Java bytecode Verification using Analysis and transformation of logic programs

Laurent Hubert

June 2006

A thesis submitted for the degree of:
`Research Master in Computer Science`

Adviser:
`Germán Puebla`

Acknowledgment

First, I would like to thank my adviser, Germán Puebla, for his time, his support and his help on this work. I am also grateful to Edison Mera, for his technical support with `Ciao` and `CiaoPP`, and David Pichardie for useful discussions on Bicolano. This internship has also been a time period to think about my future and the possibility of doing a Ph.D and I thank Mireille Ducassé, David Pichardie and Manuel Hermenegildo for their useful advices. Finally, last but not least, I thank Charlotte for her moral support despite the one thousand kilometers that were between us.

Abstract

When a new programming language comes out, previous analyses can be applied to this new language by re-implementing them all. This work proposes the opposite method, by translating the low-level language of the Java Virtual Machine into the high-level language that Logic Programming (LP) is, to allow using all the already well-developed analyses for LP.

The technique used relies on the partial evaluation of a Java bytecode interpreter developed in LP with respect to (an LP representation of) a set of Java bytecode classes. The residual LP program can then be analyzed by the state-of-the-art analyzer for (Constraint) Logic Programming — (C)LP. Interestingly, at least for the examples we have studied, it produces very simple LP representation of the original Java programs by recovering the structure hidden in the bytecode representation. Reasoning about properties of such residual programs allows automatically proving some non-trivial properties of Java bytecode programs like termination and run-time error freeness.

A short version of this work will be presented in August 2006 in Seattle (USA) at the international workshop on *Software Verification and Validation* (which is in conjunction with *Federated Logic Conferences* (FLoC) 2006,

Contents

Introduction

The technique of abstract interpretation [7] 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 many 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 pointer sharing), bounds on data structure sizes, and other 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` [14] is the abstract interpretation-based preprocessor of the `Ciao` (C)LP system [5]. It uses modular, incremental abstract interpretation as a fundamental tool to obtain information about programs. The semantic approximations thus produced have been applied to program verification and optimizations during program compilation, including transformations such as parallelization and resource usage control.

A principal advantage of verifying programs on a *source* code level is that complex global properties can be infer 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, 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 Java bytecode verification which exploits the expressiveness, automation and genericity of the advanced analysis tools for (C)LP. In order to achieve this goal, we have developed a meta-program implementing the semantic of the Java Virtual Machine Language (JVML) that can be partially evaluated by `CiaoPP` to produce a residual program that can be analyzed by either `CiaoPP` or another LP analyzer. The whole verification process is split in three parts.

1. *Translation to LP.* We use LP as a language for representing and manipulating JVML programs. We have implemented an automatic translator

Parser which, given a set $\{\texttt{class}_1, \ldots, \texttt{class}_n\}$ of `.class` files returns an LP representation P of them in JVML_r (a representative subset of JVML which will be discussed in detail in Chapter 1). Furthermore, we have also implemented in LP an interpreter *Int*, which captures the JVML semantics. In addition, the interpreter computes *execution traces*, which will be very useful for reasoning about certain properties.

2. *Partial evaluation.* We have used an existing partial evaluator for (C)LP in order to specialize *Int* with respect to the LP representation P of $\{\texttt{class}_1, \ldots, \texttt{class}_n\}$, 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 analyzing the residual program I_P obtained in 2) with state-of-the-art analyzers developed for (C)LP.

The resulting scheme has been incorporated in the `CiaoPP` preprocessor. The first chapter is focused on the JVML_r language, its grammar and the resulting parser, its semantics and the interpreter. The second chapter will then present the generation of the residual program with partial evaluation and the analysis of this residual program. The example shown in Figure 2 will be used throughout this report to show the results of the different parts.

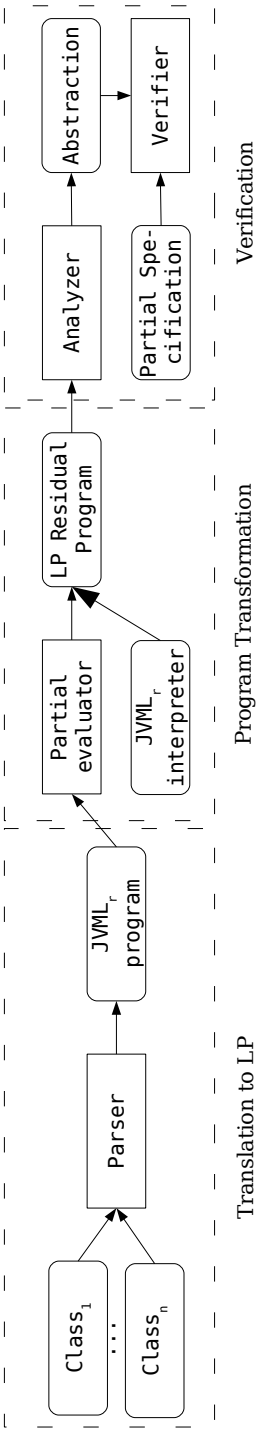

Figure 1: Java Bytecode Verification using Transformation and Analysis Tools for (C)LP

```
class ExpFact{
  private int _fact;
  private int _exp;

  public static void main(int base, int exponent, int fact){
    ExpFact e = new ExpFact();
    int t;
    t = ExpFact.exp(base,exponent);
    e.setExp(t);
    try{
      t = ExpFact.fact3(fact);
      e.setFact(t);
    }catch(java.lang.ArithmeticException ex){}
  }

  public void setExp(int exp){_exp=exp;}
  public int getExp(){return _exp;}

  public void setFact(int fact){_fact=fact;}
  public int getFact(){return _fact;}

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

  public static int fact3(int n){
    if(n>11) throw(new java.lang.ArithmeticException());
    if(n>1) return n*fact2(n-1);
    if(n>=0) return 1;
    throw(new java.lang.ArithmeticException());
  }
}
```

Figure 2: Java Source Code of the Main Example

Chapter 1

The JVML$_{\mathrm{r}}$ Language

JVML is optimized for speed and compactness of the `.class` files. Those optimizations are achieve through the use of the whole set of bytecode instruction, which means that several instructions have very similar semantics, and thought the use of a constant-pool [19] (a structure present in the `.class` file which stores constants, field and method names and descriptors, class names, etc.) with several levels of indirections. Those optimizations are adapted to execution but makes the code harder to read and meta-programs like interpreters and analyzers harder to program and to maintain. JVML also handles some features not yet supported by the interpreter we have developed like operations on floats, doubles and longs, concurrency and static initializations. To formalized all the simplifications made and to represents JVML in LP, we have specified the JVML$_{\mathrm{r}}$ language (where we have to chosen to add the subscript r to denote that it is a $r$educed version of JVML). In the first section of this chapter, the grammar of the language and the parser are presented while the second section presents its dynamic semantics and the interpreter.

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 *Parser* and Section 1.2 presents *Int*.

1.1 A simplified version of the Java Virtual Machine Language

The input of the verification process is a set of JVML `.class` files, denoted as $\{\texttt{class}_1, \ldots, \texttt{class}_n\}$, which describe the information of a set of Java classes (as specified by JVML, see the Java Virtual Machine Specification [19]). Then, the program named *Parser* in Figure 1 takes $\{\texttt{class}_1, \ldots, \texttt{class}_n\}$ and returns an LP program which contains the information available in the classes and represents it in the JVML$_{\mathrm{r}}$ language. JVML$_{\mathrm{r}}$ 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 types as float, double, long and string, concurrency and `tableswitch` and `lookupswitch` instructions are left out of the chosen subset.

Figure 1.1 shows the formal syntax of JVML$_r$. 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, we can see that a JVML$_r$ program encapsulated in a fact and consists in a term 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 they belong 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$_r$, as well as the `.class` file stores all information relative to the compilation of a Java class as it is specified by the JVML.

As notation, we use respectively *Prog*, *JProg*, and *Classes* to denote LP programs, LP program containing a JVML$_r$ representation, and a the set of all correct (as specified in the *The `class` file format* specification [19]) `.class` files.

Definition 1 (*Parser*) *We define function Parser* : $2^{Classes} \rightarrow$ *JProg which takes a set of* `.class` *files* $\{\texttt{class}_1, \ldots, \texttt{class}_n\} \in 2^{Classes}$ *and returns an LP program* $P \in$ *JProg which is the* LP representation *of* $\texttt{class}_1 \ldots \texttt{class}_n$.

The implementation of *Parser* in `Ciao` [5] reads the `.class` files byte by byte and organizes and interprets them as it is specified in [19]. As a result, it produces an LP program which consists of a set of facts containing the same information as the original `.class` files. This set of fact can be devied into 1) a set of `bytecode` facts reprensting the bytecode instructions for the methods in all the involved Java classes and 2) a single `program` fact obtained by putting together the `class` and `interface` terms which store all required information — except for the bytecode instructions which appear separately in the aforementioned facts. While Figure 1.1 gives the formal syntax of the language, Example 1 shows a concrete utilization of it. The differences between JVML and JVML$_r$ are essentially the two following ones.

