Towards Verification of Java Bytecode using Logic Programming Tools

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

[1] Complutense University of Madrid,
elvira@sip.ucm.es
mzamalloa@clip.dia.fi.upm.es
[2] Technical University of Madrid,
{laurent,german}@clip.dia.fi.upm.es

Abstract. State of the art analyzers in the (Constraint) Logic Programming paradigm (or (C)LP for short) are nowadays mature and sophisticated. They allow inferring a wide variety of global properties including termination, run-time error freeness, 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. 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 Java bytecode interpreter implemented in LP w.r.t. (a LP representation of) a set of Java bytecode classes 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 Java bytecode programs. This can be seen as an automatic translation and decompilation from Java bytecode to LP source. Reasoning about properties of such residual programs allows automatically proving some non-trivial properties of Java bytecode programs such as termination and run-time error freeness.

1 Motivation

The technique of abstract interpretation [6] has allowed the development of very sophisticated global static program analyses which are at the same time automatic, provably correct, and practical. The basic idea of abstract interpretation is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus, obtaining safe approximations of programs behavior. A classical application of the semantic approximations produced by an abstract interpreter is to perform program *verification*.

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. 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 a different computational cost. This includes error freeness, data structure shape (like

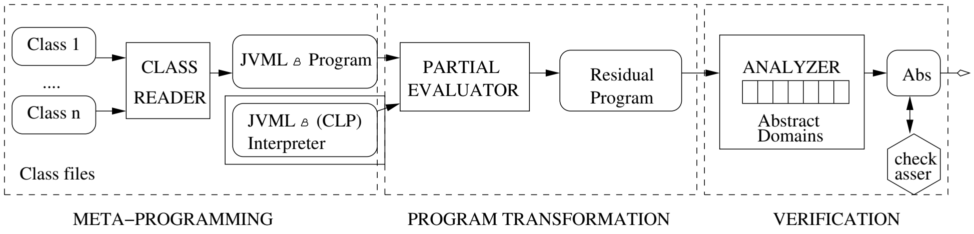

Fig. 1. Verification of Java Bytecode using Logic Programming Tools

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` [11] is the abstract interpretation-based preprocessor of the `Ciao` (C)LP system. It uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs. The semantic approximations thus produced have been applied to perform high- and low-level optimizations and program verification.

A principal advantage of verifying programs on the (LP) *source* code level is that we can infer complex global properties for them. However, in certain applications like within a mobile environment, one may only have the *object* code available, since mobile components are typically deployed as bytecode. In general, it is not immediate 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 such as Java [2]). Also, 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. The aim of this work is to provide a practical framework for the verification of Java bytecode which exploits the expressiveness, automation and genericity of the advanced analysis tools for LP. In order to achieve this goal, we will focus on the techniques of meta-programming, program specialisation and static analysis that together support the use of LP tools to analyse JVML programs (i.e., programs written in the Java Virtual Machine Language). Similar interpretative approaches have been applied to analyse high-level imperative languages [19] and also the PIC processor [10] by relying on CLP tools.

Figure 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 CLASS_READER which, given a set of `.class` files {`Class 1`,..., `Class n`} returns an LP representation of them in JVML$_\beta$ (a representative subset of JVML which will be discussed in detail in Section 1), i.e., a JVML$_\beta$ Program

denoted by P. Furthermore, we have also implemented in LP an interpreter JVML$_\beta$_INT, which captures the JVM semantics. In addition, 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 [13] has allowed the so-called "interpretative approach" to compilation. We have used an existing PARTIAL_EVALUATOR for LP in order to specialize the JVML$_\beta$_INT w.r.t. the LP representation of {`Class 1`,..., `Class n`}, as described in 1). 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.
3. *Verification of Java bytecode.* The final goal is that the JVML program can be verified by analysing the residual program I_P obtained in 2) with state-of-the-art ANALYZERs developed for LP.

The resulting scheme has been incorporated in the `CiaoPP` preprocessor. By means of two running examples, we demonstrate that it can be used to verify non-trivial properties on Java bytecode.

2 The Class Reader (JVML to JVML$_\beta$ in LP)

In this section and the next one we describe (and give some implementation details of) the "meta-programming" phase in Figure 1. In particular, this section presents the elements depicted as CLASS_READER and Section 3 presents JVML$_\beta$_INT.

