]. The following regular type `goodtrace` defines this notion of safety for our example (for conciseness, we omit the bytecode instructions which do not appear in our program):

```
:- regtype goodtrace/1.
goodtrace(T) :- list(T,goodstep).

:- regtype goodstep/1.
goodstep(iinc_step).          goodstep(aload_step_ok).        goodstep(invokevirtual_step_ok).
goodstep(iload_step).         goodstep(if0_step_jump).        goodstep(invokestatic_step_ok).
goodstep(normal_end).         goodstep(const_step_ok).        goodstep(if0_step_continue).
goodstep(new_step_ok).        goodstep(return_step_ok).       goodstep(if_icmp_step_jump).
goodstep(pop_step_ok).        goodstep(astore_step_ok).       goodstep(putfield_step_ok).
goodstep(dup_step_ok).        goodstep(istore_step_ok).       goodstep(getfield_step_ok).
goodstep(goto_step_ok).       goodstep(ibinop_step_ok).       goodstep(if_icmp_step_continue).
goodstep(areturn_step_ok).    goodstep(invokespecial_step_here_ok).
```

Next, the version with traces of the residual program in Fig. 4 is extended with the following assertions:

```
:- entry expMain(Num,Den,Exp,Trace):(num(Num),num(Den),num(Exp),var(Trace)).
:- check success expMain(Num,Den,Exp,Trace) => goodtrace(Trace).
```

The entry assertion describes the valid external queries to predicate `expMain/4`, where the first three parameters are of type `num` and the fourth one is a variable. We use the "`success`" assertion as a way to provide a partial specification of the program. It should be interpreted as: for all calls to `expMain(Num,Den,Exp,Trace)`, if the call succeeds, then `Trace` must be a `goodtrace`.

Finally, we use `CiaoPP` to perform regular type analysis using the *eterms* domain [18]. This allows computing safe approximations of the success states of all predicates. After this, `CiaoPP` performs compile-time checking of the `success` assertion above, comparing it with the assertions inferred by the analysis, and produces as output the following assertion:

```
:- checked success expMain(Num,Den,Exp,Trace) => goodtrace(Trace).
```

Thus, the provided assertion has been *validated* (marked as `checked`).

5.2 Cost Analysis and Termination

As mentioned before, *abstract interpretation*-based program analysis techniques allow inferring very rich information including also resource-related issues. For example, CiaoPP can compute upper and lower bounds on the number of execution steps required by the computation [9,4]. Such bounds are expressed as functions on the sizes of the input arguments. Various metrics are used for the "size" of an input, such as list-length, term-size, term-depth, integer-value, etc. Types, modes, and size measures are first automatically inferred by the analyzers and then used in the size and cost analysis.

Let us illustrate the cost analysis in CiaoPP on our running example. We consider a slightly modified version of the residual program in Fig. 4 in which we have eliminated the accumulating parameter due to a current limitation of the cost analysis in CiaoPP. The cost analysis can then infer the following property of the recursive predicate `execute/5` (and a similar one of `expMain/4`) using the same entry assertion as in Sect. 5.1:

```
:- true pred execute(A,B,C,D,E): (num(A),num(B),num(C),var(D),var(E))
        => ( num(A), num(B), num(C), num(D), num(E),
             size_ub(A,int(A)), size_ub(B,int(B)), size_ub(C,int(C)),
             size_ub(D,expMain(int(A),int(C)+1)+int(A)),
             size_ub(E,expMain(int(B),int(C)+1)+int(B)) )
         + steps_ub(int(C)+1).
```

which states that `execute/5` is called in this program with the first three parameters being of type `num` (i.e., bound to numbers) and two variables. The part of the assertion after the `=>` symbol indicates that on success of the predicate all five parameters are bound to numbers. This is used by the cost analysis in order to set the integer-value as size-metric for all five arguments. The first three arguments are input to the procedure and thus their size (value) is fixed. The last two arguments are output and their size (value) is a function on the value of (some of) the first three arguments. The upper bound computed by the analysis for D (i.e., the fourth argument) is $A^{C+1} + A$. Note that this is a correct upper bound, though the most accurate one is indeed A^{C+1}. A similar situation occurs with the upper bound for the fifth argument (E). Finally, the part of the assertion after the `+` symbol indicates that an upper bound on the number of execution steps is $C + 1$, which corresponds to a linear algorithmic complexity. This is indeed the most accurate upper bound possible, since predicate `execute/5` is called $C+1$ times until C becomes zero. Note that, in this case, we do not mean the number of JVM steps in Def. 1, but the number of computational steps.