1. *Bytecode factorization.* In order to optimize JVML [19], Sun has duplicated some instructions, like the instruction that puts an integer onto the stack, to specialized them depending on their arguments. For the previous instruction, eleven different bytecodes can be used (`iconst_m1`, `iconst_0`, `iconst_1`, `iconst_2`, `iconst_3`, `iconst_4`, `iconst_5`, `bipush`, `sipush`, `ldc`, and `ldc_w`). This as well as other duplicated instructions can be factorized in order to have less instructions without affecting expressiveness.[1] This makes the JVML$_r$ code easier to read (as well as the traces which will be discussed in Section 1.2) and the interpreter easier to program and maintain.

2. *References resolution.* Original JVML instructions very often use indexes onto the *constant-pool* [19], which can refer to other indexes into this structure and which leads to several levels of indirections. *Parser* removes all

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

```
Program                 ::=  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
```

Figure 1.1: JVML_r Grammar

```
bytecode('ExpFact_class',0,7,const(primitiveType(int),1),1).
bytecode('ExpFact_class',1,7,istore(2),1).
bytecode('ExpFact_class',2,7,iload(1),1).
bytecode('ExpFact_class',3,7,istore(3),1).
bytecode('ExpFact_class',4,7,iload(3),1).
bytecode('ExpFact_class',5,7,if0(eqInt,13),3).
bytecode('ExpFact_class',8,7,iload(2),1).
bytecode('ExpFact_class',9,7,iload(0),1).
bytecode('ExpFact_class',10,7,ibinop(mulInt),1).
bytecode('ExpFact_class',11,7,istore(2),1).
bytecode('ExpFact_class',12,7,iinc(3,-1),3).
bytecode('ExpFact_class',15,7,goto(-11),3).
bytecode('ExpFact_class',18,7,iload(2),1).
bytecode('ExpFact_class',19,7,ireturn,1).
```

Figure 1.2: Partial output of *Parser* for `exp`

references to the constant-pool table in the bytecode instructions by replacing them with the complete information. This can be seen as *unfolding* steps which could benefit an analyzer's inference task later.[2] Thus, we no longer need the constant-pool table as all the required data are included within the JVML$_r$ representation.

Example 1 *In Figure 2, we show the Java method* `exp` *which computes the exponential for the parameters* `base` *and* `exponent`*. The execution of Parser on this example returns an LP program in the JVML$_r$ language containing all the information concerning the class to which* `exp` *belongs. Due to space limitations, Figure 1.2 we only show the bytecode facts which correspond to the method* `exp`*. 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 identify the class. This allows to deal with bytecode instructions coming from different Java classes. It can be noted 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).*

1.2 Dynamic Semantics of JVML$_r$ and Interpretation

The formal JVML specification chosen as a based to implement the semantics of JVML$_r$ in an interpreter is Bicolano [24]. Bicolano is developed within the Mobius [3] european project, which aims to verify the Java programs for mobile phones. Bicolano is written with the Coq Proof Assistant [2] — this allows checking that the specification is consistent and also proving properties on the behavior of some programs — and describes a superset[3] of JVML$_r$.

In the specification, a state is modeled by a 3-tuple, which can either be a normal state $\langle$ *Heap*, *Frame*, *StackFrame* $\rangle$ or an exception state $\langle$ *Heap*,

[2] It should be noted that *Mix* 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).

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

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.

Figure 1.3: Sun specification of `bipush`

$ExceptionFrame$, $StackFrame$ ⟩, and represents the machine's state where:

- $Heap$ represents the content of the heap,
- $Frame$ represents the execution state of the current *Method*,
- $ExceptionFrame$ represents the execution state of the current method when an exception has been thrown and not yet caught, and
- $StackFrame$ is a list of $Frame$s corresponding to the call stack.

Each $Frame$ is of the form ⟨ $Method, PC, OperandStack, LocalVar$ ⟩ and contains the stack of operands $OperandStack$ and the values of the local variables $LocalVar$ at the program point PC of the method $Method$. In an $ExceptionFrame$, the stack $OperandStack$ is replaced by the address of an object — which must inherit from the class *Exception* — in the heap $Heap$.

The definition of the dynamic semantics is based on the notion of *step*.

Definition 2 (*step*) *The dynamic semantics of each instruction is specified as a partial function* $step : JProg \times States \rightarrow States \times Step\_Names$ *that, given a program* $P \in JProg$ *and a state* $S \in States$*, computes the next state* $S' \in States$ *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 *Figure 1.3 is an extract of Sun's Java Virtual Machine Specification that describes the* `bipush` *instruction.* `sipush` *and* `iconst_<i>` *instructions are also described in the JVM Specification and the three of them are very similar and have been factorized to the* `const` *instruction in JVML$_r$. The Coq representation in Bicolano of those three JVM instruction 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)
```

Below is the `Ciao` *translation of the same instruction:*

```
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).
```

The full implementation of the `step` relation can be find in Appendix A.

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 States with* $S \xrightarrow{T}{}^*_P S'$ *if and only if:*

- *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* $S'' \in States$ *such that* $S' \xrightarrow{L}_P S''$, *and*
- $T = [L_1, \ldots, L_n]$, *with* $T \in Traces$, *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* $P \in JProg$ *be a program to be executed and* $S \in States$ *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 *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 (*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* $Int(P, M) = (R, T)$ *with*

- $S = initialState(P, M)$ *where* $initialState$ *builds a state* $S \in States$ *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 *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 *Int* returns the trace and the result of the method but it is straightforward to modify the definitions of *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).

Chapter 2

Transformation and Analysis of Logic Programs

2.1 Specialization of the Interpreter

Partial evaluation 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 with respect to 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 the computations that depend only on the static data are performed — at partial evaluation time — once and for all.

2.1.1 Basics of Partial Evaluation

In this section, some needed definitions of logic programming presented in [21] are re-introduced and then partial evaluation of logic programs is formally defined (see [20] for more detailed definitions).

Definition 6 (computation rule) *A* computation rule $\mathcal{R}$ *is a function that, given a goal* $G = \leftarrow s_1, \ldots, s_r, \ldots, s_n$ *with* $n \geq 1$*, returns an atom* s_r*, called the* selected *atom, in that goal.*

When executing a Prolog program, the selected atom is the one the most at the left. For partial evaluation, it can be interesting in some cases to "jump" over some atoms that cannot be evaluated.

Definition 7 (derivation step) *Let* $G = \leftarrow s_1, \ldots, s_r, \ldots, s_n$ *be a goal,* $\mathcal{R}$ *be a computation rule with* $\mathcal{R}(G) = s_r$*, and* $C = h \leftarrow t_1, \ldots, t_m$ *be a properly renamed apart clause.* G' *is* derived *from* G *and* C via $\mathcal{R}$ *if the following conditions hold:*

$$\theta = mgu(s_r, h)$$
$$G' = \leftarrow (s_1, \ldots, s_{r-1}, t_1, \ldots, t_m, s_{r+1}, \ldots, s_n)\theta$$

In the following, it's is denoted by $G \rightsquigarrow_\theta G'$*.*

Several derivation steps can give an *SLD derivation*. An *SLD derivation* is a possibly infinite sequence of goals $G_0, G_1, \ldots$ and a sequence $\theta_1, \theta_2, \ldots$ of most general unifiers such that $G_i \rightsquigarrow_{\theta_{i+1}} G_{i+1}$. A finite (possibly incomplete) SLD derivation of goals $G_0, \ldots, G_n$ and mot general unifiers $\theta_1, \ldots, \theta_n$ with $\theta = \theta_1 \ldots \theta_n$, is denoted by $G_0 \rightsquigarrow_\theta^* G_n$. When an atom s_r can be unified with several clauses of P, the derivation step can be non-deterministic. Such SLD derivations can be organized in *SLD trees*. The execution of a logic program corresponds to a traversal of a such tree; it can be either breadth-first in some cases or more generally depth-first like in the case of Prolog. Partial evaluation therefore rely on the simplification of those SLD trees.

Definition 8 (resultant of a derivation) *Let P be a program, s an atom, t a conjunction of atoms, and $\leftarrow s \rightsquigarrow_\theta^* \leftarrow t$ an SLD derivation of the goal $\leftarrow s$ in P that leads to the goal $\leftarrow t$. The* resultant *of this SLD derivation is the clause $s\theta \leftarrow t$.*

Definition 9 (partial evaluation) *let P be a logic program, s an atom, and τ an SLD tree for $\leftarrow s$ in P. Let $G_1, \ldots, G_n$ be (non root) goals in τ chosen so that each non failing branch of τ contains exactly one of them. Let R_i be the resultant of the derivation $\leftarrow s \rightsquigarrow G_i$. Then the set of clauses $\{R_1, \ldots, R_n\}$ is called a* partial evaluation of s in P.
If $S = \{s_1, \ldots, s_r\}$ is a finite set of atoms, then a partial evaluation of S in P *is the union of partial evaluations of $s_1, \ldots, s_r$ in P.*

For a given atom s in a program P, there exists in general infinitely many different partial evaluation of s in P. This choice is abstracted by the definition of an *unfolding rule*.

Definition 10 (unfolding rule) *An* unfolding rule *U is a function which, given a program P and an atom s, returns exactly one finite set of resultants that is a partial evaluation of s in P. If S is a finite set of atoms and P a program, then $U(S, P)$ denote the union of $U(s, P)$ for all $s \in S$.*

In order to have termination, an abstraction operator is needed. This operator also ensures independence[1] and closedness, two conditions to correctness, as proved in [20].

Definition 11 (abstraction) *Given a finite set of atoms S, an* abstraction operator *abstract is a function which returns abstract(S), a finite independent set of atoms with the same predicates as those in S, such that every atom in S is an instance of an atom in abstract(S).*

The the naive algorithm presented in Figure 2.1 computes the set of atoms T_{out} that need to be partially evaluated in order to have a sound partial evaluation $P' = U(T_{out}, P)$ of the program P with respect to a set of atoms T_{in}.

2.1.2 Futamura Projections

Possible uses of partial evaluation are known as Futamura projections [11]. Our work is based on the first one and could be extended with the second one. They

[1] Independence is often achieve through renaming, not discussed here.

