THESIS / THÈSE

MASTER IN COMPUTER SCIENCE

A declarative approach to Java Virtual Machine modelisation and Bytecode execution

Laffineur, Gérome

Award date:
2017

Awarding institution:
University of Namur

Université de Namur
Faculty of Computer Science
Academic Year 2016–2017

A declarative approach to Java Virtual Machine modelisation and Bytecode execution.

Laffineur Gérôme

Internship mentors: PAYET Étienne & MESNARD Frédéric

Supervisor: _____ (Signed for Release Approval - Study Rules art. 40)
VANHOOF Wim

A thesis submitted in the partial fulfillment of the requirements
for the degree of Master of Computer Science at the Université of Namur

Abstract

Algorithm recognition, which is the problem of verifying whether a program implements a given algorithm, is an important topic in program analysis. This work is in the continuity of the framework for algorithm recognition in binary code defined in [25]. This article describes a transformation based approach to compare algorithms written in Horn clauses. We propose a decompiler that translate Java bytecode programs (.jar) into a declarative CLP representation based on Horn clauses. The first purpose of this decompiler is to act as the front-end of the framework described in [25]. The particularity of the approach presented here is that we propose a direct translation between bytecode instructions and the declarative representation (i.e. without an intermediate representation).

Keywords

Decompiler, Java Virtual Machine (JVM), Constraint Logic Programming (CLP),Horn clauses, Bytecode.

Preface

This paper is submitted for the Master's Degree of Computer Sciences at the University of Namur. The research described here was conducted under the supervision of Professor Mr.Wim Vanhoof from the University of Namur and the Professors Étienne Payet and Frederic Mesnard from the University of La Reunion between September 2016 and June 2017.

I am particularly grateful to my Professor and supervisor Mr.**Wim Vanhoof** from the University of Namur without whom the internship related to this project and this unusual experience would not have taken place. Thanks to his experience and his craving for field of research, Professor Vanhoof was of a great assistance to me. Despite of the distance, Mr. Vanhoof has shown a great implication during the project progress and it was a pleasure to work in his company.

I would also thank my internship mentors Mrs. **Étienne Payet** and **Frédéric Mesnard** from the University of La Réunion for their hospitality and their advices during the internship.

I also wish to thank Mr.**François-Xavier Fievez** from the University of Namur for the proofreading of this work.

Finally I would also thank the team of the computer sciences laboratory from the University of La Réunion and the all of the administrative staff of the University of Namur for their involvement.

Laffineur Gérome

June 2017

Contents

Chapter 1

Introduction

In this work we propose a decompiler that translates Java Virtual Machine bytecode into a universal declarative representation. Having a declarative representation of a program is very useful to perform various analysis on that program. The decompiler proposed in this work was developed to act as a part of a framework for algorithm recognition in binary code [25].

Being able to perform some algorithm recognition analysis can have multiple applications. Thanks to the universal representation provided by our decompiler, one would be able to extract the algorithmic core of a program. Thus, we could assert that two programs that seems different by their structure of their code have the same algorithmic core. An evident application of this process is plagiarism detection [27]. A comparison could also be done between various versions of a program. During the evolution of a program, it could be interesting to compare its different versions in order to ensure that a given algorithm is not altered by some code modifications. This kind of regression test would be particularly interesting for debugging and to verify the consistency of a program with respect to some specifications [27]. More generally, algorithm recognition could also help for program comprehension, at the basis of some reverse engineering processes [2] [27]. It is also possible to optimise a program by replacing the code of a known algorithm by a more efficient implementation of this algorithm [27].

Algorithm recognition is just one possible application of our work. Various analyses can be performed from the declarative representation of a program. Indeed, analysis tools have been largely developed for declarative representations and various breakthroughs have been achieved in this domain [11]. Some analysis tools also plays a role when it comes to deliver proper software. Several program analysis tools have emerged in the recent past years to tackle the issue of the so called software crisis. Those tools can really help with the automation of related concerns. For instance, we can find analysis tools that ensure automatically that a program meet its requirements by verifying some specified properties [11].

With respect to that, we can assert that our work on the declarative representation of Java compiled programs and the produced decompiler could be at the very basis of numerous analyses in a wide range of domains.

1.1 Context & Motivations

1.1.1 The framework

The work presented in this paper is part of a wider approach that aims to develop a framework for algorithm recognition in binary code [25]. A large range of applications could take advantages of verifying whether a program implements a given algorithm or not [26]. Indeed, algorithm recognition is an important topic in program analysis [25] and numerous applications can be found in diverse areas such as plagiarism detection [29], malware detection [28] and even more advanced analyses and optimisations [18].

The approach proposed by the framework is fully described in [25]. This approach first translates binary code into Horn clauses. Two programs are considered as implementing the same algorithm if their Horn clauses representations can be reduced to a single common set of Horn clauses by means of a sequence of transformations [25].

The framework was created to study the process of algorithm recognition from the angle of binary code for good reasons. Even if other approaches based on source code can be totally justified and are sufficient in certain cases such as programming tutoring - ie. assessing whether students have correctly implemented a particular algorithm [23] - they become useless without any access to source code. As an example, if we imagine the case where a company wants to verify if a competitor's software program uses a given algorithm. Then the competitor's software is might be written in a different programming language and is probably only available as binary code. Therefore, the verification must be performed directly over the competitor's software binary code [25] [29].

It is important to note that the notion of two algorithms being the same is not easily defined and remains subject to debate [8]. It is often considered that programs implement the same algorithm if it is possible to reduce one to another by a given sequence of syntactical transformations [8].

A particularity of the framework surrounding the work that will be presented here is that it uses Horn clauses to represent a model of the algorithm as well as of the compiled code under scrutiny. Horn clauses are simply a disjunction of literals with a most one negated literal. Every logic program clause is a Horn clause [27]. Using Horn clauses has several advantages. This universal intermediate language is a suitable abstraction between binary code and a more high-level programming language. Moreover, Horn clauses are an already well-known formalism that has proven itself to be suitable for a lot of different program analysis. [25].

More detailed information about the general framework can be found in [25].

1.1.2 Extending the framework to handle Java Bytecode

The framework was first developed to work with Android binary code (Dalvik) as a starting point, but with the idea that the approach could be extended to other assembly languages [25]. This is where our work fits in. We developed an independent decompiler which is able to translate a subset of Java Bytecode programs into a universal Horn clauses representation. Due to this, the framework is no longer limited to a subset of Android programs and can handle some Java programs as well. From a scientific point of view it was interesting to encounter problems related to direct linking between Java Bytecode and an executable declarative representation. One of the roles of this paper is to report those problems as well as methods used to resolve them (section 3.1). An advantage of this extension is that Java programs are much easier to set up than Android programs if one wants to create some test cases for the framework. Android development is a little bit more complex as it requires some extension libraries. In addition, Android applications have special lifecycles, there is no main function and the application contains activities with special functions that must be overwritten. In the Java settings, one doesn't have to worry about all the mobile-related settings surrounding the core of the application. Another great contribution to the framework described in [25] is that the tool developed for Java Bytecode decompilation handles a larger scope of input programs in terms of features than the original Dalvik decompiler (section 1.2.2).

Thus, the work presented here is basically the development of a tool to decompile Java Bytecode into Horn clauses. A great part of this work was dedicated to the redefinition and adaptation of a declarative formalism to represent bytecode instructions. However our decompiler is not aimed to be a full-featured application, only a subset of Java programs were meant to be taken in account. Indeed, not all Java Bytecode instructions were considered. Here we focus on programs that work only with numbers (all Java's primitives types except 'char'). We also deliberately put aside thread and exceptions related instructions as well as bitwise operators.

The decompiler has been developed for the purpose of algorithm recognition and similarity analyses. In this respect, the produced output had to be suitable for further transformations. Series of transformations to perform analysis often require a lot of CPU time [27]. Thus, an effort has been made to minimize the complexity of produced clauses and some optimisations have been developed

(chapter 5). However, it is not because the decompiler has been developed in the first place as an extension to the algorithm recognition framework [25] that it is limited to this scope. Our decompiler may have numerous other uses thanks to the popularity of the CLP formalism for the specification and verification of program properties. Automated verification of properties of imperative programs with the help of Horn clauses is studied by many researchers [11] and we hope this tool will be exploited in a wider context than the algorithm recognition framework [7].

1.2 State of the art

1.2.1 From imperative languages to declarative representations

When it comes to program analysis it turns out that logic programming is predominant in this field [7]. Since analysis specifications are generally written in a declarative style, as a system of constraints [3] [13] [21] or as a type inference rules [12] [1], in order to develop tools to verify specifications or termination of imperatives programs, researchers often rely on previous works based on Horn clauses [11] [20] [7] [10]. As a result, transformations from an imperative representation to a declarative one is often the first step of the process. In this section we review some of those transformations that have been made in different contexts but still similar to our work. However, a major difference between works presented here and the concrete implementation of our decompiler is that translations to declarative clauses are always made by the means of an intermediate representation. To the best of our knowledge, our decompiler is the first to propose a direct mapping between each single instruction of the Java Bytecode and a declarative clause (chapter 3).