CiaoPP's termination analysis relies on the cost analysis described in the previous section. In particular, it is able to prove termination of a program provided it obtains a non-infinite upper bound of its cost. Following the example of Sect. 5.2, CiaoPP is able to turn into `checked` status the following assertion (and the similar one for `expMain/4`): "`:- check comp execute(A,B,C,D,E) + terminates`". which ensures that the execution of the recursive predicate always terminates w.r.t. the previous entry.

6 Experiments and Discussion

We have implemented and performed a preliminary experimental evaluation of our framework within the `CiaoPP` preprocessor [9], where we have available a partial evaluator and a generic analysis engine with a good number of abstract domains, including the ones illustrated in the previous section. Our interpretative approach has required the implementation in `Ciao` of two new packages: the CLASS_READER (1141 lines of code) which parses the `.class` files into `Ciao` and the JVML_r_INT interpreter for the JVML_r (3216 lines). These tools, together with a collection of examples, are available at: `http://cliplab.org/Systems/jvm-by-pe`.

Table 1 studies two crucial points for the practicality of our proposal: the size of the residual program and the relative efficiency of the full transformation+analysis process. As mentioned before, the algorithms are parametric w.r.t. the abstract domain. In our experiments we use *eterms*, an abstract domain based on regular types, that is very useful for reasoning about functional properties of the code, run-time errors, etc., which are crucial aspects for the safety of the Java bytecode. The system is implemented in Ciao 1.13 [2] with compilation to WAM bytecode. 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.

The input "program" to be partially evaluated is the JVML_r_INT interpreter in all the examples. Then, the first group of columns **Bytecode** shows information about the input "data" to the partial evaluator, i.e., about the `.class` files. The columns **Class** and **Size** show the names of the classes used for the experiments and their sizes in bytes, respectively. The second column **Method** refers to the name of the method within each class which is going to form the MIS, i.e., to be the starting point for PE and context-sensitive program analysis. We use a set of classical algorithms as benchmarks. The first 9 methods belong to programs with iterations and static methods but without object-oriented features, where **mod**, **fact**, **gcd** and **lcm**, compute respectively the modulo, factorial, greatest-common-divisor and least-common-multiple (two versions); the **Combinatory** class has different methods for computing the number of selections of subsets given a set of elements for every ordering/repetition combination. The next two benchmarks, **LinearSearch** and **BinarySearch**, deal with arrays and correspond to the classic linear and binary search algorithms. Finally, the last four benchmarks correspond to programs which make extensive use of object-oriented features such as instance method invocation, field accessing and setting, object creation and initialization, etc.

The information about the "output" of the PE process appears in the second group of columns, **Residual**. The columns **Size** and **NUnfs** show the size in bytes of each residual program and the number of unfolding steps performed by the partial evaluator to generate it, respectively. We can observe that the partial evaluator has done a good job in all examples by transforming a rather large interpreter (86,326 bytes) in relatively small residual programs. The sizes range from 317 bytes for **m2** (99.4% reduction) to 4.911 for **Lcm2** (83.6 %). The number of required unfolding steps explains the high PE times, as we discuss below. A relevant point to note is that, for most programs, the size of the LP

Bytecode			Residual		Times (ms)			
Class	**Size**	**Method**	**Size**	**NUnfs**	**Trans**	**PE**	**Ana**	**Total**
Mod	314	mod	956	1645	18	1244	59	1322
Fact	324	fact	1007	1537	19	1432	74	1525
Gcd	265	gcd	940	1273	18	1160	125	1303
Lcm	299	lcm	2260	4025	21	5832	817	6670
Lcm2	547	lcm2	4911	3724	26	3963	1185	5174
Combinatory	703	varNoRep	1314	1503	32	1837	87	1955
Combinatory	703	combNoRep	2177	2491	34	3676	150	3860
Combinatory	703	combRep	2151	3033	29	5331	950	6310
Combinatory	703	perm	1022	1256	29	1234	65	1328
LinearSearch	318	search	3114	8832	22	45228	296	45546
BinarySearch	412	search	3670	14117	23	72945	313	73282
Np	387	m2	317	527	20	502	12	534
ExpFact	890	main	2266	8353	35	23773	95	23903
Rational	559	expMain	3131	6613	31	13692	16	13739
Date	602	forward	11046	26982	36	80960	218	81213