```
Input: a program P and a set of atoms T_in
Output: a set of atoms T_out
begin
 i := 0;
 S_0 := T_in ;
 repeat
  P' := U(S_i,P) ;
  A := S_i ∪ {p_j(t_{1,j},...,t_{n_j,j})|
              B←p_{1,1}(t_{1,1},...,t_{n_1,1}),...,p_m(t_{1,m},...,t_{n_m,m}) ∈ P'} ;
  S_{i+1} := abstract(S_i) ;
  i := i+1 ;
 until S_i = S_{i-1} ;
 T_out := S_i ;
end
```

Figure 2.1: A General Algorithm for Partial Evaluation

will be presented in sections 2.1.2 and 2.1.2 but few notations and definitions need to be introduced before.

$[\![P]\!]_L : D \times D^* \rightarrow D^*$ denotes the semantics of the program $P \in D$ in the language L that takes argument in $D \times D^*$ and returns a result in D^*. One can notice that the program is in the domain of the data. Several languages can be used: a language for the implementation of the different tools, a source language and an object language, respectively represented by L_I, L_S and L_O. We denote by L_X and L_Y some languages that can be any of the previous ones.

In the following definitions, let `source` be a program written in L_S and $\mathtt{d} = (\mathtt{d}_1, \mathtt{d}_2) \in D \times D^*$ be some input data.

Definition 12 (Interpreter) *An* interpreter `int` *is a program that takes a source program* `source` *and its data* `d` *and that must return the same result as if the source program had been directly applied to the data. It can be formalized as:*

$$\mathtt{output} = [\![\mathtt{source}]\!]_{L_S}(\mathtt{d}) = [\![\mathtt{int}]\!]_{L_I}(\mathtt{source}, \mathtt{d}) \tag{2.1}$$

Definition 13 (Compiler) *A* compiler `comp` *is a program that takes a source program as input and generate a program* `object` *written in* L_O *which, when applied to some data* `d` *must return the same result as if the program* `source` *had been directly applied to* `d`*. This can be formalized as:*

$$\mathtt{object} = [\![\mathtt{comp}]\!]_{L_I}(\mathtt{source}) \tag{2.2}$$

$$\mathtt{output} = [\![\mathtt{source}]\!]_{L_S}(\mathtt{d}) = [\![\mathtt{object}]\!]_{L_O}(\mathtt{d}) \tag{2.3}$$

Definition 14 (Partial Evaluator) *A* partial evaluator $\mathtt{mix}_{L_X,L_Y}$ *is a program, here written in an language* L_X*, that takes a program* `prog` *written in* L_X *and a part* $\mathtt{d}_1 \in D$ *of its input data* $(\mathtt{d}_1, \mathtt{d}_2) \in D \times D^*$ *and generate a partial evaluation, written in* L_Y*, of* `prog` *with respect to* $\mathtt{d}_1$*. As explained in Section 2.1.1, given the rest of its input data, the partial evaluated program behaves as if* `prog` *was directly applied to the full input data, this can be formalized as:*

$$[\![\mathtt{prog}]\!]_{L_X}(\mathtt{d}_1, \mathtt{d}_2) = [\![[\![\mathtt{mix}_{L_X,L_Y}]\!]_{L_X}(\mathtt{prog}, \mathtt{d}_1)]\!]_{L_Y}(\mathtt{d}_2) \tag{2.4}$$

First Futamura Projection

The first Futamura Projection rely on the above definitions to define a generic way of compiling programs using a partial evaluator and an interpreter. The equations used in order to deduce the following ones are written above the equal sign.

$$\begin{aligned} \texttt{output} &\overset{(2.1)}{=} [\![\texttt{int}]\!]_{L_I}(\texttt{source}, \texttt{d}) \\ &\overset{(2.4)}{=} [\![[\![\texttt{mix}_{L_I,L_O}]\!]_{L_I}(\texttt{int}, \texttt{source})]\!]_{L_O}(\texttt{d}) \qquad (2.5) \\ &\overset{(2.3)}{=} [\![\texttt{object}]\!]_{L_O}(\texttt{d}) \qquad (2.6) \end{aligned}$$

The two latest equations lead to the following definition.

$$\texttt{object} \overset{(2.5\ \&\ 2.6)}{=} [\![\texttt{mix}_{L_I,L_O}]\!]_{L_I}(\texttt{int}, \texttt{source}) \qquad (2.7)$$

This equation implies that it is possible to replace a compiler by a partial evaluator and an interpreter. The main advantage of this replacement is that an interpreter is usually easier to write than a compiler and, even if the result of the partial evaluation is not as efficient as a compiler written from scratch, it in general more efficient that simply interpreting the program, i.e., without partial evaluation.

The other advantage is that an existing partial evaluator can be reused, and should therefore be more robust. As an interpreter can be seen as a specification of the language, the risk of having an error with a such compilation is very low.

Second Futamura Projection

The second Futamura Projection introduce a way to generate a compiler from an interpreter and a partial evaluator.

$$\begin{aligned} \texttt{object} &\overset{(2.7)}{=} [\![\texttt{mix}_{L_I,L_O}]\!]_{L_I}(\texttt{int}, \texttt{source}) \\ &\overset{(2.1)}{=} [\![[\![\texttt{mix}_{L_I,L_O}]\!]_{L_I}(\texttt{mix}_{L_I,L_O}, \texttt{int})]\!]_{L_O}(\texttt{source}) \qquad (2.8) \\ &\overset{(2.2)}{=} [\![\texttt{comp}]\!]_{L_I}(\texttt{source}) \qquad (2.9) \end{aligned}$$

In those equations, if only one partial evaluator is used, it needs to be self-applicable, which implies that the language of implementation of the partial evaluator need to be the same as the one of the programs it evaluates, which is the case of the partial evaluator `mix` defined in Definition 14.

$$[\![\texttt{comp}]\!]_{L_I} \overset{(2.8\ \&\ 2.9)}{=} [\![[\![\texttt{mix}_{L_I,L_O}]\!]_{L_I}(\texttt{mix}_{L_I,L_O}, \texttt{int})]\!]_{L_O} \qquad (2.10)$$

This second Futamura Projection means we can automatically generate a compiler for programs in L_S from a partial evaluator and an interpreter both written in the implementation language L_I. In the general equation (2.10), the equality is between the semantics because the languages L_I and L_O of the two compilers are different. However, the result produced by the partial evaluation studied in Section 2.1.1 and the input program are both in the same language (in that case, a logic programming language). As aforementioned, if we use only one partial evaluator, it needs to be self-applicable, so the implementation language

must be the logic programming language chosen previously for the program. It could therefore give a compiler in the same language as `comp`, but the object program `object` would also be in this language. In the studied case, L_I and L_O are both high level languages and a compiler is needed to compile `object`. It could also be possible that L_I and L_O are low level languages, but this would be more difficult to implement and not as efficient.

2.1.3 Automatic Generation of Residual LP Programs

We use the partial evaluator for (C)LP programs of [25] written in `Ciao` and which is part of `CiaoPP`. We represent it here as a function $Mix = [\![\texttt{mix}_{\text{LP,LP}}]\!] : Prog \times Data \rightarrow Prog$ 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* [16] of P with respect to S. It

The development of partial evaluation, program specialization and related techniques [10, 17, 16, 12, 4] has led to the now established approach to compilation (known as the first Futamura projection presented in Section 2.1.2) based on specializing an interpreter with respect to a fixed source program. This allows the translation of the program S into another programming language, in this case `Ciao`. The residual program is ready now to be, for instance, executed in such language or, as in this case, analyzed by tools for the language in which it has been translated. In the (C)LP context, this interpretative approach has been applied to analyze high-level imperative languages [23] and also the PIC processor [13] 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 *Int* with respect to a method invocation specification M (see Definition 5 above) and an object program $P = Parser(\{\texttt{class}_1, \ldots, \texttt{class}_n\})$. This results in a residual LP program, I_P.

Definition 15 (LP residual program) *Let $Int \in Prog$ be a $JVML_{\text{r}}$ interpreter, M be a method invocation specification, $\{\texttt{class}_1, \ldots, \texttt{class}_n\} \in 2^{Classes}$ be a set of classes and $P \in JProg$, with $P = Parser(\{\texttt{class}_1, \ldots, \texttt{class}_n\})$, be a $JVML_{\text{r}}$ program. The* LP residual program*, I_P, for Int with respect to P and M is defined as $I_P = Mix(Int, (P, M))$.*

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 2.2 the result of the automatic partial evaluation of an implementation of the interpreter which does not output the trace (see Definition 5) w.r.t. the LP translation of the program in Example 1, an empty 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.*

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

exp_notrace([B,C],A) :-
        C\=0, D is B, E is-1+C,
        execute_notrace(A,B,C,D,E) .
exp_notrace([A,0],st(heap(dynamicHeap([]),staticHeap([])),fr(method(methodSignature(methodName(
  className(packageName(''),shortClassName('ExpFact')),shortMethodName(exp)),[primitiveType(int),
  primitiveType(int)],primitiveType(int)),bytecodeMethod(4,2,0,methodId('ExpFact_class',7),[]),
  final(false),static(true),public),19,[num(int(1))],[num(int(A)),num(int(0)),num(int(1)),
  num(int(0))]),[])).

execute_notrace(A,B,C,D,E) :-
        E\=0,
        F is D*B,
        G is-1+E,
        execute_notrace(A,B,C,F,G) .
execute_notrace(st(heap(dynamicHeap([]),staticHeap([])),fr(method(methodSignature(methodName(
  className(packageName(''),shortClassName('ExpFact')),shortMethodName(exp)),[primitiveType(int),
  primitiveType(int)],primitiveType(int)),bytecodeMethod(4,2,0,methodId('ExpFact_class',7),[]),
  final(false),static(true),public),19,[num(int(C))],[num(int(A)),num(int(B)),num(int(C)),
  num(int(0))]),[]),A,B,C,0).
```