An example of transformation from an intermediate representation of Java Bytecode to a declarative representation is the front-end of DIMPLE [7]. In short, DIMPLE is a fully-featured declarative analysis framework for Java. In order to provide a total round-trip solution to analysis design and evaluation DIMPLE features a system for encoding an intermediate representation of Java Bytecode as a database of facts [7]. This is the part of the work which interests us. The front-end translates from Java bytecode to the DIMPLE-IR : a set of Prolog relations that fully describe the input application and library classes. It is implemented as a whole-program transformation that extends the Scoot [24] compiler framework. As mentioned above, it is not a direct translation, the translation is made in two steps. First, Scoot converts from stack-based bytecode to a type three-address representation (Jimple) and generates a conservative method call graph. Then the Jimple's abstract syntax is transformed to the concrete syntax of a database of DIMPLE IR relations. In the DIMPLE IR Java classes are represented in terms of subtyping relationships, set of method declarations and sets of field declarations. Methods are represented as a set of statements and all inter-procedural control flow is explicit and is modeled by a control-flow graph for each method. See [7] for further information about DIMPLE.

DIMPLE is just one of the numerous analysis frameworks that rely on a declarative representation. Another approach, closer to the declarative representation used in our compiler, is the one used in the front-end of VeriMAP [11]. Indeed this approach based on transformations of Horn clauses with constraints - constrained Horn clauses (CHC). Although the proposed architecture is parametric with respect to the programming language, the VeriMap system implements this approach for C programs. The translation from C to CHC is based on the C Intermediate Language (CIL) infrastructure [11]. Again, the translation from the source language to the CHC encoding is not direct and goes through intermediate representations. The C Intermediate Language, that provide a set of tools that ease source to source transformations of C programs is primarily a high level intermediate representation. It aims to break down complicated constructs of C into simpler ones [19].

1.2.2 Rundroid

A much more related work that should be discussed is Rundroid, the initial front-end developed for the framework of algorithm recognition in binary code presented in [25]. The Rundroid front-end has been developed in order to work with the Dalvik Virtual Machine (DVM) bytecode and thus works with Android compiled programs. More specifically the role of this decompiler is to translate Dalvik binary code into Horn clauses. Indeed, as mentioned above, the framework uses Horn clauses as a universal representation to model algorithms as well as compiled code [25].

Android programs are written in Java before they get compiled to Google's Dalvik Virtual Machine (DVM) bytecode format. Contrary to the Java Virtual Machine bytecode format discussed in section 2.1 we only give a brief description of the DVM, see [25] for a more detailed presentation. The main difference between the Java Virtual Machine and the Dalvik Virtual Machine is that the JVM is stack-based while the Dalvik virtual Machine is register-based. In a stack-based Virtual Machine (VM), operands are stored in a stack date structure and operations are carried out by popping or pushing data from the stack (section 2.1). In a register-based VM operands are stored into registers hence instructions need to contain addresses of operand registers. There are no more push or pop operations to be performed. A particularity of the DVM registers is that they are statically typed, hence the Dalvik bytecode is a strongly typed assembly language. However, in our work only integers are considered as value of basic types [25].

The Dalvik Virtual Machine runs a bytecode program by keeping a stack of activation frames. Each method call creates a new frame that exists during the entire method call and dies upon return. Each frame also contains its own registers and a method can only operate in the scope of its frame. The number of registers contained in a frame depends on the static definition of the method, thus the number of registers used by a method is statically known [25]. Objects are contained in the memory of the system, connected through pointers.

The goal of this section is to give an informal intuition about the representation in constrained Horn clauses (CHC) that is largely inspired by the representation used for our decompiler described in chapter 3, see [25] for a complete formal presentation of the DVM to CHC translation. As for the Java bytecode, many Dalvik bytecode instructions are similar and only differ in the type or size of their operands thus we work with generalized (typeless) instructions which exemplifies the translation process.

Fig 1.1 describes a few simple representatives rules used to decompile DVM instructions into clauses. Dalvik bytecode instructions work over frames and their execution affects the registers in the frame or the memory. Rules for compiling the instructions have the form $q : ins \rightarrow E$ where q denotes a program point and E is the set of clauses resulting from the compilation of instruction ins occurring at q. A predicate symbol p_q is assigned to each program point q of the program P. In atoms of the form $p_q(\tilde{V}, M, M')$, $\tilde{V}$ represents a sequence of r variables that represents the registers just before executing the instruction ins at q. M and M' denote the memory before and upon termination of the method where p_q occurs. Th reader should also note that $\tilde{V} = V_0, ..., V_{r-1}$ where $V_0, ..., V_{r-1}$ represents individual values of the frame's registers where r is the number of registers used by the method where ins occurs. In each rule $\tilde{V'}$, denotes the states (values) of the registers after executing ins. id denotes the sequence $(V_0 = V_0, ..., V_{r-1} = V_{r-1})$ and id_{-i} (where i $\in$ [0, r-1]) the sequence $(V_0 = V_0, ..., Vi - 1 = V_{i-1}, V_{i+1} = V_{i+1}, ..., V_{r-1} = V_{r-1})$. Modifications between $\tilde{V}$ and $\tilde{V'}$ are visible through constraints of the rule. With respect to that, the reader should have a good intuition about the the mechanisms of our clauses. For instance, const d, c writes constant c into register d, so in Fig 1.1 the output register variable V_d is set to c while the other register variables remain unchanged (modeled with id_{-d}).

Theses previous works have been useful in the development of our decompiler. Although the Rundroid implementation is limited to basic instructions and does not deal with all the features proposed by our decompiler, the Rundroid tool was a great example of what needed to be done with Java bytecode. A comparison of pertinent implemented features is shown in Tab 1.1 for information. It is also profitable that [25] gave a first formal definition of the universal representation that has been put in place. Even if this representation had to be modified on many points (chapter 3), the general structure of clauses remains unchanged.

Features	Rundroid	Our compiler
Basic instructions	✓	✓
Method calls	✗	✓
Memory related instructions	✗	✓
Exceptions handling	✗	✗
Thread related instructions	✗	✗

Table 1.1: Comparison of implemented features

$$q : const\ d, c \rightarrow \{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = c\} \cup id_{-d},\ p_{q+1}(\tilde{V}', M, M')\}$$
$$q : add\ d, s, c \rightarrow \{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = V_s + c\} \cup id_{-d},\ p_{q+1}(\tilde{V}', M, M')\}$$
$$q : goto\ q' \rightarrow \{p_q(\tilde{V}, M, M') \leftarrow id,\ p_{q'}(\tilde{V}', M, M')\}$$

Figure 1.1: Compilation of some simple DVM bytecode instructions

Chapter 2

Overview of used formalisms

This chapter describes different formalisms discussed in the following sections. The purpose of this chapter is to give the reader a sufficient amount of knowledge about the Java Virtual Machine and the CLP paradigm to fully understand the different parts of the the document. It is obvious that all subtleties of theses technologies cannot be covered in this single chapter and that's not the aim of the chapter. A remainder on the basic operative of a LIFO stack is also explained in order to understand Java bytecode execution. We assume that the reader has basic notions from object-oriented and logic programming.

2.1 Java Virtual Machine and bytecode execution

The Java programming language is a general-purpose, concurrent and object-oriented language. Initially developed to address problems of building software for networked consumer devices, the Java platform proposes compiled code that is made to survive transport across networks. The main characteristic of the Java platform is that it is initially designed to support multiple host architectures [16]. Although the Java Programming language is a very interesting topic, it is not relevant for our needs. We will rather focus on the Java Virtual Machine and bytecode execution. The Java Virtual Machine (JVM) is an abstract computing machine that forms the cornerstone of the Java platform. As a virtual machine, it has an instruction set and manipulates various memory areas at runtime. The main purpose of a virtual machine is to provide hardware and operating system independence [16]. The JVM knows nothing about the Java programming language, the virtual machine manipulates binary files that respect the well structured *class* file format [16].

Inside the Java Virtual Machine

A JVM is a stack-based machine. Each Java virtual Machine thread has a private *Java Virtual Machine Stack*, created at the same time as the thread. A Java Virtual Machine stack stores frames (Fig 2.1).The frame holds local variables and partial results, and plays a part in method invocation and return. An important detail for our Horn clauses representation is that the Java Virtual Machine stacks can be of a fixed size. Each method's frame is composed of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method. The size of the local variable array and the operand stack are determined at compile-time and are supplied along with the code for the method associated with the frame. Each time a method is invoked a frame is created in the method's thread environment. The JVM deals with classical primitives types such as *byte, short, int, long, float, etc.* Reference types are also used to reference class instances, interfaces or arrays [16] [5]. Within the context of our work we are only concerned by numerical and references types.

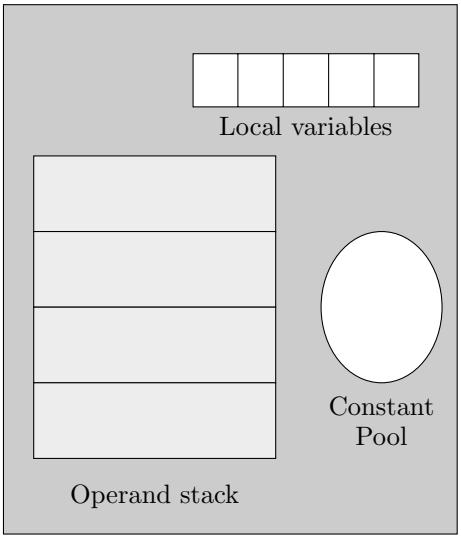

Figure 2.1: A JVM stack frame

As the name suggests, the constant pool contains constant definitions. For each class, the *class file* of the frame method's class contains a representation of the constant pool table. The constant definitions are composable, meaning the constant might be composed from other constants referenced from the same table. It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run-time. The constant pool also contains references to methods to be invoked and variables to be accessed via symbolic references. These symbolic method references are translated into concrete method references via the dynamic linking process. This process consists in loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the run-time location of those variables (section 4.2) [16] [5].