As notation, we use *Prog* to denote LP programs and *Classes* to denote `.class` files (i.e., JVML classes). The input of our verification process is a set of JVML `.class` files, denoted as $C_1 \ldots C_n \in \mathit{Classes}$, which describe the information of a set of Java classes (as specified by JVML, see the Java Virtual Machine Specification [16]). Then, the process named CLASS_READER in the figure takes $C_1 \ldots C_n$ and returns an LP program which contains all the information available in the classes and represents it in the JVML$_\beta$ language. JVML$_\beta$ is a representative subset of the JVML language which is able to handle: classes, interfaces, arrays, objects, constructors and object initialization, virtual, interface and static invocations, exceptions, method call to class and instance methods, etc. For simplicity, some other features such as packages, types as float, double, long and string, concurrency and tableswitch instructions are left out of the chosen subset. Figure 7 in Appendix A describes in the form of a grammar the formal syntax of JVML$_\beta$. For the sake of conciseness, in the following we use the prefix JVML$_\beta$ on *Prog*, written as JVML$_\beta$_*Prog*, to make explicit that a LP program contains a JVML$_\beta$ representation.

Definition 1 (CLASS_READER). *We define function* CLASS_READER: *Classes*$^+ \rightarrow$ JVML$_\beta$_*Prog which takes a set of* `.class` *files* $C_1 \ldots C_n \in \mathit{Classes}$ *and returns an LP program* $P \in$ JVML$_\beta$_*Prog which is the* LP representation *of* $C_1 \ldots C_n$.

The implementation of CLASS_READER in `Ciao` [4] reads the `.class` files byte by byte and organizes and interprets them as it is specified in *the* `class` *file*

```
public int exp(int base, int exponent){
  int result=1;
  int i=exponent;
  while(i>0){
    result*=base;
    i--;
  }
  return result;
}
```

Fig. 2. Method for Computing the Exponential

format specification (see [16]). As a result, it produces a Ciao program (i.e., an LP program) which consists of a set of facts containing exactly the same information as the original .class files. The differences between JVML and JVML$_\beta$ are essentially the following:

1. *Bytecode factorization.* Some instructions in JVML have similar behavior and have been factorized in JVML$_\beta$ in order to have fewer instructions [3] but without affecting expressiveness. This will make the JVML$_\beta$ code easier to read (as well as the traces which will be discussed in Section 3) and the JVML$_\beta$_INT easier to program and maintain.
2. *References resolution.* Original JVML instructions very often use indexes onto the *constant-pool* table [16], a structure present in the .class file which stores different kinds of data (constants, field and method names and descriptors, class names, etc). The CLASS_READER removes all references to the constant-pool table in the bytecode instructions by replacing them with the complete information. This can be seen as an *unfolding* step whose purpose is to get rid of intermediate steps and which could benefit an analyzer's inference task later.[4] Thus, we no longer need the constant-pool table and all the required data are included within the JVML$_\beta$ class representation.

As a result, CLASS_READER produces in the output, on one hand, the bytecode instructions for the methods in all the involved Java classes and represented as "bytecode" facts. On the other hand, a single fact "program" obtained by putting together the *Class* terms which store all required information (except for the bytecode instructions which appear separately in the above facts). All such facts (bytecode and program) are structured as specified by the JVML$_\beta$ syntax (see Figure 7 in Appendix A for details). Let us see an example.

[3] This allows covering over 200 bytecode instructions of JVML in 54 instructions in JVML$_\beta$.

[4] It should be noted that the PARTIAL_EVALUATOR can automatically perform this unfolding step. But we prefer to have a translator with reference resolution which can be used independently of our current approach (e.g., by a Java bytecode analyzer written in Ciao directly).

Example 1. In Figure 2, we show the Java method `exp` which computes the exponential for the parameters `base` and `exponent`. The execution of CLASS_READER on this example returns the LP program depicted in Figure 3, which contains all the information concerning the class to which `exp` belongs in the JVML$_\beta$ language. Due to space limitations, we only show the bytecode facts which correspond to the method *exp* (and omit the remaining information about the class). Each bytecode fact is of the form *bytecode(ModuleName,Bi,Mi,Inst,L)*, where *Bi* is the index of this instruction in the code array, *Mi* is the index of the actual method, *Instruction* is a term with its "opcode" as functor and its parameters as arguments, and *L* is the instruction length, i.e., the number of bytes it uses in the code array. The *ModuleName* argument represents the class name. This allows us to deal with bytecode instructions which come from different Java classes (since we are considering Java programs as sets of Java classes). It should be noted that there are no indexes to the constant-pool table since, as already mentioned, they have been replaced by the full information. It can also be seen that some original instructions have been replaced by their factorized version (e.g. in the first bytecode fact, `const(primitiveType(int),1))` corresponds in JVML to the `iconst_1` opcode without arguments).

```
bytecode('Exp_class',0,2,const(primitiveType(int),1),1).
bytecode('Exp_class',1,2,istore(3),1).
bytecode('Exp_class',2,2,iload(2),1).
bytecode('Exp_class',3,2,istore(4),2).
bytecode('Exp_class',5,2,goto(10),3).
bytecode('Exp_class',8,2,iload(3),1).
bytecode('Exp_class',9,2,iload(1),1).
bytecode('Exp_class',10,2,ibinop(mulInt),1).
bytecode('Exp_class',11,2,istore(3),1).
bytecode('Exp_class',12,2,iinc(4,-1),3).
bytecode('Exp_class',15,2,iload(4),2).
bytecode('Exp_class',17,2,if0(gtInt,-9),3).
bytecode('Exp_class',20,2,iload(3),1).
bytecode('Exp_class',21,2,ireturn,1).
```