Figure 2.2: Residual Exponential Program

For this example, we have used the local control strategy based on homeomorphic embedding *which is described in [25]. 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 (except that he would not have output the whole step).*

Example 4 *The program in Figure 2.2 provides a very satisfactory translation from the Java bytecode method* `exp`*. While the availability of a LP program which computes just the final state 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 2.3. Now, we have a predicate* `exp/3` *whose third argument, on success contains the execution trace at the level of Java bytecode.*

2.2 Analysis of Logic Programs

Having obtained an LP representation of a Java bytecode program, the next task is to use existing analysis tools for (C)LP in order to infer and verify properties about the original bytecode program. The analysis tools used are based on the technique of abstract interpretation [7] and are part of the `CiaoPP` system [14]. Abstract interpretation provides a general formal framework for computing safe approximations (i.e., abstractions) of program behavior. 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

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

exp([A,0],st(heap(dynamicHeap([]),staticHeap([])),fr(method(methodSignature(methodName(
  className(packageName(''),shortClassName('ExpFact')),shortMethodName(exp)),
  [primitiveType(int),primitiveType(int)],primitiveType(int)),bytecodeMethod(4,2,0,
  methodId('ExpFact_class',7),[]),final(false),static(true),public),19,[num(int(1))],
  [num(int(A)),num(int(0)),num(int(1)),num(int(0))]),[]),
  [const_step_ok,istore_step_ok,iload_step,istore_step_ok,iload_step,if0_step_jump,
  iload_step,normal_end]).
exp([B,C],A,[const_step_ok,istore_step_ok,iload_step,
  istore_step_ok,iload_step,if0_step_continue,iload_step,iload_step,ibinop_step_ok,
  istore_step_ok,iinc_step|D]) :-
        C\=0,
        E is B,
        F is-1+C,
        execute_3_1(A,D,B,C,E,F) .

execute_3_1(st(heap(dynamicHeap([]),staticHeap([])),fr(method(methodSignature(methodName(
  className(packageName(''),shortClassName('ExpFact')),shortMethodName(exp)),
  [primitiveType(int),primitiveType(int)],primitiveType(int)),bytecodeMethod(4,2,0,
  methodId('ExpFact_class',7),[]),final(false),static(true),public),19,[num(int(C))],
  [num(int(A)),num(int(B)),num(int(C)),num(int(0))]),[]),
  [goto_step_ok,iload_step,if0_step_jump,iload_step,normal_end],A,B,C,0).
execute_3_1(A,[goto_step_ok,iload_step,if0_step_continue,iload_step,iload_step,ibinop_step_ok,
  istore_step_ok,iinc_step|F],B,C,D,E) :-
        E\=0,
        G is D*B,
        H is-1+E,
        execute_3_1(A,F,B,C,G,H) .
```

Figure 2.3: Residual Exponential Program with Trace

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 denote by $ADom$ the set of all possible abstract domain. We rely on a generic analysis algorithm (in the style of [15]) that takes as parameter, as well as a program, an abstract domain and produces an approximation of the program with respect to the given abstract domain. An approximation of a program is a set of 3-tuple $\langle A : CP \rightarrow AP \rangle$ where A is an predicate name, CP (*Calling Pattern*) is a substitution in the abstract domain that describes a call to A, and AP (*Answer Pattern*) is another substitution in the abstract domain that describes the answer of the predicate. The set of all possible approximation is denoted by $AApprox$. Correctness of analysis ensures that $Approx_\alpha$ 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 $Assert_\alpha \in AApprox$ in terms of *assertions* (these are linguistic constructions which allow expressing properties of programs) [26]. 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. $Assert_\alpha$.[2] The verifier used is the abstract interpretation-based verifier integrated in `CiaoPP`. It is dealt here as a function $AIVerifier : Prog \times ADom \times AApprox \rightarrow boolean$ which for a given program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and an intended semantics $Assert_\alpha \in D_\alpha$ succeeds if the abstraction $Analyzer(P, D_\alpha) = Approx_\alpha$ entails that Psatisfies $Assert_\alpha$, i.e., $Approx_\alpha \sqsubseteq Assert_\alpha$.

[2]Comparison between actual and intended semantics of the program is easier 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.

Definition 16 (verified bytecode) *Let $I_P \in Prog$ be an LP residual program for Int w.r.t. $\{\texttt{class}_1, \ldots, \texttt{class}_n\} \in 2^{Classes}$ and a method invocation specification M. Let $D_\alpha \in ADom$ be an abstract domain and $Assert_\alpha \in AApprox$ be the abstract intended semantics. We say that $(\{\texttt{class}_1, \ldots, \texttt{class}_n\}, M)$ is verified w.r.t. $Assert_\alpha$ in D_α if $AIVerifier(I_P, D_\alpha, Assert_\alpha)$ 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.

2.2.1 Run-time Error Freeness Analysis

For this analysis, we will focus on the `main` method of the example shown in Figure 2. Features used in the execution of this method include object creation, calls to static (`exp` and `fact`) and instance method (`setExp` and `setFact`) — which demonstrates that this approach is not restricted to intra-procedural analysis. The use of objects could in principle throw exceptions of type `NullPointerException` and the method `fact` exceptions of type `ArithmeticException`. Clearly, the execution of the `main` method will not produce any exception. However, the JVM is unaware of this and has to perform the corresponding run-time test. By using this approach one can statically verify that the previous code cannot issue such an exception (nor any other kind of run-time error). Since the data other than for the computation of exponential are known, the generated code is very similar to the partial evaluation of *exp* (see Figure 2.3). However, as there is the creation of an object in the heap and more code executed, the heap and the trace are respectively larger and longer. Figure 2.4 presents the result of the partial evaluation of the class `ExpFact` with respect to a call to the `static void main` method with both the base and the exponent unknown.

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., `ArithmeticException`, 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`.[3] The regular type `goodtrace` defines this notion of safety for the current example (for conciseness, bytecode instructions which do not appear in the example program have been omitted):