Parameters of the method are stored in the *local variable table* which is basically an array of local variables that hold the values of the local variables and the parameters. The number of local variables plays an obvious role in the size of the stack frame. The length of this array is also determined at compile-time and supplied in the binary representation of a class or interface. Each element of the array typically stores a value of a primitive type except for primitive types *long* and *double* that need a pair of consecutive local variables to hold their values. Local variables are addressed by indexing and values of type *long* or *double* may only be addressed using the lesser index. The JVM uses the local variable array to pass parameters on method invocation. Parameters are passed in the n first consecutive elements starting at index 1 and 0 is always used to pass the a reference to the object on which the instance method is being invoked (*this*) [16] [5].

The Last part of the JVM stack frame to be discussed is the operand stack. As a reminder, a stack is a very simple idea. A stack is a data structure that has two fundamental operations, push and pop. The push operation is used to store something on the top of the stack, and the pop operation retrieves something from the top of the stack. The operand stack is a *last in first out* (LIFO) stack used to push and pop values. Certain instructions load values from local variables or fields onto the top of the operand stack, others take values from the top of the operand stack, perform an operation on them and push the result back onto the operand stack in order to respect the LIFO principle. The maximum depth of the operand stack of a frame is determined at compile time and is supplied along with the bytecode of the method. Knowing the maximum depth of the operand stack will be very useful to make our constrained horn clauses translations (chapter 3). The operand stack also plays a role when receiving method results and preparing parameters to be passed in the local variable table of the called method [16] [5].

The Java Virtual Machine also contains a heap that is shared between all JVM threads. The heap is the run-time data area from which memory for all class instances and arrays is allocated [16] [5]. It is reclaimed by an automatic storage management system that will not be discussed here as it is not modeled by our Horn clauses representation.

The Java bytecode and its execution

The Java bytecode is the intermediate representation of the Java programming language. It must be structured with respect to the *class* file format in order to be executed inside the JVM. The model of computation of Java bytecode is that of a stack-oriented programming language. As the name implies, Java bytecode consists of one-byte instructions. An instruction is followed by zero or more operands [16]. Java bytecode is strongly typed, instructions are composed from a type prefix and the operation name [5]. The fact that operations are replicated in multiple instructions depending on the their types implies the existence of a great number of bytecode instructions. Instructions can be classified into several groups depending on their nature [5] :

- control flow instructions
- Stack manipulation instructions
- Arithmetics and type conversion
- Object manipulation and method invocation
- Specialized thread related and exception throwing instructions

However, as mentioned in section 1.1.2 our decompiler is more a proof a concept than a full-featured application. Thus, not all the Java bytecode instructions are taken in account. Globally we focus on the whole Java bytecode instruction set, except for the specialized thread related, exception throwing instructions and bitwise operators. Also we only consider number manipulations and put apart the char primitive type. As a consequence we cannot treat Java objects like String other objects that make use of the char primitive type. In the case our decompiler is confronted with an instruction that is not taken in account, we simply signal that a non-implemented instruction is part of the program and signal its position in the CLP output. Also, calls to external libraries not provided in the input jar file are translated into a predicate call that contains the explicit name of the external call and that always returns true. Thus it is possible to keep a trace of external calls into the CLP representation. This is useful when comparing two different programs, in order to be able to know if they use the same libraries. In order to be complete we now provide a list of the different bytecode instructions that are not handled by our decompiler :

- Exceptions related instructions
  {athrow}
- Char related instructions
  {caload, castore,i2c}
- Bitwise operators
  {iand,ior,ishl,ishr,iushr,ixor,land,lor,lshl,lshr,lushr,lxor}
- Others
  { breakpoint,dup2,imdep,instanceof,invokedynamic,
  jsr,monitorenter,monitorexit,ret,tableswitch,wide}

The translation and the full description of rest of the bytecode instruction set is discussed in chapters 3 and 4.

To understand the Java bytecode computation model we start with a very basic example. First, let us consider a trivial bytecode fragment that simply computes the sum of 1 and 3 :

```
iconst_1
iconst_3
iadd
```

The i prefix at the beginning of each instruction indicates that the instruction manipulates an integer. Bytecode is evaluated sequentially, in the instructions' arrival order. The first two instructions push constants 1 and 3 on the operand stack, respectively. The third instruction computes the sum of the two values by popping these 2 values and pushing the result of the operation to the stack (Fig 2.2). One should note that the stack grow upward.

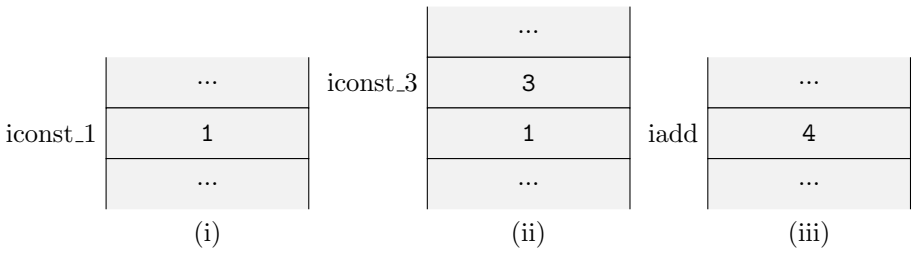

Figure 2.2: Step by step execution of a bytecode fragment

2.2 The CLP paradigm

In order to understand the representation provided by our decompiler it is required to introduce the key features of the Constraint Logic Programming (CLP) paradigm. This section defines the basic notions of Constraint Logic Programming that will be used. As a reminder, the CLP paradigm is used to represent the Java bytecode instructions translated by our decompiler.

A logic program consists of a set of Horn clauses and can be used to express a query on relational data bases [6]. A Horn clause is simply a disjunction of literals with at most one unnegated literal, they are the basis of logic programming. CLP represents a successful attempt to merge the best features of logic programming and constraint solving. This combination helps make CLP programs both expressive and flexible [15]. A constraint is just a restriction imposed over the combination of values of some variables of the program. Solving a problem with constraints means finding a way to assign values to all its variables such that all constraints are satisfied [14]. As it is a declarative formalism the programmer only has to define the constraints, then some constraint solvers are used to automatically find a good variable assignment.

A CLP program is a finite set of clauses or rules of the form $A \leftarrow c, B$ where A is an atom, c is a constraint and B is a goal [10]. A CLP clause is just like a Logic Programming (LP) clause, except that its body may contain also constraints of the considered sort [14]. The semantics of a CLP program is defined as its least model [10]. Rules may be recursive and there may be multiples rules in the definition of a predicate [17]. Constraints are added to logic programming by considering a specific constraint sort (e.g linear equations over the real numbers). Moving from LP to CLP, the concept of unification is generalized to constraint solving : the relationship between a goal and a clause can be described not only via term equation but via more general statements, i.e. constraints. This allows for a more general and flexible way to control the flow of the computation [14].
An example of a CLP clause could be [14]:

$$p(X, Y) \leftarrow \{X < Y + 1\},\ q(X),\ r(X, Y, Z).$$

This clause states that $P(X, Y)$ is true if $q(X)$ and $r(X, Y, Z)$ are true, and if the value of x is smaller than that of $y + 1$. From an operational point of view, in a LP resolution step, we have to check the existence of a most general unifier between the selected sub-goal and the head of a clause. In CLP, we also have to check the consistency of the current set of constraints with the constraint in the body of the clause [14]. In CLP computation we accumulate substitutions and constraints during a computation. On a syntactic level, note that we write the constraints of a clause between { and } in order to distinguish them from the goal of the body.

Chapter 3

From bytecode instructions to clauses

This chapter aims at describing the decompilation process that has been put in place and some implementation details of our decompiler. We also introduce the way basics Java bytecode instructions are translated into CLP clauses.

3.1 The decompilation process

We now describe the work-flow of our decompiler.

Our decompiler produces a Horn clause representation of a given jar file. A jar file is a package file format used to aggregate many Java class files and associated meta-data and resources. In the context of our work we are only interested in the set of class files contained in the jar. A class file contains the definition of a single class or interface and consists of a stream of 8-bit bytes [16]. Although the class files follows a well defined structure, we have chosen to work with the mnemonic representation rather than the binary (hexadecimal) one. Providing our own implementation of a lexer for the hexadecimal representation was considered unproductive. Indeed, some tools provide a convenient way to read Java class files. For the purpose of this project we used the Byte Code Engineering Library (BCEL) reference in order to switch from the hexadecimal representation of the class files into a mnemonic one (Figure 3.1). This translation is the first step of our decompilation process. We apply this transformation to every class file contained in the give jar file in order to have a collection of convenient class file representations. The mnemonic representation of the Java bytecode is the one briefly illustrated in section 2.1.