Fig. 3. Partial output of CLASS_READER for Exponential

3 Specification of the Dynamic Semantics

(C)LP programs have been used for expressing the semantics of both high and low-level languages [19, 25]. In our approach, we want to express the JVM semantics in `Ciao`. The formal JVML specification chosen for our work is Bicolano [20], which is a superset[5] of JVML$_\beta$. Bicolano 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.

[5] It also includes the `tableswitch` and `lookupswitch` instructions.

In the specification, a state is modeled by a 3-tuple[6] $\langle\ Heap, Frame, StackFrame\ \rangle$ which represents the machine's state where:

- $Heap$ represents the content of the heap,
- $Frame$ represents the execution state of the current *Method*,
- $StackFrame$ 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 2 **($step \xrightarrow{L}_P$).** *The dynamic semantics of each instruction is specified as a partial function step* : $\text{JVML}_\beta\text{\_}Prog \times State_{JVM} \rightarrow State_{JVM} \times Step\_Names$ *that, given a program* $P \in \text{JVML}_\beta\text{\_}Prog$ *and a state* $S \in State_{JVM}$*, computes the next state* $S' \in State_{JVM}$ *and returns the name of the step* $L \in Step\_Names$*. For convenience, we write* $S \xrightarrow{L}_P S'$ *to denote* $step(P, S) = (S', L)$*.*

The operational semantics of an instruction is expressed differently in the original JVM specification, in Bicolano and in our implementation. The next example shows the different specifications for the `const` instruction, which pushes onto the stack the value of its parameter.

Example 2. The Coq representation in Bicolano of the JVM instruction `const`, which corresponds to the Sun specification showed in Appendix B, is as follows:

```
Inductive step (p:Program) : State.t → State.t  → Prop :=
  | const_step_ok: ∀ h m pc pc' s l sf t z,
    instructionAt m pc = Some (Const t z) →
    next m pc = Some pc' →
    step p (St h (Fr m pc s l) sf)
           (St h (Fr m pc' (Num (I (iconst z))::s) l) sf)
```

The above representation is written in `Ciao` as the program rule:

```
step(const_step_ok, _Program,
     st(H,fr(M,PC,S,L),SF),
     st(H,fr(M,PCb,[num(int(Z))|S],L),SF)):-
   instructionAt(M,PC,const(_T,Z)),
   next(M,PC,PCb).
```

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.

Definition 3 **($\xrightarrow{T}{}^{*}_{P}$).** *Let* $\xrightarrow{T}{}^{*}_{P}$ *be a relation on* $State_{JVM}$ *with* $S \xrightarrow{T}{}^{*}_{P} S'$ *iff:*

[6] There also exists in Bicolano and in our implementation another kind of state that models exceptions, but, to keep this presentation simpler we have omitted it from this formalization

- *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 then define two different interpreters. One that takes as only parameters a program and a list of strings, and starts the execution for the `static void main(java.lang.String[])` method of the first class of the program. This has been implemented, but we have also defined a more general interpreter which takes as parameters a program and a *method invocation specification* that indicates in which method the execution should start from, the corresponding effective parameters (which will often contain logical variables or partially instantiated terms, which should be interpreted as the set of all their instances) of the method and a heap. Both interpreters rely on the following EXECUTE function.

Definition 4 (EXECUTE). *Let* $P \in$ JVML$_\beta$_*Prog be a program to be executed and* $S \in State_{JVM}$ *be a state. We define the execution of this program as* $execute(P, S) = (S', T)$ *with* $S \xrightarrow{T}{}^{*}_{P} S'$.

The following definition of JVML$_\beta$_INT computes, in addition to the return value of the method called, also the trace which captures the computation history. This will allow observing a good number of interesting properties about the program.