```
:- 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(ireturn_step_ok).
goodstep(if_icmp_step_continue).       goodstep(invokespecial_step_here_ok).
```

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

[3] Formally, this property is defined as a *regular unary logic* program, see [9].

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

main([A,B],
  st(heap(dynamicHeap([object(locationObject(className(packageName(''),shortClassName('ExpFact'))),
   [objectField(fieldSignature(fieldName(className(packageName(''),shortClassName('ExpFact')),
   shortFieldName('_fact')),primitiveType(int)),num(int(2))),objectField(fieldSignature(
   fieldName(className(packageName(''),shortClassName('ExpFact')),shortFieldName('_exp')),
   primitiveType(int)),num(int(1)))])]),staticHeap([])),fr(method(methodSignature(methodName(
   className(packageName(''),shortClassName('ExpFact')),shortMethodName(main)),[primitiveType(int),
   primitiveType(int)],none),bytecodeMethod(5,2,0,methodId('ExpFact_class',1),[exceptionHandler(
   className(packageName('java/lang/'),shortClassName('ArithmeticException')),19,29,32)]),
   final(false),static(true),public),34,[],[num(int(A)),num(int(B)),ref(loc(1)),num(int(2)),
   num(int(0))]),[]),
  [new_step_ok,dup_step_ok,invokespecial_step_here_ok,aload_step_ok,invokespecial_step_here_ok,
   return_step_ok,return_step_ok,astore_step_ok,iload_step,iload_step,invokestatic_step_ok,
   const_step_ok,istore_step_ok,iload_step,istore_step_ok,iload_step,if0_step_jump,iload_step,
   ireturn_step_ok,istore_step_ok,aload_step_ok,iload_step,invokevirtual_step_ok,aload_step_ok,
   iload_step,putfield_step_ok,return_step_ok,const_step_ok,invokestatic_step_ok,iload_step,
   const_step_ok,if_icmp_step_jump,iload_step,const_step_ok,if_icmp_step_continue,iload_step,
   iload_step,const_step_ok,ibinop_step_ok,invokestatic_step_ok,iload_step,const_step_ok,
   if_icmp_step_jump,iload_step,const_step_ok,if_icmp_step_jump,iload_step,if0_step_continue,
   const_step_ok,ireturn_step_ok,ibinop_step_ok,ireturn_step_ok,istore_step_ok,aload_step_ok,
   iload_step,invokevirtual_step_ok,aload_step_ok,iload_step,putfield_step_ok,return_step_ok,
   goto_step_ok,normal_end]) :-
        B=<0 .
main([B,C],A,[new_step_ok,dup_step_ok,invokespecial_step_here_ok,aload_step_ok,
  invokespecial_step_here_ok,return_step_ok,return_step_ok,astore_step_ok,iload_step,iload_step,
  invokestatic_step_ok,const_step_ok,istore_step_ok,iload_step,istore_step_ok,iload_step,
  if0_step_continue,iload_step,iload_step,ibinop_step_ok,istore_step_ok,iinc_step|D]) :-
        C>0,
        E is B,
        F is-1+C,
        execute_4_1(A,D,B,C,E,F) .

execute_4_1(
  st(heap(dynamicHeap([object(locationObject(className(packageName(''),shortClassName('ExpFact'))),
   [objectField(fieldSignature(fieldName(className(packageName(''),shortClassName('ExpFact')),
   shortFieldName('_fact')),primitiveType(int)),num(int(2))),objectField(fieldSignature(fieldName(
   className(packageName(''),shortClassName('ExpFact')),shortFieldName('_exp')),primitiveType(int)),
   num(int(C)))])]),staticHeap([])),fr(method(methodSignature(methodName(className(packageName(''),
   shortClassName('ExpFact')),shortMethodName(main)),[primitiveType(int),primitiveType(int)],none),
   bytecodeMethod(5,2,0,methodId('ExpFact_class',1),[exceptionHandler(className(packageName
   ('java/lang/'),shortClassName('ArithmeticException')),19,29,32)]),final(false),static(true),
   public),34,[],[num(int(A)),num(int(B)),ref(loc(1)),num(int(2)),num(int(0))]),[]),
 [goto_step_ok,iload_step,if0_step_jump,iload_step,ireturn_step_ok,istore_step_ok,
  aload_step_ok,iload_step,invokevirtual_step_ok,aload_step_ok,iload_step,putfield_step_ok,
  return_step_ok,const_step_ok,invokestatic_step_ok,iload_step,const_step_ok,if_icmp_step_jump,
  iload_step,const_step_ok,if_icmp_step_continue,iload_step,iload_step,const_step_ok,
  ibinop_step_ok,invokestatic_step_ok,iload_step,const_step_ok,if_icmp_step_jump,iload_step,
  const_step_ok,if_icmp_step_jump,iload_step,if0_step_continue,const_step_ok,ireturn_step_ok,
  ibinop_step_ok,ireturn_step_ok,istore_step_ok,aload_step_ok,iload_step,invokevirtual_step_ok,
  aload_step_ok,iload_step,putfield_step_ok,return_step_ok,goto_step_ok,normal_end],A,B,C,D) :-
        D=<0 .
execute_4_1(A,[goto_step_ok,iload_step,if0_step_continue,iload_step,iload_step,ibinop_step_ok,
  istore_step_ok,iinc_step|F],B,C,D,E) :-
        E>0,
        G is D*B,
        H is-1+E,
        execute_4_1(A,F,B,C,G,H) .
```

Figure 2.4: Complete Residual Program

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

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

Now, `CiaoPP` performs regular type analysis using, for example, the *eterms* domain [28]. This allows computing safe approximations of the success states of all predicates. After this, `CiaoPP` performs compile-time checking [27] of the `success` assertion above, comparing it with the assertions inferred by analysis, and produces as output the following assertion:

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

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*.

2.2.2 Termination and Cost Analyses

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 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 (C)LP is more advanced than that in imperative programming. Some well-known termination analysis systems for (C)LP are TerminWeb [6] and cTi [22]. 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 Figure 2.4. Let us also consider the following entry declaration:

```
:- entry main([B,C],A,D) : (int(B), int(C), var(A), var(D)).
```

which describes the valid external queries to the predicate `main/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 with respect to some norm which assigns values always greater or equal than zero for any term, and (2) the program is *rigid* with respect to the size of the corresponding argument (all instances of the term have the same size) then the program terminates. In Example 3, the only loop we have is for predicate `execute_4_1/6`. 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.

`CiaoPP` also offers an upper bound cost analysis [8] that can be used in order to prove termination. Due to a current limitation of this analysis, the residual program must *not* have accumulating parameters. Unfortunately, the partial evaluator integrated into `CiaoPP` generates residual program with accumulating parameters. Although the most studied problem is usually the opposite, that is to say, adding accumulative parameters in order to reduce computation time by allowing the compilator not to have to allocate a new execution frame for

the recursion and using the current one, a solution to remove this parameter is proposed in [18]. Because of a lack of time, we have not yet implemented this in CiaoPP. Figure 2.5 is a modified-by-hand version of the residual program in order to get rid of the accumulating parameters and to simplify the state to make the program easier to read. We consider the same entry assertion as before. The cost analysis can then infer the following property :

```
:- true pred execute_4_1(A,_1,D,_2)
          : ( term(A), num(_1), num(D), term(_2) )
         => ( num(A), num(_1), num(D), rt85(_2),
              size_ub(A,exp(int(_1),int(D))+1), size_ub(_1,int(_1)),
              size_ub(D,int(D)), size_ub(_2,8*int(D)+48) )
          + steps_ub(int(D)+1).
```

which describe states that `execute_4_1/4` is called with four parameters of type `term`, `num`, `num` and `term` (where `term` is anything and `num` a number), and that, on success, it returns respectively three numbers and a term of type `rt85` which correspond to a auto-generated type that represents the type of the trace. The assertions also gives upper bound on the data, where the first argument is bounded by the exponential of `_1` (the base) and `D` (the exponent), and the last argument (the trace) is bounded by 8 times the value of the exponent plus 48, which implies that it is linear in function of the exponent. Finally, after the sign `+` comes the upper bound of the cost of the predicate which is the value of the exponent plus 1. This proves termination of the predicate.

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

:- entry main([A,B],C,D) : (num(A),num(B),term(C),term(D)).

main([_A,B],1,[new_step_ok,dup_step_ok,invokespecial_step_here_ok,aload_step_ok,
  invokespecial_step_here_ok,return_step_ok,return_step_ok,astore_step_ok,iload_step,
  iload_step,invokestatic_step_ok,const_step_ok,istore_step_ok,iload_step,istore_step_ok,
  iload_step,if0_step_jump,iload_step,ireturn_step_ok,istore_step_ok,aload_step_ok,iload_step,
  invokevirtual_step_ok,aload_step_ok,iload_step,putfield_step_ok,return_step_ok,const_step_ok,
  invokestatic_step_ok,iload_step,const_step_ok,if_icmp_step_jump,iload_step,const_step_ok,
  if_icmp_step_continue,iload_step,iload_step,const_step_ok,ibinop_step_ok,invokestatic_step_ok,
  iload_step,const_step_ok,if_icmp_step_jump,iload_step,const_step_ok,if_icmp_step_jump,
  iload_step,if0_step_continue,const_step_ok,ireturn_step_ok,ibinop_step_ok,ireturn_step_ok,
  istore_step_ok,aload_step_ok,iload_step,invokevirtual_step_ok,aload_step_ok,iload_step,
  putfield_step_ok,return_step_ok,goto_step_ok,normal_end]) :-
        B=<0 .
main([B,C],A,[new_step_ok,dup_step_ok,invokespecial_step_here_ok,aload_step_ok,
  invokespecial_step_here_ok,return_step_ok,return_step_ok,astore_step_ok,iload_step,
  iload_step,invokestatic_step_ok,const_step_ok,istore_step_ok,iload_step,istore_step_ok,
  iload_step,if0_step_continue,iload_step,iload_step,ibinop_step_ok,istore_step_ok,
  iinc_step|D]) :-
        C>0,
        F is-1+C,
        execute_4_1(A,B,F,D) .