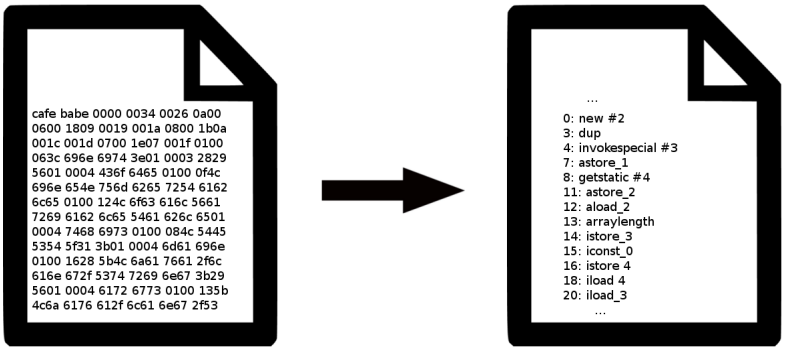

Figure 3.1: Switching from hexadecimal to mnemonic representation (BCEL)

The next step of our decompilation process is to handle the mnemonic representation of the class files produced by BCEL. In order to be able to perform a sequential translation of each bytecode instruction for the program we have to store some information about their context. We construct a graph that contains all the information needed. The graph contains information about the structure described by the class files and also computed information. This graph should not be confused with a classical abstract syntax tree or parse tree that are found in traditional compilers. As we work with machine code, each instruction performs a very specific task and is executed sequentially. Thus the syntactic structure is rather simple and does not justify the construction of a tree to represent theses constructs. However, a graph with some information about methods and class fields still useful to simplify the translation phase.

The graph is composed of a node for each class of the jar file. Attributes of the classes are described in each class node and heritage relations are also encoded in the graph. For each method of the class, the produced graph also contains positions and offsets of bytecode instructions that compose the method. Local variables positions and offsets as well as the signature of the method are also part of the graph. For each method, the maximum needed stack size is also added to the graph. A representation of the constant pool described in section 2.1 is necessary to fully translate the mnemonic representation of bytecode instructions.

Once all this information is retrieved and structured into the graph the translation phase takes place. As mentioned in the previous sections, this translation phase is sequential. The decompiler, with the help of a sliding windows goes through each method of each class file and translates the bytecode instructions one by one relatively to their static context. In the case of the default decompilation process, the size of the sliding window is set to 1. In that way the decompiler simply goes through the instructions one by one and translates each of them into a single CLP predicate. The fact that the size of the sliding window is configurable is justified in chapter 5 where some optimisations are discussed.

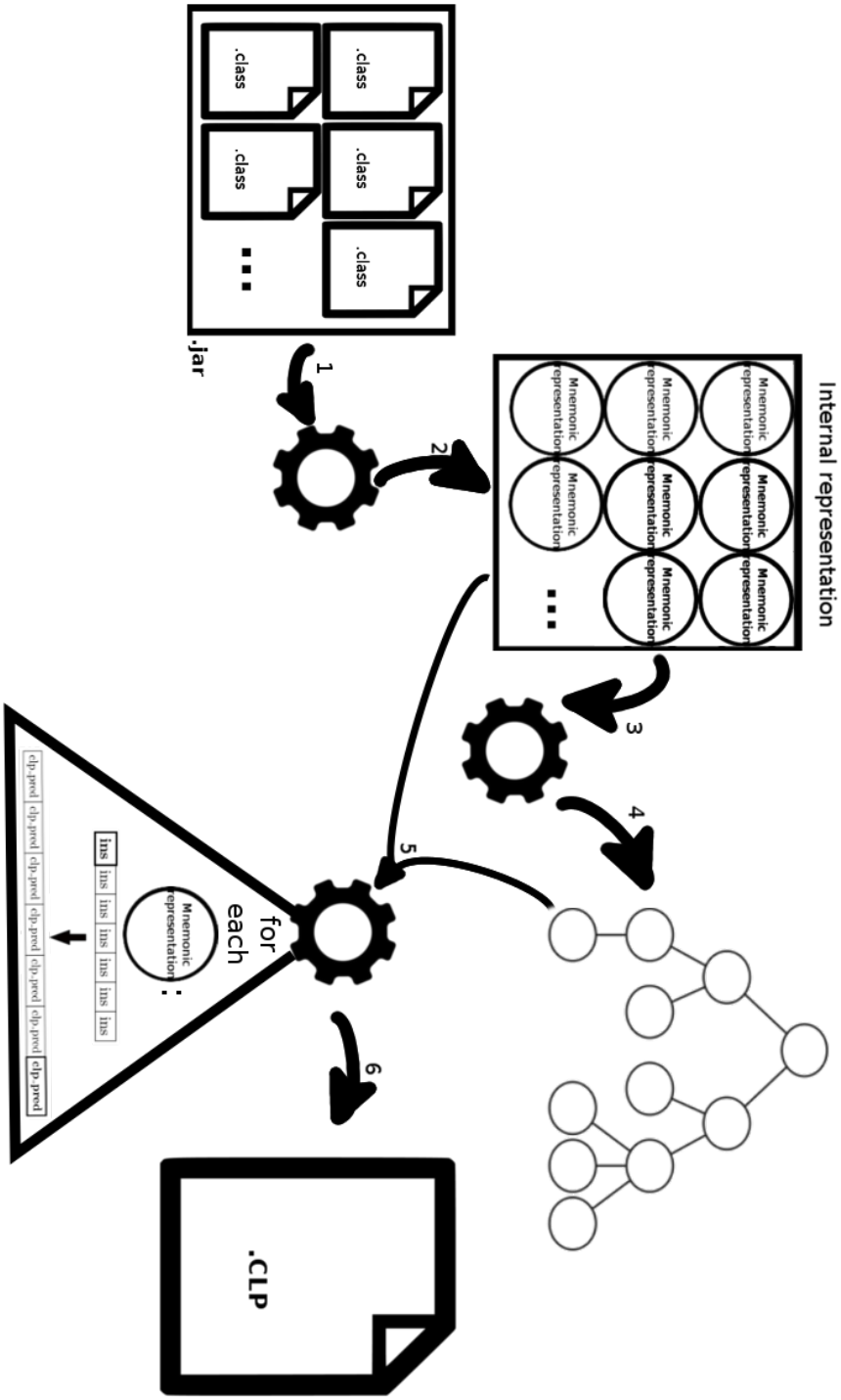

Figure 3.2: The decompilation process

3.2 Ensuring the correctness of the decompilation

In order to ensure the correctness of our decompilation process we put in place a testing module. The principle behind our test process is pretty simple. Given a batch of compiled Java programs and expected results (verified by an execution of the jar) associated with each of those programs, the automated test module decompiles the Java programs one by one and executes their decompiled (clp) versions. Then executions results are compared with their expected results and we get detailed information of each execution flow. The test process is illustrated in Figure 3.3.

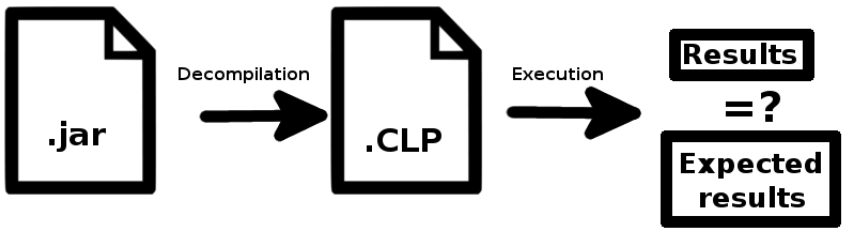

Figure 3.3: Illustration of the test process

Currently our testing pool is composed of about fifty Java programs and their associated expected results and covers the whole set of bytecode instructions presently handled by our decompiler. This allows us to ensure that future modifications or extensions of the decompiler still produce consistent outputs.

3.3 Basic instructions

We now describe the rules that are applied for the translation of each type of instruction. As our Horn clause representation is a level of abstraction above bytecode representation we can generalize some instructions. For instance we can group descriptions of bytecode instructions that compute the same operations for different types. Indeed, although this differentiation is relevant inside the Java Virtual Machine, our representation does not make any distinction between numeric types. In order to develop our decompiler we grouped similar bytecode instructions into meta-instructions. Each of these groups corresponds to a particular type of clause in our decompiler.

We first describe the decompilation rules that apply for the set of instructions related to stack manipulations, arithmetic operations and control flow. Those instructions are considered simple as they do not directly involve object manipulations or method calls. Each instruction is located at a program point that can be seen as the instruction's position in the file. The execution of an instruction at program point q is modelled by a predicate p_q. Rules have the form $q : ins \rightarrow E$ where E is the set of clauses resulting from the compilation of instruction ins occurring at the program point q. We generate clauses with constraints on integers. As mentioned in section 2.1, the Java Virtual Machine is stack-based . The JVM stores a stack frame for each method invocation. Most of the bytecode instructions directly work with the operand stack and the local variables table of its method's stack frame. Hence, the operand stacks and local variables of the corresponding method are represented in each instruction's clause. We represent the local variables table and the operand stack as a set of constrained variables ($\tilde{IV}$) present as arguments in the head of the clause. We let $\tilde{IV} = IV_0, ..., IV_{n+m-1}$ and $\tilde{IV}' = IV'_0, ..., IV'_{n+m-1}$ be sequences of distinct variables. For each $i \in [0, n+m-1]$, variable V_i (resp. V'_i) denotes the value of variable i before (resp. afer) executing ins. The first n variables of the set represent the local variables table and the m following variables model the operand stack. The structure of $\tilde{IV}$ is represented in Figure 3.4.