Definition 5 (JVML$_\beta$_INT). *Let M be a method invocation specification that contains a method signature, parameters for the method and a heap. We define a general interpreter* JVML$_\beta$_INT$(P, M) = (R, T)$ *with*

- $S = initialState(P, M)$ *where initialState builds a state* $S \in State_{JVM}$ *from the program P and the method invocation specification M,*
- $execute(P, S) = (S', T)$,
- $finalState(S')$, *which checks that* S' *is a valid final state, that is to say that the program counter points to a* **`return`** *instruction and the call stack is empty, 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'*).*

If the state computed by EXECUTE is not a final state, then JVML$_\beta$_INT fails. When we can prove non failure, it means the initial state built from the provided method invocation specification is guaranteed to be consistent.

This definition of JVML$_\beta$_INT returns the trace and the result of the method but it is straightforward to modify the definitions of JVML$_\beta$_INT and EXECUTE (and the corresponding code) to return less information or to add, for example, the list of all the states if needed (to prove properties which may require a deeper inspection of execution states).

4 Automatic Generation of Residual Programs

Partial evaluation (PE) [13] is a semantics-based program optimization technique which has been deeply investigated within different programming paradigms and applied to a wide variety of languages. The main purpose of partial evaluation 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—at partial evaluation time—once and for all. We use the partial evaluator for LP programs of [21] written in `Ciao` and which is part of `CiaoPP`. We represent it here as a function PARTIAL_EVALUATOR: $\mathit{Prog} \times \mathit{Data} \rightarrow \mathit{Prog}$ which, for a given program $P \in \mathit{Prog}$ and static data $S \in \mathit{Data}$, returns a residual program $P_S \in \mathit{Prog}$ which is a *specialization* [13] of P w.r.t. S.

The development of partial evaluation, program specialization and related techniques [8, 14, 13, 9, 3] has led to the now established approach to compilation (known as the first Futamura projection) based on specializing an interpreter with respect to a fixed object program. The overhead of parsing the program, fetching instructions, etc., can often be completely eliminated, 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, executed in such language or, as in our case, analysed by tools for the language in which it has been translated. In the LP context, this interpretative approach has been applied to analyse high-level imperative languages [19] and also the PIC processor [10] by relying on CLP tools.

The application of this interpretative approach to compilation from JVML to LP within our framework consists in partially evaluating the JVML$_\beta$_INT with respect to a method invocation specification M (see Definition 5 above) and a program $P = $ CLASS_READER$(C_1, \ldots, C_n)$. This results in a residual LP program, I_P.

Definition 6 (LP residual program). *Let* JVML$_\beta$_INT $\in \mathit{Prog}$ *be a* JVML$_\beta$ *interpreter, M a method invocation specification and $C_1, \ldots, C_n \in \mathit{Classes}$ be a set of classes. The* LP residual program, *I_P, for* JVML$_\beta$_INT *w.r.t. $C_1, \ldots, C_n$ and M is defined as $I_P =$* PARTIAL_EVALUATOR(JVML$_\beta$_INT, (CLASS_READER$(C_1, \ldots,$ $C_n), M)$)*.*

Note that, alternatively to the interpretative approach, we could have implemented a compiler from Java bytecode to LP. However, the interpretative approach has the advantages that it is simpler to implement, provided that a partial evaluator for LP programs is available, and more flexible in the sense that it is easy to modify the interpreter in order to observe new properties of interest.

Example 3. We show in Figure 4 the result of the automatic partial evaluation of an implementation of the interpreter which does not output the trace (see Section 3) w.r.t. the LP translation of the program in Example 1, an empty

```
:- module( _, [exp/2] ).

exp(args(B,C), A) :-
        C>0, E is B, F is-1+C,
        execute(B,E,A,F) .
exp(args(_1,A), 1 ) :-
        A=<0 .

execute(A,C,E,G) :-
        G>0, H is C*A,  I is-1+G,
        execute(A,H,E,I) .
execute(_A,C,C,D) :-
        D=<0 .
```

Fig. 4. Residual Exponential Program

heap, the signature of the exp method and two variables as parameters. The partial evaluator has different options for tuning the level of specialization. In particular, the so-called local control decides when to stop derivations and the global control when to generalize a new term resulting from a previous unfolding. For this example, we have used the local control strategy based on *homeomorphic embedding* which is described in [21]. For the global control, we have also used homeomorphic embedding in order to flag when generalization is required. The most relevant point to notice about the residual program is that our PE tool has achieved an optimal specialization by transforming a rather large interpreter into a small residual program (where all the interpretation overhead has been removed). It can also be seen that partial evaluation has done a very good job since the residual program basically corresponds to the Ciao version one would have written by hand.

Example 4. The program in Figure 4 provides a very satisfactory translation from the Java bytecode method exp. In fact, the second argument of predicate exp/2 computes the same value as the result value of the original exp method. While the availability of a LP program which computes the same result as a bytecode method can be of a lot of interest when reasoning about functional properties of the code, it is also of great importance to have augmented the interpreter with an additional argument which computes a trace (see Definition 5) in order to capture the computation history. This will allow observing a good number of interesting properties about the program. The residual program which additionally computes execution traces can be seen in Figure 5. Now, we have a predicate exp/3 whose third argument, on success contains the execution trace at the level of Java bytecode.