execute_4_1(1,_,D,[goto_step_ok,iload_step,if0_step_jump,iload_step,ireturn_step_ok,
  istore_step_ok,aload_step_ok,iload_step,invokevirtual_step_ok,aload_step_ok,iload_step,
  putfield_step_ok,return_step_ok,const_step_ok,invokestatic_step_ok,iload_step,const_step_ok,
  if_icmp_step_jump,iload_step,const_step_ok,if_icmp_step_continue,iload_step,iload_step,
  const_step_ok,ibinop_step_ok,invokestatic_step_ok,iload_step,const_step_ok,if_icmp_step_jump,
  iload_step,const_step_ok,if_icmp_step_jump,iload_step,if0_step_continue,const_step_ok,
  ireturn_step_ok,ibinop_step_ok,ireturn_step_ok,istore_step_ok,aload_step_ok,iload_step,
  invokevirtual_step_ok,aload_step_ok,iload_step,putfield_step_ok,return_step_ok,goto_step_ok,
  normal_end]) :-
        D=<0 .
execute_4_1(A,B,E,[goto_step_ok,iload_step,if0_step_continue,iload_step,iload_step,
  ibinop_step_ok,istore_step_ok,iinc_step|F]) :-
        E>0,
        H is-1+E,
        execute_4_1(R,B,H,F),
A is R*B.
```

Figure 2.5: Residual Program without Accumulating Parameters

Conclusion

From a formal (but incomplete) specification of the JVML (Bicolano), we have specified a language (JVML_r) and implemented the corresponding interpreter. This interpreter has been slightly modified in order to be able to start the execution of whatever method we want (*method invocation specification*) and to be partially evaluable (some adaptations have been done to fit within the limitations of the partial evaluator). By partially evaluating this interpreter with respect to a JVML_r program, we obtain a LP program from which we can infer rich properties as termination and run-time error freeness. We should then be able to use other existing analyses on those residual programs in order to infer other useful property. This is the main contribution of this work, to allow the use of all existing analyses for LP to check and verify Java bytecode (although, some analyses can be meaningless on Java bytecode like determinacy). As we use `CiaoPP` for the analyses, its *Abstraction Carrying Code* [1] (ACC) architecture can also be used to generate certificates for the residual program. Now, the main limitations of this approach is that, as the certificate is generated for the residual program, the consumer has to re-generated the residual program and the partial evaluator has to be part of the trusted code. Partial evaluation can be costly and solutions have to be found to reduce its cost on the consumer side (e.g., providing the fixpoint), or removing it of the checking process by "translating" the certificate on the LP program to the Java bytecode program. Partial evaluation is also costly on the producer side. To have a more time- and cost-efficient partial evaluation, as it is always applied on the same interpreter and only the source program changes, it might be possible to give some indications to the partial evaluator so it does not need to *look for* all the predicates that can be unfolded, as some general conditions can be stated under which some predicates can be *evaluated* in one step, or have to be kept as is. For larger programs, we hope the modular architecture of the partial evaluator and analyzers of `CiaoPP` will allow to focus on small parts of large programs.

Bibliography

[1] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code: A Model for Mobile Code Safety. Technical Report CLIP7/2005.0, Technical University of Madrid, School of Computer Science, UPM, July 2005.

[2] 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.

[3] G. Barthe. Mobius european project. http://mobius.inria.fr.

[4] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *Journal of Logic Programming*, 6(1 & 2):135–162, January 1989.

[5] 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/`.

[6] 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.

[7] 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.

[8] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[9] 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.

[10] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[11] Yoshihiko Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[12] J. Gallagher. Transforming logic programs by specializing interpreters. In *Proc. of the 7th. European Conference on Artificial Intelligence*, 1986.

[13] Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.

[14] 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.

[15] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS*, 22(2):187–223, March 2000.

[16] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[17] 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.

[18] Armin Kühnemann, Robert Glück, and Kazuhiko Kakehi. Relating accumulative and non-accumulative functional programs. In *RTA '01: Proceedings of the 12th International Conference on Rewriting Techniques and Applications*, pages 154–168, London, UK, 2001. Springer-Verlag.

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

[20] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(3-4):217–242, 1991.

[21] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[22] 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.

[23] 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.

[24] D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

Appendix A

Implementation in `Ciao` of the Dynamic Semantics of JVML_{r}

```
step(exception_caught, P,
    stE(H,frE(M,PC,Loc,L),SF),
    st(H,fr(M,PCb,[ref(Loc)],L),SF)):-
 method_body(M,BM),
 bytecodeMethod_exceptionHandlers(BM,ExL),
 lookup_handlers(P,ExL,H,PC,Loc,PCb).
step(exception_uncaught, P,
    stE(H,frE(M,PC,Loc,_L),[fr(Mb,PCb,_Sb,Lb)|SF]),
    stE(H,frE(Mb,PCb,Loc,Lb),SF)):-
 method_body(M,BM),
 bytecodeMethod_exceptionHandlers(BM,ExL),
 \+ exception_uncaught_step_cond(P,ExL,H,PC,Loc).

step(aaload_step_ok, _P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[Val|S],L),SF)):-
 instructionAt(M,PC,aaload),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,refType(_RT))),
 0 =< I,
 I < Length,
 heap_get(H,arrayElement(Loc,I),Val).
step(aaload_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Loc,L),SF)):-
 instructionAt(M,PC,aaload),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Loc,Hb).
step(aaload_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,aaload),
 heap_typeof(H,Loc,locationArray(Length,refType(_RT))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(aastore_step_ok, P,
    st(H,fr(M,PC,[Val,num(int(I)),ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,aastore),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,TP)),
 assign_compatible(P,H,Val,TP),
 0 =< I, I < Length,
 heap_update(H,arrayElement(Loc,I),Val,Hb).
step(aastore_step_NullPointerException, P,
    st(H,fr(M,PC,[_Val,num(int(_I)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,aastore),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(aastore_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[_Val,num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,aastore),
 heap_typeof(H,Loc,locationArray(Length,refType(_RT))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).
```

Figure A.1: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 1)

```
step(aastore_step_ArrayStoreException, P,
    st(H,fr(M,PC,[Val,num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,aastore),
 heap_typeof(H,Loc,locationArray(Length,TP)),
 0 =< I, I < Length,
 \+ assign_compatible(P,H,Val,TP),
 arrayStoreException(ASE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,ASE)),Locb,Hb).

step(aconst_null, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,[null|S],L),SF)):-
 instructionAt(M,PC,aconst_null),
 next(M,PC,PCb).

step(aload_step_ok, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,[Val|S],L),SF)):-
 instructionAt(M,PC,aload(X)),
 next(M,PC,PCb),
 localVar_get(L,X,Val),
 isReference(Val).

step(anewarray_step_ok, P,
    st(H,fr(M,PC,[num(int(Length))|S],L),SF),
    st(Hb,fr(M,PCb,[ref(Loc)|S],L),SF)):-
 instructionAt(M,PC,anewarray(T)),
 next(M,PC,PCb),
 0 =< Length,
 heap_new(H,P,locationArray(Length,T),Loc,Hb).
step(anewarray_step_NegativeArraySizeException, P,
    st(H,fr(M,PC,[num(int(Length))|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,anewarray(_T)),
 Length < 0,
 negativeArraySizeException(NASE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NASE)),Locb,Hb).

step(areturn_step_ok, P,
    st(H,fr(M,PC,[Val|_S],_L),CallStack),
    st(H,fr(Mb,PCbb,[Val|Sb],Lb),SF)):-
 instructionAt(M,PC,areturn),
 nonvar(CallStack),
 CallStack = [fr(Mb,PCb,Sb,Lb)|SF],
 next(Mb,PCb,PCbb),
 method_signature(M,MSig),
 methodSignature_result(MSig,refType(RT)),
 assign_compatible(P,H,Val,refType(RT)).

step(arraylength_step_ok, _P,
    st(H,fr(M,PC,[ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(Length))|S],L),SF)):-
 instructionAt(M,PC,arraylength),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,_TP)).
step(arraylength_step_NullPointerException, P,
    st(H,fr(M,PC,[null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,arraylength),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
```

Figure A.2: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 2)

```
step(astore_step_ok,_P,
    st(H,fr(M,PC,[Val|S],L),SF),
    st(H,fr(M,PCb,S,Lb),SF)):-
 instructionAt(M,PC,astore(X)),
 isReference(Val),
 next(M,PC,PCb),
 localVar_update(L,X,Val,Lb).

step(athrow_step, _P,
    st(H,fr(M,PC,[ref(Loc)|_S],L),SF),
    stE(H,frE(M,PC,Loc,L),SF)):-
 instructionAt(M,PC,athrow).
step(athrow_step_NullPointerException, P,
    st(H,fr(M,PC,[null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,athrow),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).

step(baload_step_ok, _P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(B))|S],L),SF)):-
 instructionAt(M,PC,baload),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(TP))),
 0 =< I,
 I < Length,
 (TP=boolean;TP=byte),
 heap_get(H,arrayElement(Loc,I),num(byte(B))).
step(baload_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,baload),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(baload_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,baload),
 heap_typeof(H,Loc,locationArray(Length,_RT)),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(bastore_step_ok, _P,
    st(H,fr(M,PC,[num(int(Ib)),num(int(I)),ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,bastore),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(byte))),
 0 =< I, I < Length,
 i2b(Ib,B),
 heap_update(H,arrayElement(Loc,I),num(byte(B)),Hb).
step(bastore_step_ok, _P,
    st(H,fr(M,PC,[num(int(Ib)),num(int(I)),ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,bastore),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(boolean))),
 0 =< I, I < Length,
 i2bool(Ib,B),
 heap_update(H,arrayElement(Loc,I),num(byte(B)),Hb).
```

Figure A.3: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 3)

```
step(bastore_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(_I)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,bastore),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(bastore_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,bastore),
 heap_typeof(H,Loc,locationArray(Length,refType(_RT))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(checkcast_step_ok, P,
    st(H,fr(M,PC,[Val|S],L),SF),
    st(H,fr(M,PCb,[Val|S],L),SF)):-
 instructionAt(M,PC,checkcast(T)),
 next(M,PC,PCb),
 assign_compatible(P,H,Val,refType(T)).
step(checkcast_step_ClassCastException, P,
    st(H,fr(M,PC,[Val|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,checkcast(T)),
 \+ assign_compatible(P,H,Val,refType(T)),
 classCastException(CCE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,CCE)),Locb,Hb).