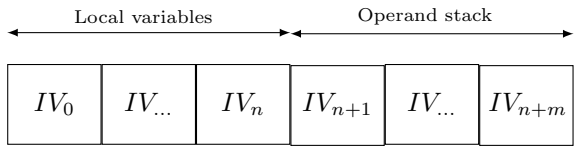

Figure 3.4: The $\tilde{IV}$ structure

In reality, the operand stack size continually changes during the execution steps, thus the value of m is based upon the maximum stack size. This number is statically known and retrieved from the class file. We let id denote the sequence $(V'_0 = V_0, ..., V'_{n+m-1} = V_{n+m-1})$ and id_{-i} (where $i \in [0, n+m-1]$) the sequence $(V'_0 = V_0, ..., V'_{i-1} = V_{i-1}, V'_{i+1} = V_{i+1}, ..., V'_{n+m-1} = V_{n+m-1})$. In order to simulate the JVM behaviour and navigate in our operand stack we also represent the stack pointer as an argument (SP). Each clause also contains five arguments H, HO, HS, HSO *and* R. H and HO represent the Heap before and after the execution of instruction p_q respectively. HS and HSO represent the heap size before and after the execution of p_q respectively. R represent the return value of the instruction's method. Those representations are detailed in sections 4.1 and 4.2. With respect to that we can determine the arity of each predicate. Predicate of clauses that represent the instructions of a single method have the same arity. The arity of p_q is $s + 6$ where the s is size of $\tilde{IV}$ and is only determined by the method from which the instruction was taken.

In an atom of the form

$$p_q(\tilde{IV}, SP, H, HO, HS, HSO, R)$$

the parameters correspond to the state of the system just before executing the instruction at q except for HO (heap out) and HSO (heap size out) that represent the state of the Heap after the execution of p_q.

In order to simulate the stack-based mechanism of bytecode instructions we introduced two predicates read and write. Those predicates are defined as follows :

$$read(I, \tilde{IV}, V_i) \leftarrow \{I = i\}.$$
$$write(I, \tilde{IV}, \tilde{IV'}, V) \leftarrow \{I = i,\ V'_i = V\} \cup id_{-i}.$$
$$\forall i \in [0, n + m - 1]$$

The read predicate is used in its mode $read(IN, IN, OUT)$. For a given index number (e.g. I) and a set of variables (e.g. $\tilde{IV}$), the predicate returns the value of the variable of the set indexed by the constrained variable ($\tilde{V_I}$).

The write predicate is used in its mode $write(IN, IN, OUT, IN))$. for a given index number (e.g I), a set of variables (e.g. $\tilde{IV}$) and a constrained variable V, the predicate returns the set $\tilde{IV'}$ where $\tilde{IV'} = (V'_0 = V_0, ..., V'_{I-1} = V_{I-1}, V'_I = V, V'_{I+1} = V_{I+1}, ..., V'_{n+m-1} = V_{n+m-1})$.

Those read and write predicates could seem excessive. However, having such mechanisms to read and write $\tilde{IV}$ variables is necessary. The fact that a clause can be evaluated multiple times during a program execution implies a dynamic mechanism to read and write elements on the stack to handle the fact that the actual stack size is not necessary the same for two evaluations of a predicate. As an example, a loop mechanism is represented by a branch to an instruction defined higher-up in the sequence, hence the instructions located between the branched instructions and the branching instructions will be evaluated multiple times (body of the loop). However, nothing garanties that the stack size does not change during this time. By dynamic mechanism we refer to the fact that the index variable (e.g. I) is not statically established in the clause. The reader should note that the top of the current stack is represented by V_{SP}. As a result, a push operation of the value contained by the variable V on the operand stack is represented by the predicate $write(SP, \tilde{IV}, \tilde{IV'}, V)$ associated with an incrementation of the stack pointer, where $\tilde{IV}$ contains the operand stack with the pushed value.

We now describe the idea behind each of our representations for the different groups of bytecode instructions. In this chapter, each bytecode instruction is identified by a program point q and is translated in a predicate p_q.

3.3.1 Stack manipulation instructions

This section describes representations used to model the Java bytecode instructions that are dedicated to retrieve and store values on the operand stack and the local variable table.

- CONST {lconst_n,iconst_n,dconst_n,fconst_n,ldc2_w, bipush}

  Push a specified constant value on the top of the operand stack.

  $q\ : CONST\ d \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ V = d,\ SP' = SP + 1\}\ ,$
  $write(SP', \tilde{IV}, \tilde{IV}', V),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$
  $\}$

  The $CONST$ meta-instruction defines the push of a constant d on the top of the stack. In order to represent this operation we constraint a variable V to be equal to the value d and write it to the top of the stack. The stack pointer is incremented by one as we added an element on the stack.

- LOAD {aload_n,aload,fload,fload_n,iload,iload_n,lload,lload_n}

  Push a value from a specified local variable on the top of the operand stack.

  $q\ : LOAD\ d \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ I = d,\ SP' = SP + 1\}\ ,$
  $read(I, \tilde{IV}, V),$
  $write(SP', \tilde{IV}, \tilde{IV}', V),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$
  $\}$

  The $LOAD$ meta-instruction groups all bytecode instructions that retrieve an element from the local variables and push it on the top of the operand stack. In order to represent this operation a variable I is constrained to the value d which represents the index of the local variable to retrieve in the local variable table. The read predicate is used to read the value at index d and places it in the variable V. The retrieved value is pushed onto the top of the operand stack via the write predicate. The stack pointer is incremented by one as we added an element on the stack.

- STORE {astore_n,astore,dstore_n,dstore,fstore_n,fstore,istore_n,istore,lstore_n,lstore}

  Store the value from the top of the operand stack into a specified local variable and pop the value.

  $q\ : STORE\ d \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ I = d,\ SP' = SP - 1\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $write(I, \tilde{IV}, \tilde{IV'}, V),$
  $p_{q+1}(\tilde{IV'}, SP', H, HO, HS, HSO, R).$
  $\}$

  The $STORE$ meta-instruction defines an operation that retrieves the first element from the operand stack and stores it in a given slot of the local variables table. In order to represent this operation an index variable I is constrained to be equal to d. The read predicate is used with SP as the first argument in order to retrieve the last value from the operand stack, variable V is constrained to be equal to this value.the write predicate is used to store this value in the variable V_d. The stack pointer is decremented for the next instruction as we popped an element from the stack.

- DUP {dup}

  Duplicate value at the top of the stack in accordance with a specified scheme.

  $q\ : DUP\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP + 1\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $write(SP', \tilde{IV}, \tilde{IV'}, V),$
  $p_{q+1}(\tilde{IV'}, SP', H, HO, HS, HSO, R).$
  $\}$

  The DUP meta-instruction is used to duplicate the first element of the operand stack (i.e. element at the top of the stack). To perform this operation the *read* predicate is simply used to retrieve the top value of the operand stack (with SP as the index). The stack pointer is incremented and the value retrieved by the *read* is pushed on the operand stack.

Now, lets us consider the following Java program to illustrate some of those stack manipulation instructions :

```
public class DEMO {

public static void main(String[] args){
    int a = 10;
    int b = a;
    }
}
```

(a) Source code version

```
0:    bipush  10
2:    istore_1
3:    iload_1
4:    istore_2
5:    return
```

(b) Bytecode version (main)

Figure 3.5: A very simple Java program

Figure 3.5 shows that the initializations and assignment of variables is mainly performed by stack manipulation instructions. The first instruction of the *main* method is represented by the first two bytecode instructions 0 and 2 that push a constant (10) on the top of the operand stack and subsequently pop this value into the first variable of the local variable table. The second assignment is performed by the bytecode instructions 3 and 4 that push the value of the variable on the top of the operand stack and then pop this value into the second variable of the local variable table. The bytecode *return* statement is ignored for now.

The application of our decompilation rules on the Java bytecode sequence described by Figure 3.5b result in the CLP program defined in 3.1.

Even with this very simple example, we can already notice that the decompiled CLP version is much more verbose than the original source code. A first explanation for this gap is that the number of instructions introduced by the bytecode version is much higher than in the original source code. A lot of operations are necessary because of the stack-based mechanism of the JVM. Many of these operations are unnecessary, and could be bypassed in our decompiled declarative version. Chapter 5 aims to describe how our decompiler optimises and aggregates those representations.

The reader should note that Java variables a and b from the Java source code presented in Figure 3.5a correspond to CLP variables $V1$ and $V2$, respectively. The CLP variable $V0$ handles the reference towards the array of String passed as parameter of the method. The parameter passing mechanism will be described in section 4.2. The operand stack is entirely described by the $V3$ CLP variable as the maximum possible stack size for this method is one. The stack pointer SP is constrained at 2 in the first predicate of the method, which corresponds to the zero level of the operand stack. The stack pointer should always be initialized to the size of the local variable table in the first predicate of each method.