Table 1. Sizes of residual programs and transformation and analysis times

translation is larger than the original bytecode. This can be justified by the fact that the resulting program does not only represent the bytecode program but it also makes explicit some internal machinery of the JVM. This is the case, for instance, of the exception handling. As there are no `Ciao` exceptions in the residual program, the implicit exceptions in JVML have been made explicit in LP. Furthermore, the Java bytecode has been designed to be really compact, while the LP version has been designed to be easier to read by human beings and contains type information that must be inferred on the JVML. It should not be difficult to reduce the size of the residual bytecode if so required by, for example, simply using short identifiers.

The final part of the table provides the times for performing the transformations and the analysis process. Execution times are given in milliseconds and measure *runtime*. They are computed as the arithmetic mean of five runs. For each benchmark, **Trans**, **PE** and **Ana** are the times for executing the CLASS_READER, the partial evaluator and the analyzer, respectively. The column **Total** accumulates all the previous times. We can observe that most of the time is due to the partial evaluation phase (and this time is directly related to the number of unfolding steps performed). This is to be expected because the specialization of a large program (i.e., the interpreter) requires to perform many unfolding steps in all the examples (ranging from 14.117 steps for **search** in **BinarySearch** to 527 for **m2**), plus many additional generalization steps which are not shown in the table. The analysis time is then relatively low, as the residual programs to be analyzed are significantly smaller than the program to be partially evaluated.

As for future work, we plan to obtain accurate bounds on resource consumption by considering the traces that the residual program contains and the concrete cost of each bytecode instruction. Also, we are in the process of studying the scalability of our approach to the verification of larger Java bytecode pro-

grams. We also plan to exploit the advanced features of the partial evaluator which integrates abstract interpretation [15] in order to handle recursion.

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 (TIN-2005-09207 *MERIT*), and the Madrid Regional Government (S-0505/TIC/0407 *PROMESAS*). The authors would like to thank David Pichardie and Samir Genaim for useful discussions on the Bicolano JVM specification and on termination analysis, respectively.

References

1. B. Barras et al. The Coq proof assistant reference manual: Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.
2. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla (Eds.). The Ciao System. (v1.13). At `http://clip.dia.fi.upm.es/Software/Ciao/`.
3. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL'77*, pages 238–252, 1977.
4. S. Debray, P. López, M. Hermenegildo, and N. Lin. Estimating the Computational Cost of Logic Programs. *Proc. of SAS'94*, LNCS 864, pp. 255–265. Springer.
5. T. Früwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Proc. LICS'91*, pages 300–309, 1991.
6. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
7. J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.
8. Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.
9. M. Hermenegildo, G. Puebla, F. Bueno, and P. López. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation. *Science of Computer Programming*, 58(1–2):115–140, October 2005.
10. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
11. M. Leuschel. On the power of homeomorphic embedding for online termination. *Proc. of SAS'98*, pages 230–245, 1998. Springer-Verlag.
12. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. A-W, 1996.
13. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of CLP. In *Proc. of SAS'98*, LNCS 1503, pp. 246–261, 1998.
14. D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.
15. G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *Proc. of SAS'06*, LNCS. Springer, 2006. To appear.
16. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for CLP. In *Analysis and Visualization Tools for CP*, pages 23–61. Springer LNCS 1870, 2000.
17. Brian J. Ross. The partial evaluation of imperative programs using prolog. In *META*, pages 341–363, 1988.
18. C. Vaucheret and F. Bueno. More Precise yet Efficient Type Inference for Logic Programs. In *Proc. of SAS'02*, pages 102–116. Springer LNCS 2477, 2002.