step(const_step_ok, _P,
    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).

step(dup_step_ok, _P,
    st(H,fr(M,PC,[V|S],L),SF),
    st(H,fr(M,PCb,[V,V|S],L),SF)):-
 instructionAt(M,PC,dup),
 next(M,PC,PCb).

step(dup_x1_step_ok, _P,
    st(H,fr(M,PC,[V1,V2|S],L),SF),
    st(H,fr(M,PCb,[V1,V2,V2|S],L),SF)):-
 instructionAt(M,PC,dup_x1),
 next(M,PC,PCb).

step(dup_x2_step_ok, _P,
    st(H,fr(M,PC,[V1,V2,V3|S],L),SF),
    st(H,fr(M,PCb,[V1,V2,V3,V1|S],L),SF)):-
 instructionAt(M,PC,dup_x2),
 next(M,PC,PCb).

step(dup2_step_ok, _P,
    st(H,fr(M,PC,[V1,V2|S],L),SF),
    st(H,fr(M,PCb,[V1,V2,V1,V2|S],L),SF)):-
 instructionAt(M,PC,dup2),
 next(M,PC,PCb).

step(dup2_x1_step_ok, _P,
    st(H,fr(M,PC,[V1,V2,V3|S],L),SF),
    st(H,fr(M,PCb,[V1,V2,V3,V1,V2|S],L),SF)):-
 instructionAt(M,PC,dup2_x1),
 next(M,PC,PCb).
```

Figure A.4: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 4)

```
step(dup2_x2_step_ok, _P,
    st(H,fr(M,PC,[V1,V2,V3,V4|S],L),SF),
    st(H,fr(M,PCb,[V1,V2,V3,V4,V1,V2|S],L),SF)):-
 instructionAt(M,PC,dup2_x2),
 next(M,PC,PCb).

step(getfield_step_ok, _P,
    st(H,fr(M,PC,[ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[V|S],L),SF)):-
 instructionAt(M,PC,getfield(F)),
 next(M,PC,PCb),
 heap_get(H,dynamicField(Loc,F),V).
step(getfield_step_NullPointerException, P,
    st(H,fr(M,PC,[null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,getfield(_F)),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).

step(getstatic_step_ok, P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,[V|S],L),SF)):-
 instructionAt(M,PC,getstatic(F)),
 next(M,PC,PCb),
 isStatic(P,F),
 heap_get(H,staticField(F),V).

step(goto_step_ok, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,goto(O)),
 PCb is PC+O.

step(i2b_step_ok, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,[num(int(Ib))|S],L),SF)):-
 instructionAt(M,PC,i2b),
 next(M,PC,PCb),
 i2b(I,Ib).

step(i2s_step_ok, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,[num(int(Ib))|S],L),SF)):-
 instructionAt(M,PC,i2s),
 next(M,PC,PCb),
 i2s(I,Ib).

step(ibinop_step_ok, _P,
    st(H,fr(M,PC,[num(int(I2)),num(int(I1))|S],L),SF),
    st(H,fr(M,PCb,[num(int(R))|S],L),SF)):-
 instructionAt(M,PC,ibinop(Op)),
 ibinop_step_cond(Op,I2),
 next(M,PC,PCb),
 semBinopInt(Op,I1,I2,R).
step(ibinop_ArithmeticException, P,
    st(H,fr(M,PC,[num(int(0)),num(int(_I1))|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,ibinop(Op)),
 (Op=divInt;Op=remInt),
 arithmeticException(AE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,AE)),Locb,Hb).
```

Figure A.5: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 5)

```
step(iaload_step_ok, _P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(Ib))|S],L),SF)):-
 instructionAt(M,PC,iaload),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(int))),
 0 =< I,
 I < Length,
 heap_get(H,arrayElement(Loc,I),num(int(Ib))).
step(iaload_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,iaload),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(iaload_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,iaload),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(int))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(iastore_step_ok, _P,
    st(H,fr(M,PC,[num(int(Ib)),num(int(I)),ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,iastore),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(int))),
 0 =< I, I < Length,
 heap_update(H,arrayElement(Loc,I),num(int(Ib)),Hb).
step(iastore_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(_I)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,iastore),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(iastore_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,iastore),
 heap_typeof(H,Loc,locationArray(Length,refType(int))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(if_acmpeq_step_jump, _P,
    st(H,fr(M,PC,[V1,V1|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_acmpeq(O)),
 isReference(V1),
 PCb is PC+O.
step(if_acmpeq_step_continue, _P,
    st(H,fr(M,PC,[V1,V2|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_acmpeq(_O)),
 next(M,PC,PCb),
 isReference(V1),
 isReference(V2),
 V1 =\= V2.
```

Figure A.6: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 6)

```
step(if_acmpne_step_jump, _P,
    st(H,fr(M,PC,[V1,V2|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_acmpne(O)),
 isReference(V1),
 isReference(V2),
 V1 =\= V2,
 PCb is PC+O.
step(if_acmpne_step_continue, _P,
    st(H,fr(M,PC,[V1,V1|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_acmpeq(_O)),
 next(M,PC,PCb),
 isReference(V1).

step(if_icmp_step_jump, _P,
    st(H,fr(M,PC,[num(int(I2)),num(int(I1))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_icmp(Cmp,O)),
 semCompInt(Cmp,I1,I2),
 PCb is PC+O.
step(if_icmp_step_continue, _P,
    st(H,fr(M,PC,[num(int(I2)),num(int(I1))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if_icmp(Cmp,_O)),
 next(M,PC,PCb),
 noSemCompInt(Cmp,I1,I2).

step(if0_step_jump, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if0(Cmp,O)),
 semCompInt(Cmp,I,0),
 PCb is PC+O.
step(if0_step_continue, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,if0(Cmp,_O)),
 noSemCompInt(Cmp,I,0),
 next(M,PC,PCb).

step(ifnonnull_step_jump, _P,
    st(H,fr(M,PC,[ref(_Loc)|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,ifnonnull(O)),
 PCb is PC+O.
step(ifnonnull_step_continue, _P,
    st(H,fr(M,PC,[null|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,ifnonnull(_O)),
 next(M,PC,PCb).

step(ifnull_step_jump, _P,
    st(H,fr(M,PC,[null|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,ifnull(O)),
 PCb is PC+O.
step(ifnull_step_continue, _P,
    st(H,fr(M,PC,[ref(_Loc)|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,ifnull(_O)),
 next(M,PC,PCb).

step(iinc_step, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,S,Lb),SF)):-
 instructionAt(M,PC,iinc(X,Z)),
 next(M,PC,PCb),
 localVar_get(L,X,num(int(I))),
 semBinopInt(addInt,I,Z,R),
 localVar_update(L,X,num(int(R)),Lb).
```

Figure A.7: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 7)

```
step(iload_step, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,[num(int(I))|S],L),SF)):-
 instructionAt(M,PC,iload(X)),
 next(M,PC,PCb),
 localVar_get(L,X,num(int(I))).

step(ineg_step, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,[num(int(Ib))|S],L),SF)):-
 instructionAt(M,PC,ineg),
 next(M,PC,PCb),
 negInt(I,Ib).

step(instanceof_step_ok1, P,
    st(H,fr(M,PC,[ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(1))|S],L),SF)):-
 instructionAt(M,PC,instanceof(T)),
 next(M,PC,PCb),
 assign_compatible(P,H,ref(Loc),T).
step(instanceof_step_ok2, P,
    st(H,fr(M,PC,[ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(0))|S],L),SF)):-
 instructionAt(M,PC,instanceof(T)),
 next(M,PC,PCb),
 \+ assign_compatible(P,H,ref(Loc),T).

step(invokespecial_step_here_ok, P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(Meth,PCb,[],Lb),[fr(M,PC,Sb,L)|SF])):-
 instructionAt(M,PC,invokespecial(Mid)),
 methodSignature_name(Mid,methodName(MidCn,_)),
 resolve_method(P,MidCn,Mid,Meth),
 method_signature(Meth,MethSig),
 methodSignature_parameters(MethSig,Param),
 length(Param,NbParam),
 length(Args,NbParam),
 append(Args,[ref(Loc)|Sb],S),
 heap_typeof(H,Loc,locationObject(Cn)),
 method_signature(Meth,Meths),
 methodSignature_name(Meths,Methn),
 Methn = methodName(Methcl,shortMethodName('<init>')),
 method_signature(M,Msig),
 methodSignature_name(Msig,methodName(CCn,_)),
 ( ( method_visibility(Meth,protected),
      subclass_name(P,CCn,Methcl),
      subclass_name(P,Cn,CCn))
  ;
     (method_visibility(Meth,Visib),
      Visib \= protected)),
 compatible_param(P,H,Args,Param),
 method_body(Meth,BMeth),
 bytecodeMethod_firstAddress(BMeth,PCb),
 bytecodeMethod_localVarSize(BMeth,Llength),
 RLlength is Llength - NbParam -1,
 length(RL,RLlength),
 init_localVar(RL,RLlength),
 reverse(Args,RL,Lb1),
 Lb = [ref(Loc)|Lb1].
```

Figure A.8: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 8)