Listing 3.1: A very simple Java program (Decompiled)

```
%    0: bipush 10
p1_1_0(V0,V1,V2,V3,SP,H,HO,HS,HSO,R) :-
            {SP=2,V=10,SP0=SP+1},
            write(SP0,V0,V1,V2,V3,W0,W1,W2,W3,V),
            p1_1_1(W0,W1,W2,W3,H,HO,HS,HSO,R).
%    2: istore_1
p1_1_1(V0,V1,V2,V3,SP,H,HO,HS,HSO,R) :-
            {I=1,SP0=SP-1},
            read(SP,V0,V1,V2,V3,V),
            write_1(I,V0,V1,V2,V3,W0,W1,W2,W3,V),
            p1_1_2(W0,W1,W2,W3,SP0,H,HO,HS,HSO,R).
%    3: iload_1
p1_1_2(V0,V1,V2,V3,SP,H,HO,HS,HSO,R) :-
            {I=1,SP0=SP+1},
            read(I,V0,V1,V2,V3,V),
            write(SP0,V0,V1,V2,V3,W0,W1,W2,W3,V),
            p1_1_3(W0,W1,W2,W3,SP0,H,HO,HS,HSO,R).
%    4: istore_2
p1_1_3(V0,V1,V2,V3,SP,H,HO,HS,HSO,R) :-
            {I=1,SP0=SP-1},
            read(SP,V0,V1,V2,V3,V),
            write_1(I,V0,V1,V2,V3,W0,W1,W2,W3,V),
            p1_1_4(W0,W1,W2,W3,SP0,H,HO,HS,HSO,R).
%    5: return
p1_1_4(V0,V1,V2,V3,SP0,H,HO,HS,HSO,R) :-
                    {HS = HSO,  unify(H,HO)}.
```

3.3.2 Arithmetic operations

This section describes representations used to model from bytecode's arithmetic operations as addition, subtraction, division, modulo and multiplication. Incrementation and negation instructions that are derived from theses arithmetic operations are also described.

- ADD{dadd, fadd, iadd, ladd}

  Add the two top stack values, pop them and push the result.

  $q\ : ADD\ \rightarrow \{\ p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, VR = V1 + V2\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $write(SP', \tilde{IV}, \tilde{IV}', VR),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$

  $\}$

We illustrate the representation of the ADD meta-instruction. The ADD meta-instruction groups the definition of the $(d|f|i|l)add$ bytecode instructions. The ADD models popping two values from the operand stack and the push of their sum. This is represented by two calls to the *read* predicate with SP and $SP'(= SP - 1)$ as indexes. The constrained variables $V1$ and $V2$ returned by the read predicate are used to constrain a third variable VR to be equal to the sum of $V1$ and $V2$. The *write* predicate is used to model the push of the sum (VR) on the top of the operand stack. The representation of the ADD meta-instruction is very similar to the other representation used to model the $SUB, DIV, MODULO$ and MUL meta-instructions, that is why they are not described here.

- IINC{iinc}

Increment a specified local variable by specified a constant.

$q\ : IINC\ i, n\ \rightarrow \{$
$p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
$\{\ I = i,\ VR = V + n\}\ ,$
$read(I, \tilde{IV}, V),$
$write(I, \tilde{IV}, \tilde{IV}', VR),$
$p_{q+1}(\tilde{IV}', SP, H, HO, HS, HSO, R).$

$\}$

The $IINC$ meta-instruction is special case of addition that is defined for optimisation purposes inside the JVM. As the incrementation of a variable is frequently used in Java programs the JVM is designed to deal more efficiently with this type of operation. Without this optimisation, incrementing a variable would involve loading a constant ($CONST$) and a specified variable ($LOAD$) followed by an addition (ADD).

The two arguments i and n represents the index of the local variable to increment and the integer to add to this variable. The *read* predicate is used to read the value of the local variable to increment (V_i) and constraint the variable V to be equal to this value. A new variable VR is introduced in order to be constrained to be equal to the sum of V and n. The *write* predicate is used to store the new value of V_i' into the local variable table.

- NEG{dneg,fneg,ineg, lneg}

  Pop the value at the top of the stack and push the negated value.

  $q : NEG \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{ VR = -V \}$,
  $read(SP, \tilde{IV}, V)$,
  $write(SP, \tilde{IV}, \tilde{IV}', VR)$,
  $p_{q+1}(\tilde{IV}', SP, H, HO, HS, HSO, R)$.

  $\}$

  The NEG meta-instruction is used to negate a given value at the top of the operand stack. The description of this representation is pretty simple to understand. The *read* predicate pops the value located at the top of the operand stack and constrains the variable V to be equal to this value. A new variable VR is introduced and constrained to be equal to $-V$. The *write* predicate pushes the negated value (VR) on the top of the operand stack.

We can now extend our previous example and deal with some arithmetic operations.

```
public class DEMO {

public static void main(String[] args){
    int a = 10;
    int b = a + 5;
    a++;
    }
}
```

(a) Source code version

```
0:   bipush  10
2:   istore_1
3:   iload_1
4:   iconst_5
5:   iadd
6:   istore_2
7:   iinc    1, 1
10:  return
```

(b) Bytecode version (main)

Figure 3.6: A simple Java program

The first instruction is the same as in our previous example. We replaced the second instruction of the Java program by an addition and added a third incrementation operation. The (**int** b = a + 5;) Java statement is compiled into bytecode instructions 3,4,5 and 6. The value of the variable a is first pushed onto the operand stack (*iload_1*), then the constant 5 (*iconst_5*) is pushed onto the stack. Then, the *iadd* performs the addition of those two values and the result is stored in the local variable table via *istore_2*. The (a++;) statement is directly translated as an optimised *iinc* bytecode instruction.

The application of our decompilation rules over the Java bytecode sequence described by the Figure 3.6b result in the following CLP program :

Listing 3.2: A simple Java program (Decompiled)

```
%    0: bipush 10
p1_1_0(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {SP=2,V=10,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,W0,W1,W2,W3,V),
          p1_1_1(W0,W1,W2,W3,W4,H,HO,HS,HSO,R).
%    2: istore_1
p1_1_1(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,SP0=SP-1},
          read(SP,V0,V1,V2,V3,V4,V),
          write_1(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_2(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSOS,R).
%    3: iload_1
p1_1_2(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,SP0=SP+1},
          read(I,V0,V1,V2,V3,V4,V),
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_3(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    4: iconst_5
p1_1_3(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {V=5,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_4(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    5: iadd
p1_1_4(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {SP0=SP-1,VR=V1+V2},
          read(SP,V0,V1,V2,V3,V4,V1),
          read(SP0,V0,V1,V2,V3,V4,V2),
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,VR),
          p1_1_5(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    6: istore_2
p1_1_5(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=2,SP0= SP-1},
          read(SP,V0,V1,V2,V3,V4,V),
          write(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_6(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    7: iinc 1 1
p1_1_6(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,VR=V+1},
          read(I,V0,V1,V2,V3,V4,V),
          write(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,VR),
          p1_1_7(W0,W1,W2,W3,W4,SP,H,HO,HS,HSO,R).
%   10: return
p1_1_7(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
                    {HS = HSO,  unify(H,HO)}.
```

3.3.3 control flow instructions

We now describe the representation of the bytecode instructions that impact the flow of the program. Those instructions are mainly simple branches and conditional branches.

- COMPARE {dcmpl, fcmpl, lcmpl}

  Compare the to top values on the top of the stack, pop them and push 1,0 or -1 depending on the comparison's result.

  $q\ : COMPARE\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V1 > V2\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $write(SP', \tilde{IV}, \tilde{IV}', 1),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V1 = V2\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $write(SP', \tilde{IV}, \tilde{IV}', 0),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V1 < V2\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $write(SP', \tilde{IV}, \tilde{IV}', -1),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$
  $\}$

  We illustrate the representation of the $COMPARE$ statement for $(d|f|l)cmpl$ bytecode instructions. The $(d|f|l)cmpl$ (compare less) instructions pop and compare the top two values of the operand stack and push a value depending of the comparison result. The comparison operates over two values V_1 and V_2 where V_1 is the first value to be popped from the sack and V_2 the second. The result pushed onto the stack is either 1,0 or -1 depending if $(V_1 > V_2)$, $(V_1 = V_2)$ or $(V_1 < V_2)$ respectively. The representation is similar for $(d|f|l)cmplg$ (compare greater).

  The representation is divided into three predicates, one for each possible case of the comparison. As those three cases are mutually exclusive, only one of the three predicates is evaluated for a given state of the operand stack. Predicates differ from each other only in the constraint for the comparison of V_1 and V_2 part and the associated *write* value. Each predicate consists of two *read* invocations that retrieve the first two values, and a *write* instruction that push the comparison's result on the top of the stack. The stack pointer is decremented by one for this instruction as the operation pop two values but pushes also a value on the operand stack.

- GOTO{goto}

  Branch to the specified instruction.

  $q\ : GOTO\ x\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ \}\ ,$
  $p_x(\tilde{IV}, SP, H, HO, HS, HSO, R).$
  $\}$

  The representation of the *GOTO* instruction is pretty straightforward. There is no constraint for this representation. The evaluation of the predicate p_q branches directly to the evaluation of the predicate p_x

- IF_COMPARE {if_icmpge}

  Compare the top two values at the top of the stack, pop them and branch to the specified instruction or not depending on the comparison result.