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

```
:- module( _, [exp/3] ).

exp(args(B,C),A,[const_step_ok,istore_step_ok,iload_step,istore_step_ok
                 goto_step_ok,iload_step,if0_step_jump,iload_step,
                 iload_step,ibinop_step_ok,istore_step_ok,iinc_step|D]) :-
        C>0, E is B, F is-1+C,
        execute(D,B,E,A,F).
exp(args(_,A),1,[const_step_ok,istore_step_ok,iload_step,istore_step_ok,
                 goto_step_ok,iload_step,if0_step_continue,
                 iload_step,normal_end]) :-
        A=<0.

execute([iload_step,if0_step_jump,iload_step,iload_step,
         ibinop_step_ok,istore_step_ok,iinc_step|F],A,C,E,G) :-
        G>0, H is C*A, I is-1+G,
        execute(F,A,H,E,I).
execute([iload_step,if0_step_continue,iload_step,normal_end],_,C,C,D) :-
        D=<0.
```

Fig. 5. Residual Exponential Program with Trace

about the original bytecode program. The analysis tools we use are based on the technique of abstract interpretation [6] and are part of the CiaoPP system [11]. Abstract interpretation provides a general formal framework for computing safe approximations (i.e., abstractions) of program behaviour. 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. We rely on a generic analysis algorithm (in the style of [12]) defined as a function ANALYZER: $\mathit{Prog} \times \mathit{AAtom} \rightarrow \mathit{AApprox}$ which for a given program $P \in \mathit{Prog}$ and an abstract domain D_α returns $\mathit{Approx}_\alpha \in D_\alpha$. Correctness of analysis ensures that Approx_α safely approximates the semantics of P.

In order to verify the program, the user has to provide the intended semantics (or program specification) as a semantic value $\mathit{Assert}_\alpha \in D_\alpha$[7] in terms of *assertions* (these are linguistic constructions which allow expressing properties of programs) [23]. This intended semantics embodies the requirements as an expression of the user's expectations. The *verifier* has to compare the (actual) inferred semantics Approx_α w.r.t. Assert_α.[8] We use the abstract interpretation-based verifier integrated in CiaoPP. It is dealt here as a function AI_VERIFIER: $\mathit{Prog} \times \mathit{ADom} \times \mathit{AAssert} \rightarrow \mathit{boolean}$ which for a given program $P \in \mathit{Prog}$, an ab-

[7] We denote that Assert_α is a specification given as abstract semantic values of D_α by using the same subscript α.

[8] Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison.

```
class Np{
    public static void main(java.lang.String args[]){
        Np o = new Np();
        o.m2(null);
    }
    public void m2(Np o){
        o = new Np();
        o.m();
    }
    public void m(){}
}
```

Fig. 6. Example for run-time-error freeness

stract domain $D_\alpha \in ADom$ and an intended semantics $Assert_\alpha \in D_\alpha$ succeeds if the abstraction ANALYZER(P, D_α)=$Approx_\alpha$ entails that P satisfies $Assert_\alpha$, i.e., $Approx_\alpha \sqsubseteq Assert_\alpha$.

Definition 7 (verified bytecode). *Let $I_P \in Prog$ be an LP residual program for* JVML$_\beta$_INT *w.r.t. $C_1, \ldots, C_n \in Classes$ and a method invocation specification M. Let $D_\alpha \in ADom$ be an abstract domain and $Assert_\alpha \in D_\alpha$ be the abstract intended semantics. We say that $(C_1, \ldots, C_n, M)$ is verified w.r.t. $Assert_\alpha$ in $ADom$ if* AI_VERIFIER$(I_P,\ D_\alpha, Assert_\alpha)$ *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 the next sections, we show by means of two Java bytecode examples the kind of properties that we can verify about them.

5.1 Run-time Error Freeness Analysis

Our first example corresponds to the verification of the `.class` file associated to the Java program in Figure 6. As new features, we have included in this example objects and several methods (which demonstrates that our approach is not restricted to intra-procedural analysis). The use of objects could in principle issue exceptions of type `NullPointerException`. Clearly, the execution of class `Np` will not produce any exception. 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 above code cannot issue such an exception (nor any other kind of run-time error). Since the program in Fig. 6 corresponds to the execution of a `public static void main` method with all input data statically available, partial evaluation of the interpreter w.r.t. the execution of this method can be fully evaluated. As we are using the interpreter which captures execution traces, the trace computed by partial evaluation is: `[new_step_ok,dup_step_ok,astore_step_ok,aload_step_ok,aconst_null, invokevirtual_step_ok,new_step_ok,dup_step_ok, astore_step_ok,` aload_step_ok, invokevirtual_step_ok,return_step_ok, return_step_ok, normal_end].