```
step(invokespecial_step_NullPointerException, P,
    st(H,fr(M,PC,S,L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,invokespecial(Mid)),
 methodSignature_parameters(Mid,Param),
 length(Param,NbParam),
 length(Args,NbParam),
 append(Args,[null|_Sb],S),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).

step(invokestatic_step_ok, P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(Mb,PCb,[],Lb),[fr(M,PC,Sb,L)|SF])):-
 instructionAt(M,PC,invokestatic(Mid)),
 methodSignature_name(Mid,methodName(CN,_SMN)),
 resolve_method(P,CN,Mid,Mb),
 method_isStatic(Mb),
 method_body(Mb,Bm),
 bytecodeMethod_firstAddress(Bm,PCb),
 methodSignature_parameters(Mid,Param),
 length(Param,NbParam),
 length(Args,NbParam),
 append(Args,Sb,S),
 bytecodeMethod_localVarSize(Bm,Llength),
 RLlength is Llength - NbParam,
 length(RL,RLlength),
 init_localVar(RL,RLlength),
 reverse(Args,RL,Lb).

step(invokevirtual_step_ok, P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(Mb,PCb,[],Lb),[fr(M,PC,Sb,L)|SF])):-
 instructionAt(M,PC,invokevirtual(Mid)),
 methodSignature_name(Mid,methodName(MidCn,_)),
 resolve_method(P,MidCn,Mid,Meth),
 method_signature(Meth,MethSig),
 methodSignature_name(MethSig,methodName(MethCln,Methsmn)),
 Methsmn \= shortMethodName('<init>'),
 Methsmn \= shortMethodName('<clinit>'),
 methodSignature_parameters(MethSig,Param),
 length(Param,NbParam),
 length(Args,NbParam),
 append(Args,[ref(Loc)|Sb],S),
 heap_typeof(H,Loc,locationObject(Cn)),
 method_signature(M,Msig),
 methodSignature_name(Msig,methodName(CCn,_)),
 ( ( method_visibility(Meth,protected),
     subclass_name(P,CCn,MethCln),
     subclass_name(P,Cn,CCn))
  ;
   ( method_visibility(Meth,Visib),
     Visib \= protected)),
 lookup(P,Cn,MethSig,Mb),
 method_body(Mb,Bm),
 bytecodeMethod_firstAddress(Bm,PCb),
 bytecodeMethod_localVarSize(Bm,Llength),
 RLlength is Llength - NbParam -1,
 length(RL,RLlength),
 init_localVar(RL,RLlength),
 reverse(Args,RL,Lb1),
 Lb = [ref(Loc)|Lb1].
```

Figure A.9: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 9)

```
step(invokevirtual_step_NullPointerException, P,
    st(H,fr(M,PC,S,L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,invokevirtual(Mid)),
 methodSignature_parameters(Mid,Param),
 length(Param,NbParam),
 length(Args,NbParam),
 append(Args,[null|_Sb],S),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).

step(ireturn_step_ok,_P,
    st(H,fr(M,PC,[num(int(I))|_S],_L),CallStack),
    st(H,fr(Mb,PCbb,[num(int(I))|Sb],Lb),SF)):-
 instructionAt(M,PC,ireturn),
 nonvar(CallStack),
 CallStack = [fr(Mb,PCb,Sb,Lb)|SF],
 next(Mb,PCb,PCbb),
 method_signature(M,MSig),
 methodSignature_result(MSig,primitiveType(_)).

step(istore_step_ok, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,Lb),SF)):-
 instructionAt(M,PC,istore(X)),
 next(M,PC,PCb),
 localVar_update(L,X,num(int(I)),Lb).

step(lookupswitch_step_ok1, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,lookupswitch(_Def,ListKey)),
 member((I,O),ListKey),
 PCb is PC+O.
step(lookupswitch_step_ok2, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,lookupswitch(Def,ListKey)),
 \+ member((I,_O),ListKey),
 PCb is PC+Def.

step(new_step_ok, P,
    st(H,fr(M,PC,S,L),SF),
    st(Hb,fr(M,PCb,[ref(Loc)|S],L),SF)):-
 instructionAt(M,PC,new(C)),
 next(M,PC,PCb),
 heap_new(H,P,locationObject(C),Loc,Hb).

step(newarray_step_ok, P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(Hb,fr(M,PCb,[ref(Loc)|S],L),SF)):-
 instructionAt(M,PC,newarray(T)),
 next(M,PC,PCb),
 0 =< I,
 heap_new(H,P,locationArray(I,T),Loc,Hb).
step(newarray_step_NegativeArraySizeException, P,
    st(H,fr(M,PC,[num(int(I))|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,newarrayt(_T)),
 I<0,
 negativeArraySizeException(NASE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NASE)),Locb,Hb).
```

Figure A.10: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 10)

```
step(nop_step_ok, _P,
    st(H,fr(M,PC,S,L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,nop),
 next(M,PC,PCb).

step(pop_step_ok, _P,
    st(H,fr(M,PC,[_V|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,pop),
 next(M,PC,PCb).

step(pop2_step_ok, _P,
    st(H,fr(M,PC,[_V1,_V2|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,pop2),
 next(M,PC,PCb).

step(putfield_step_ok, P,
    st(H,fr(M,PC,[V,ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,putfield(F)),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationObject(Cn)),
 defined_field(P,Cn,F),
 fieldSignature_type(F,FT),
 assign_compatible(P,H,V,FT),
 heap_update(H,dynamicField(Loc,F),V,Hb).
step(putfield_step_NullPointerException, P,
    st(H,fr(M,PC,[_V,null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,putfield(_F)),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).

step(putstatic_step_ok, P,
    st(H,fr(M,PC,[V|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,putstatic(F)),
 next(M,PC,PCb),
 isStatic(P,F),
 fieldSignature_type(F,FT),
 assign_compatible(P,H,V,FT),
 heap_update(H,staticField(F),V,Hb).

step(return_step_ok, _P,
    st(H,fr(M,PC,_S,_L),CallStack),
    st(H,fr(Mb,PCbb,Sb,Lb),SF)):-
 instructionAt(M,PC,return),
 nonvar(CallStack),
 CallStack = [fr(Mb,PCb,Sb,Lb)|SF],
 next(Mb,PCb,PCbb),
 method_signature(M,MSig),
 methodSignature_result(MSig,none).

step(saload_step_ok, _P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|S],L),SF),
    st(H,fr(M,PCb,[num(int(Sh))|S],L),SF)):-
 instructionAt(M,PC,saload),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(short))),
 0 =< I,
 I < Length,
 heap_get(H,arrayElement(Loc,I),num(short(Sh))).
```

Figure A.11: Implementation in `Ciao` of the dynamic semantics of JVML$_r$ (part 11)

```
step(saload_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,saload),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(saload_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,saload),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(short))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).
step(sastore_step_ok, _P,
    st(H,fr(M,PC,[num(int(ISh)),num(int(I)),ref(Loc)|S],L),SF),
    st(Hb,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,sastore),
 next(M,PC,PCb),
 heap_typeof(H,Loc,locationArray(Length,primitiveType(short))),
 0 =< I, I < Length,
 i2s(ISh,Sh),
 heap_update(H,arrayElement(Loc,I),num(short(Sh)),Hb).
step(sastore_step_NullPointerException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(_I)),null|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,sastore),
 nullPointerException(NPE),
 javaLang(JL),
 heap_new(H,P,locationObject(className(JL,NPE)),Locb,Hb).
step(sastore_step_ArrayIndexOutOfBoundsException, P,
    st(H,fr(M,PC,[num(int(_)),num(int(I)),ref(Loc)|_S],L),SF),
    stE(Hb,frE(M,PC,Locb,L),SF)):-
 instructionAt(M,PC,sastore),
 heap_typeof(H,Loc,locationArray(Length,refType(_RT))),
 (I < 0 ; I >= Length),
 javaLang(JL),
 arrayIndexOutOfBoundsException(AIOOBE),
 heap_new(H,P,locationObject(className(JL,AIOOBE)),Locb,Hb).

step(swap_step_ok, _P,
    st(H,fr(M,PC,[V1,V2|S],L),SF),
    st(H,fr(M,PCb,[V2,V1|S],L),SF)):-
 instructionAt(M,PC,swap),
 next(M,PC,PCb).

step(tableswitch_step_ok1, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,tableswitch(Def,Low,High,List_offset)),
 (I < Low ; High < I),
 check((length(List_offset,N), N = High - Low +1)),
 PCb is PC + Def.
step(tableswitch_step_ok2, _P,
    st(H,fr(M,PC,[num(int(I))|S],L),SF),
    st(H,fr(M,PCb,S,L),SF)):-
 instructionAt(M,PC,tableswitch(_Def,Low,High,List_offset)),
 Low =< I,
 I =< High,
 check((length(List_offset,N), N = High - Low +1)),
 Nth is I-Low+1,
 nth(Nth,List_offset,O),
 PCb is PC+O.
```

Figure A.12: Implementation in `Ciao` of the dynamic semantics of JVML_r (part 12)