  $q\ : IF\_COMPARE\ x\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1,\ SP'' = SP - 2, V2 < V1\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $p_{q+1}(\tilde{IV'}, SP'', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1,\ SP'' = SP - 2, V2 >= V1\}\ ,$
  $read(SP, \tilde{IV}, V1),$
  $read(SP', \tilde{IV}, V2),$
  $p_x(\tilde{IV'}, SP'', H, HO, HS, HSO, R).$

  $\}$

  We illustrate the representation of the $IF\_COMPARE$ meta-instruction for if_icmpge bytecode instructions. The if_icmpge (compare if greater or equals) instruction pops and compares the two top values of the operand stack and branches towards a specified instruction or not depending on the result. The representation is similar for the $if\_acmp(eq|ne)$ and $if\_icmpeq|ne|lt|le)$ bytecode instructions. The representation is divided into two mutually exclusive predicates. The constraint parts of the two predicates differ from each other only for the comparison of V_1 and V_2. The next predicate to be evaluated is either the next instruction's predicate (p_{q+1}) or the specified instruction's predicate (p_x) depending if ($V_2 < V_1$) or ($V_2 >= V_1$), respectively.

- IF {ifle}

  Branch to the specified instruction or not depending on the top stack value. Top stack value is popped in the same time.

  $q\ : IF\ x\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V > 0\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $p_{q+1}(\tilde{IV'}, SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V <= 0\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $p_x(\tilde{IV'}, SP', H, HO, HS, HSO, R).$
  $\}$

  The IF meta-instruction defines an operation that redirects the execution flow towards a specified instruction or not depending on the value at the top of the operand stack. This meta-instruction is often associated with the $COMPARE$ meta-instruction. We illustrate the representation of IF for the $ifle$ byecode instruction (if less or equal). The representation is similar for the $if(eq|ge|gt|lt|ne|nonnull|null)$ bytecode instructions.

  This representation is very similar to the $IF\_COMPARE$ representation. It is also divided into two mutually exclusive predicates. The difference is that only one value (V) is popped from the operand stack. The next predicate to be evaluated is either the next instruction's predicate (p_{q+1}) or the specified instruction's predicate (p_x) depending if ($V > 0$) or ($V <= 0$) respectively.

- LOOKUPSWITCH {lookupswitch}

  Branch to one of the specified instructions depending on the top stack value. Top stack value is popped in the same time.

  $q\ : LOOKUPSWITCH\ d\ (v,r)\ (w,s)\ (x,t)\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V = v\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $p_r(\tilde{IV}, SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V = w\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $p_s(\tilde{IV}, SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1, V = x\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $p_t(\tilde{IV}, SP', H, HO, HS, HSO, R).$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1\}\ ,$
  $p_d(\tilde{IV}, SP', H, HO, HS, HSO, R).$

  $\}$

  The last representation of the control flow operations defines the $LOOKUPSWITCH$ instruction. This representation is a bit more complex than the previous control flow instructions. This instruction is the bytecode level equivalent of the *switch* statement in Java. The principle is that the execution flow is redirected towards a specified instruction depending on the value of a specified variable. The instruction takes two arguments; The default instruction to branch to (d) and a list of pairs of the form ($value$, $instruction\_number$). If the specified variable matches one of the defined values, the execution flow is redirected towards the corresponding instruction. Otherwise, the execution flow is redirected towards the default instruction.

  We present here the representation of the $LOOKUPSWITCH$ meta-instruction in the case of 3 defined values but generalize easily for the case of n values program point pairs. The reader should note that the order in which Prolog tries to evaluate the clauses is important for this representation. Indeed, this representation assumes that the clauses are evaluated in the order they are written. With respect to that, the default case is only evaluated if no other case matches and we are free to not specify any constraint for the evaluated variable V.

We assume that the value of the variable that determines the execution flow is stored on the top of the stack. As this representation handles 3 possible cases for the value of the variable, it is represented by four predicates. One predicate for each possible value of the variable plus one for the default case. The first three predicates are almost identical. The *read* predicate is used to retrieve the actual value located at the top of the operand stack. They only differs in the constraint part when testing the value retrieved in the variable V. If the constraint $(V = v)$ is respected, the first clause is executed and the control flow goes towards predicate p_r with a decremented stack pointer. The second and thrid clauses are similar. For the fourth clause (default case), as there is no actual constraint on previously constrained variable the clause is always executed when evaluated.

Back to our example, we now illustrate some of the control flow instructions described in this section.

```
public class DEMO {

public static void main(String[] args){
    int a = 10;
    int b = 0;
    if (a > 5){
        b = a + 5;
    } else{
        b = a - 5;
        }
    }
}
```

(a) Source code version

```
0:   bipush  10
2:   istore_1
3:   iconst_0
4:   istore_2
5:   iload_1
6:   iconst_5
7:   if_icmple      17
10:  iload_1
11:  iconst_5
12:  iadd
13:  istore_2
14:  goto    21
17:  iload_1
18:  iconst_5
19:  isub
20:  istore_2
21:  return
```

(b) Bytecode version (main)

Figure 3.7: A Java program

The application of our decompilation rules over the Java bytecode sequence described by the Figure 3.6b results in the following CLP program :

Listing 3.3: A Java program (Decompiled)

```
%    0: bipush 10
p1_1_0(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {SP=2,V=10,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,W0,W1,W2,W3,V),
          p1_1_1(W0,W1,W2,W3,W4,H,HO,HS,HSO,R).
%    2: istore_1
p1_1_1(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,SP0=SP-1},
          read(SP,V0,V1,V2,V3,V4,V),
          write_1(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_2(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    3: iconst_0
p1_1_2(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {V=0,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_3(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    4: istore_2
p1_1_3(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=2,SP0=SP-1},
          read(SP,V0,V1,V2,V3,V4,V),
          write_1(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_4(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    5: iload_1
p1_1_4(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,SP0=SP+1},
          read(I,V0,V1,V2,V3,V4,V),
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_5(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    6: iconst_5
p1_1_5(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {V=5,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_6(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    7: if_icmple -> 12
p1_1_6(V0,V1,V2,V3,V4,SP0,H,HO,HS,HSO,R) :-
          {SP0=SP-1,SP1=SP-2,V2>V1},
          read(SP,IV0,IV1,IV2,IV3,IV4,V1),
          read(SP0,IV0,IV1,IV2,IV3,IV4,V2),
          p1_1_7(V0,V1,V2,V3,V4,SP1,H,HO,HS,HSO,R).
p1_1_6(V0,V1,V2,V3,V4,SP0,H,HO,HS,HSO,R) :-
          {SP0=SP-1,SP1=SP-2,V2=<V1},
          read(SP,IV0,IV1,IV2,IV3,IV4,V1),
          read(SP0,IV0,IV1,IV2,IV3,IV4,V2),
          p1_1_12(V0,V1,V2,V3,V4,SP1,H,HO,HS,HSO,R).
%    10: iload_1
p1_1_7(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {I=1,SP0=SP+1},
          read(I,V0,V1,V2,V3,V4,V),
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_8(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    11: iconst_5
p1_1_8(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {V=5,SP0=SP+1},
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
          p1_1_9(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    12: iadd
p1_1_9(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
          {SP0=SP-1,VR=V1+V2},
          read(SP,V0,V1,V2,V3,V4,V1),
          read(SP0,V0,V1,V2,V3,V4,V2),
          write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,VR),
```

```
            p1_1_10(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    13: istore_2
p1_1_10(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
            {I=2,SP0=SP-1},
            read(SP,V0,V1,V2,V3,V4,V),
            write_1(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
            p1_1_11(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%   14: goto -> 16
p1_1_11(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
                        {},
                        p1_1_16(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R).
%    17: iload_1
p1_1_12(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
            {I=1,SP0=SP+1},
            read(I,V0,V1,V2,V3,V4,V),
            write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
            p1_1_13(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    18: iconst_5
p1_1_13(V0,V1,V2,V3,V4,SP,H,HS,R) :-
            {V=5,SP0=SP+1},
            write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
            p1_1_14(W0,W1,W2,W3,W4,SP0,H,HS,R).
 %    19: isub
p1_1_14(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
            {SP0=SP-1,VR=V1-V2},
            read(SP,V0,V1,V2,V3,V4,V1),
            read(SP0,V0,V1,V2,V3,V4,V2),
            write(SP0,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,VR),
            p1_1_15(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%    20: istore_2
p1_1_15(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
            {I=2,SP0=SP-1},
            read(SP,V0,V1,V2,V3,V4,V),
            write_1(I,V0,V1,V2,V3,V4,W0,W1,W2,W3,W4,V),
            p1_1_16(W0,W1,W2,W3,W4,SP0,H,HO,HS,HSO,R).
%   21: return
p1_1_16(V0,V1,V2,V3,V4,SP,H,HO,HS,HSO,R) :-
                    {HS = HSO,  unify(H,HO)}.
```

Chapter 4

Heap representation and method calls

4.1 Heap representation

We now describe how we model bytecode instructions related to Java object manipulations. Those manipulations include the creation of an object, retrieving a value from a field or modify this value. Java arrays are treated as objects with one particular field to store the length of the array in our CLP representation. Hence, we also describe array-related operations in this section.