Now, we want to specify in Ciao the property "goodtrace" which ensures that the program is run-time error free. This includes the safety issue of not issuing NullPointerException nor any other kind of run-time error (e.g., ArrayIndexOutOfBoundsException, etc). As it is not a predefined property in Ciao, we have to declare it as a regular type using the regtype declarations in CiaoPP.[9] 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(aconst_null).                  goodstep(aload_step_ok).
goodstep(astore_step_ok).               goodstep(dup_step_ok).
goodstep(invokespecial_step_here_ok).   goodstep(invokespecial_step_ok).
goodstep(invokevirtual_step_ok).        goodstep(new_step_ok).
goodstep(return_step_ok).               goodstep(normal_end).
```

Next, we use the following "success" assertion as a way to provide a partial specification of the program.

```
:- checked success main(args(A),B) => goodtrace(B).
```

This assertion should be interpreted as: for all calls to main(args(A),B), if the call succeeds, then B must be a goodtrace on success.

Now, the residual program corresponding to the Java program in Figure 6 is extended with its partial specification and the entry assertion below which describes the valid external queries to predicate main/2:

```
:- entry main(args(A),B) : (term(A), var(B)).
```

Now, CiaoPP performs regular type analysis using, for example, the *eterms* domain [26]. This allows computing safe approximations of the success states of all predicates. After this, CiaoPP performs compile-time checking [24] of the success assertion above, comparing it with the assertions inferred by analysis, and produces as output the following assertion:

```
:- checked success main(args(A),B) => goodtrace(B).
```

Thus, the provided assertion has been marked as checked, i.e., it has been *validated*. When all assertions (in this case only one) have been moved to this checked status, the program has been *verified*.

5.2 Termination Analysis

Program termination is obviously a desirable property in many contexts. Unfortunately, and as it is well known, this is an undecidable property, and therefore we can only expect termination analysis to compute approximate results. In spite

[9] Formally, we define this property as a *regular unary logic* program, see [7].

of this, powerful static analyzers are available which can ensure termination for an important subset of terminating programs. In the termination analysis area, it can be argued that the state of the art in LP is more advanced than that in imperative programming. Some well-known termination analysis systems for LP are TerminWeb [5] and cTi [17]. Either of these systems can be used in order to prove termination of the residual exponential LP program.

Let us consider again the program in Fig 4. Let us also consider the following entry declaration:

```
:- entry exp(args(B,C),A) : (int(B), int(C), var(A)).
```

which describes the valid external queries to the predicate `exp/3`. The argument for proving termination of all calls satisfying the entry declaration above is as follows. Non-termination can only occur in loops. If (1) we can find an argument whose size decreases in every iteration of the loop w.r.t. some norm which assigns values always greater or equal than zero for any term, and (2) the program is *rigid* w.r.t. the size of the corresponding argument (all instances of the term have the same size) and, hence, the program terminates. In Example 3, the only loop we have is for predicate `execute_3/4`. We can conclude termination by reasoning on the last argument. This argument can be inferred to be bound to an integer for all computations originating from the entry assertion above. Since in the recursive path this last argument is decreased before making the recursive call, the program is guaranteed to terminate.

6 Discussion and Ongoing Work

We have implemented our framework within the `CiaoPP` preprocessor [11], where we have a generic analysis engine with a good number of abstract domains. The generic analyzer allows inferring very rich information about LP programs, including data structure shape (with pointer sharing), bounds on data structure sizes, and other operational variable instantiation properties, as well as (global) *procedure-level* properties such as determinacy, termination, non-failure, and bounds on resource consumption (time or space cost). This work attempts to transfer such advanced features available in LP analysis to the verification of Java bytecode. With this aim, we first partially evaluate a Java bytecode interpreter in LP w.r.t. (a LP representation of) a Java bytecode and then analyze the residual program using such LP analysis tools. Our examples show that we are able to reason about non-trivial properties of Java bytecode programs such as termination and run-time error freeness.

We are not aware of any other attempts to use the interpretative approach to the automatic verification of Java bytecode. In our preliminary experimentation we have been able to infer global properties of the computation of the residual LP programs. Thus, the proposed approach is very promising in order to bring the analysis power of LP programs to low-level, imperative code such as Java bytecode. However, the practical uptake of our proposal still depends on a number of open issues which are the subject of our current and future work. In particular, we are not able to properly handle recursive methods by using the traditional partial evaluation technique since it loses too much accuracy and the result is not a useful specialization. We believe the problem could be solved by making use of an advanced partial evaluator which integrates the technique of abstract interpretation (e.g., [22, 15, 19]).

By now, we have only applied our tools to achieve *accuracy* on a set of examples for run-time error freeness and termination. We still have not applied the remaining existing domains in CiaoPP to reason about Java bytecode. More concretely, we expect that we will be able to obtain bounds on resource consumption from the traces that the residual program contains. Also, we are now in the process of studying the scalability of our approach to the verification of medium Java bytecode programs. The analysis tools in CiaoPP are designed with support for incrementality and modularity. We hope that these features will facilitate the scalability of our approach. On the other hand, we also want to assess efficiency issues and, in particular, which is the overhead introduced by the PE process and compare it with existing analysis tools for Java bytecode. These are the lines of ongoing work.

The verification of Java bytecode has received even more attention after the influential Proof-Carrying Code (PCC) idea of Necula [18]. PCC is a general technique for mobile code safety which proposes to associate safety information in the form of a *certificate* to programs. The certificate is created at compile time by relying on a *verifier* on the code supplier side, and it is packaged along with the code. More recently, *Abstraction-Carrying Code* (ACC) has been proposed as a framework for PCC in which the abstraction, automatically computed by a fixed-point analyzer, plays the role of certificate. ACC relies on LP analysis tools (the same ones used in the present work) which are always parametric on the abstract domain with the resulting genericity, which is one of the main advantages of ACC w.r.t. other PCC frameworks. The main limitation of ACC is that it has only been applied by now to *source* LP programs while, in a realistic implementation, the code supplier typically packages the certificate with the *object* code. We believe that our approach here to the verification of Java bytecode by relying on the same tools could help overcome such limitation. However, there are still a number of open issues which have to be studied. For instance, without further improvements, the consumer would have to use the partial evaluator in order to generate the LP representation of the bytecode and then validate the certificate w.r.t. it. Therefore, the partial evaluator would become part of the trusted base code. Also, the resources needed to produce such LP representation can make this approach impractical for devices with resource limitations and a deeper study is required for a practical application.

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

References

1. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq proof assistant reference manual : Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.
2. G. Barthe, D.Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for java. In *Proceedings of Symposium of Security and Privacy '06*. IEEE Press, 2006. to appear.
3. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *Journal of Logic Programming*, 6(1 & 2):135–162, January 1989.
4. 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://clip.dia.fi.upm.es/Software/Ciao/`.
5. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
6. 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.
7. 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.
8. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
9. J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.
10. Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.
11. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.
12. M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.
13. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
14. J. Komorovski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta Programming in Logic, Proceedings of META'92*, volume 649 of *LNCS*, pages 49–69. Springer-Verlag, 1992.
15. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM TOPLAS*, 26(3):413 – 463, May 2004.
16. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
17. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In Michael J. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 7–21, Bonn, Germany, 1996. The MIT Press.
18. G. Necula. Proof-Carrying Code. In *Proc. of POPL'97*, pages 106–119. ACM Press, 1997.

19. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.
20. D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.
21. G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *Proc. of LOPSTR'04*, pages 149–165. Springer LNCS 3573, 2005.
22. G. Puebla, E. Albert, and M. Hermenegildo. Abstract Interpretation with Specialized Definitions. In *The 13th International Static Analysis Symposium (SAS'06)*, LNCS. Springer, August 2006. To appear.
23. G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.
24. G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.
25. Brian J. Ross. The partial evaluation of imperative programs using prolog. In *META*, pages 341–363, 1988.
26. 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.

A JVML$_\beta$ syntax

Figure 7 shows the grammar of JVML$_\beta$. In this grammar, words beginning with an uppercase represent non-terminals (except Int, Bool, UnsignedInt and String, which have the usual meaning), while words in lowercase represent terminals which could be constants, functor or predicate names in first order logic. Thus, for instance, we can see that a program in JVML$_\beta$ consists in a fact with *program* as predicate name, and two lists as arguments, the first one being a list of *Class* terms, and the second one a list of *Interface* terms. The bytecode instructions are represented separately as a set of *Bytecode* facts all together inside a same file. In order to differentiate them, they include both the method and the class which the bytecode instruction belongs to (see Example 1 for details). It is interesting to note that a full *Class* term will store all information relative to the compilation of a Java class (except the bytecode instructions) as it is specified by the JVML$_\beta$, as well as the `.class` file stores all information relative to the compilation of a Java class as it is specified by the JVML.

B Sun specification of `bipush`

Figure 8 is an extract of Sun's Java Virtual Machine Specification that describes the `bipush` instruction. `sipush` and `iconst_<i>` instructions are also discribed in the JVM Specification and the three of them are very similar and have been factorized to the `const` instruction in JVML$_\beta$.

```
Program              ::= program(Classes|Interfaces).
Classes              ::= [ ] | [Class|Classes]
Interfaces           ::= [ ] | [Interface|Interfaces]
Class                ::= class(ClassName,OptionClassName,SuperInterfaces,Fields,Methods,
                         final(Bool),public(Bool),abstract(Bool))
Interface            ::= interface(InterfaceName,SuperInterfaces,Fields,Methods,
                         final(Bool),public(Bool),abstract(Bool))
ClassName            ::= className(packageName(String),shortClassName(String))
OptionClassName      ::= none | ClassName
InterfaceName        ::= interfaceName(packageName(String),shortClassName(String))
SuperInterfaces      ::= Interfaces
Fields               ::= [ ] | [Field|Fields]
Field                ::= field(FieldSignature,final(Bool),static(Bool),
                         Visibility,initialValue(InitialValue))
FieldSignature       ::= fieldSignature(FieldName,Type)
Visibility           ::= package | protected | private | public
InitialValue         ::= undef | null | int(Int)
FieldName            ::= fieldName(ClassName,ShortFieldName)
ShortFieldName       ::= shortFieldName(String)
Type                 ::= primitiveType(PrimType) | refType(RefType)
PrimType             ::= boolean | byte | short | int
RefType              ::= classType(ClassName) | interfaceType(InterfaceName) | arrayType(Type)
Methods              ::= [ ] | [Method|Methods]
Method               ::= method(MethodSignature,OptionBytecodeMethod,
                         final(Bool),static(Bool),Visibility)
MethodSignature      ::= methodSignature(MethodName,Parameters,OptionType)
MethodName           ::= methodName(ClassName,ShortMethodName)
ShortMethodName      ::= shortMethodName(String)
Parameters           ::= [ ] | [Type|Parameters]
OptionType           ::= none | Type
OptionBytecodeMethod ::= none | bytecodeMethod(StackSize,LocalVarSize,FirstAddress,
                         methodId(ModuleName,MethodIndex),ExceptionHandlers)
StackSize            ::= UnsignedInt
LocalVarSize         ::= UnsignedInt
FirstAddress         ::= Pc
ModuleName           ::= String
MethodIndex          ::= UnsignedInt
Instructions         ::= [ ] | [Instruction|Instructions]
ExceptionHandlers    ::= [ ] | [ExHandler|ExceptionHandlers]
ExceptionHandler     ::= exceptionHandler(OptionClassName,StartPc,EndPc,HandlerPc)
StartPc              ::= Pc
EndPc                ::= Pc
HandlerPc            ::= Pc

Bytecode             ::= bytecode(ModuleName,Pc,MethodIndex,Instruction,Offset).
Pc                   ::= UnsignedInt
MethodIndex          ::= UnsignedInt
Offset               ::= Int
VariableIndex        ::= UnsignedInt
Instruction          ::= aaload | aastore | aconst_null | aload(VariableIndex) | areturn |
                         arraylength |anewArray(refType(RefType)) | astore(VariableIndex) |
                         athrow | baload | bastore | checkcast(refType(RefType)) |
                         const(primitiveType(PrimType),Int) | dup | dup_x1| dup_x2 |
                         getfield(FieldSignature) | getstatic(FieldSignature) | goto(Offset) | i2b |
                         i2s | ibinop(BinOpType) | iaload | iastore | if_acmpeq(Offset) |
                         if_acmpne(Offset) | if_icmp(Offset,CompType) | if0(Offset,CompType) |
                         ifnonnull(Offset) | ifnull(Offset) | iinc(VariableIndex,Int)|
                         iload(VariableIndex) | instanceof(refType(RefType)) |
                         invokestatic(MethodSignature) | invokevirtual(MethodSignature) |
                         ireturn | istore(VariableIndex) | multianewarray(refType(RefType)) |
                         new(ClassName) | newarray(primitiveType(PrimType)) | nop |
                         pop | pop2 | putfield(FieldSignature) |
                         putstatic(FieldSignature) | return | saload | sastore | swap |ineg |
BinOpType            ::= addInt | andInt | divInt | mulInt | orInt | remInt |
                         shlInt | shrInt | subInt | xorInt
CompType             ::= eqInt | neInt | ltInt | leInt | geInt | gtInt
```

Fig. 7. JVML$_\beta$ syntax

Operation

Push `byte`

Format

bipush
byte

Forms

bipush = 16 (0x10)

Operand Stack

... ⇒ ..., value

Description

The immediate byte is sign-extended to an int value. That value is pushed onto the operand stack.

Fig. 8. Sun specification of `bipush`