A particularity of the *heap* is that we don't always know its size at compile time. Indeed, for some programs, it is not always possible to know how many objects will be created during the execution. A simple program in which the number of created objects depend on the user's inputs should convince the reader. Moreover, in a Java program, due to the heritage and polymorphism mechanisms, the type of an object is not always known at compilation time neither. Again those mechanisms reinforce the fact that it is impossible to statically compute the size of the heap as the type of the object impact also its size. Thus, we need a dynamic mechanism to represent the heap and the creation of the different objects. We developed two internal representations for objects, a first representation using the $assert - retract$ mechanisms of Prolog and a second representation using *Prolog lists*. Even if the two representations are functional, the representation via Prolog lists was more convenient for the analysis tools of our framework. We now present those two representations :

The representation of a a single object is the same for the two representations. An object is represented by a group of facts. A fact represents a field of a given object. Those facts are of the form $object(R, F, V)$. where $R \in \mathbb{Z}$ is the (id) reference of the object's field. The variable $F \in \mathbb{Z}$ represent the (id) number of the field represented by the fact inside the object. V is the value of the field. Thus, a field is referenced via its object id reference and its own field number inside the object.

First internal representation of the heap (assert-retract) :

The assert-retract version of our heap representation works by adding and removing facts from the Prolog database. The $writeMemory$ predicate should always be used in its $writeMemory(IN, IN, IN)$ mode. The $writeMemory$ predicate is divided in two cases :

The first definition of the predicate handles the case where the value of the field is modified. As the field of the given object is already initialized in the Prolog database, this definition of the $writeMemory$ predicate first removes the corresponding fact with a *retract* statement. Then we insert the new *object* fact into the Prolog database with the *assert* statement.

The second definition is simpler. This definition of the predicate is executed if the *retract* statement of the first definition fails. This would mean that no *object* fact with the same R and F values where previously added to the Prolog database. Thus, we only need to insert the new *object* fact in the Prolog database.

- Write Memory

  $writeMemory(R, F, V) \leftarrow$
  $\{\ \}$,
  $retract(object(R, F, \_))$,
  $assert(object(R, F, V))$.

  $writeMemory(R, F, V) \leftarrow$
  $\{\ \}$,
  $assert(object(R, F, V))$.

  The definition of the $readMemory$ predicate is pretty straightforward. The predicate should be used in its $readMemory(IN, IN, OUT)$ mode. Given an object id and a field id, the predicate returns the value of the corresponding field.

- Read Memory

  $readMemory(R, F, V) \leftarrow$
  $\{\ \}$,
  $object(R, F, V)$.

Second internal representation of the heap (Prolog lists) :

Our second heap representation uses lists to handle the pool of objects rather than adding and removing facts from the Prolog database during the execution. This solution has the advantage to be more convenient for analysis tools. However, this representation of the heap requires to initialize an empty list at the start of each program and to pass this list from predicate to predicate.

- Write Memory

  This definition of the $writeMemory$ predicate should always be used in its $writeMemory(IN, OUT, IN, IN, IN)$ mode. The predicate is of the form $writeMemory(HI, HO, R, F, V)$ where HI is the Prolog list that represents the heap before the execution of the $writeMemory$ predicate and HO this list after its execution. The R,F,V variables represents the reference of the object's field, the number of the field represented by the fact inside the object and the value of the field, respectively.

  The first clause handles the modification case with the *select* statement. If the field that we try to add to the list is already an element of this list, the first clause of the predicate removes this element and call $writeMemory$ again.

  If no element of the HI list have the same R value and the same F value as the element we try to insert, the *select* statement from the first clause fails and the second clause is executed. The second clause simply adds the *object* element at the beginning of the list.

  $writeMemory(HI, HO, R, F, V) \leftarrow$
  $\{\ \}$,
  $select(object(R, F, \_), HI, HO1)$,
  $writeMemory(HO1, HO, R, F, V)$.

  $writeMemory(HI, [object(R, F, V)|HI], R, F, V)$.

- Read Memory

  The $readMemory$ predicate should always be used in its $readMemory(IN, IN, IN, OUT)$ mode. The predicate basically runs through the elements of the list until it finds an element with the corresponding object field id's and returns the associated value.

  $readMemory([object(R, F, V)|HS], R, F, V)$.

  $readMemory([object(RR, FF, \_)|HS], O, F, V) \leftarrow$
  $\{OO \neq O \mid\mid FF \neq F\}$,
  $readMemory(HS, O, F, V)$.

The following representations of bytecode instructions are defined for the Prolog list representation of the heap. However, the modification to switch from one representation is minor as we introduced an abstraction layer via $readMemory$ and $writeMemory$.

- NEW{new}

  Create a new object. Reference of the object is pushed on the top of the stack.

  $q\ : NEW\ c\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP + 1, HS' = HS + 1, C = c\}\ ,$
  $write(SP', \tilde{IV}, \tilde{IV}', HS),$
  $writeMemory(H, H', HS, 0, C),$
  $writeMemory(H', H'', HS, 1, 0),$
  $p_{q+1}(\tilde{IV}', SP', H'', HO, HS', HSO, R).$

  $\}$

  The NEW meta-instruction defines the bytecode instruction that creates a new object into the heap. We illustrate here the representation of a *new* instruction for the creation of an object of class c. We assume that the objects of class c have only one field.

  The *write* predicate pushes the reference of the created object on the top of the operand stack. Two items are added to the list H that represents the heap via the $writeMemory$ predicate. The first call to $writeMemory$ initializes the field 0 of the object with the id of its class, which is a special field that is present in every object. This field is used to retrieve the actual type of an object. The second field of the object is initialized via the second call to $writeMemory$. The value of a field is set to zero by default. The HS variable, which represents the heap size is incremented by one as we added an object to the heap.

- GETFIELD {getfield}

  Fetch specified field of the referenced object by the poped top stack value.

  $q\ : GETFIELD\ i\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ I = i\}\ ,$
  $read(SP, \tilde{IV}, R),$
  $write(SP, \tilde{IV}, \tilde{IV}', F),$
  $readMemory(H, R, I, F),$
  $p_{q+1}(\tilde{IV}', SP, H, HO, HS, HSO, R).$

  $\}$

The $GETFIELD$ representation illustrates how to retrieve the value of a non-static object field. The JVM assumes that the reference of the object is on the top of the operand stack before executing the instruction. The n argument indicates the number of the field from which we want to get the value. The $read$ predicate is used to constraint variable R to the value of the object reference. The $readMemory$ predicate is used to retrieve the field value F from the list that represents the heap (H). The retrieved field value F is pushed on the top of the operand stack via the $write$ predicate.

The definition of the $GETSTATIC$ meta-instruction is similar to this one.

- PUTFIELD{putfield}

  Set specified field of a referenced object. The two popped top stack values must be the object reference and the value to be set. The field is specified via an instruction argument.

  $q\ : PUTFIELD\ i\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ I = i, SP' = SP - 1, SP'' = SP - 2\}\ ,$
  $read(SP, \tilde{IV}, F),$
  $read(SP', \tilde{IV}, R),$
  $writeMemory(H, H', R, I, F),$
  $p_{q+1}(\tilde{IV}, SP'', H', HO, HS, HSO, R).$

  $\}$

  The $PUTFIELD$ operation changes the value of a given field in a given object. We assume that the reference of the object and the value to be assigned to the field are on the top of the operand stack before executing the instruction. The argument indicates the number of the field we want to assign. The value to be assigned to the field (F) and the reference of the object (R) are retrieved via the $read$ predicate. The object representation is modified via $writeMemory$. the stack pointer is decremented by two as we popped the values of R and F from the operand stack.

  The definition of the $PUTSTATIC$ meta-instruction is similar to this one.

- NEWARRAY{newarray}

  Create a new array. The length of the array is determined by the popped top stack value. Reference of the array is pushed on the top of the stack.

  $q\ : NEWARRAY\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ HS' = HS + 1\}\ ,$
  $read(SP, \tilde{IV}, V),$
  $write(SP, \tilde{IV}, \tilde{IV}', HS),$
  $writeMemory(H, H', HS, -1, V),$
  $p_{q+1}(\tilde{IV}', SP, H', HO, HS', HSO, R).$

  $\}$

  The $NEWARRAY$ meta-instruction is very similar to the NEW meta-instruction with the particularity that we also initialize the size of the array. This operation requires that the length of the array (V) is at the top of the operand stack. The size of the array is retrieved via the *read* predicate. The size of the array is encoded at special field -1 via the $writeMemory$ predicate. The size of the heap is incremented by one as arrays are a special form of object in our representation.

- ARRAYLENGTH {arraylength}

  Pop the top stack value that must be an array reference and pop the length of this array.

  $q\ : ARRAYLENGTH\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ \}\ ,$
  $read(SP, \tilde{IV}, R),$
  $readMemory(H, R, -1, L),$
  $write(SP, \tilde{IV}, \tilde{IV}', L),$
  $p_{q+1}(\tilde{IV}', SP, H, HO, HS, HSO, R).$

  $\}$

  $ARRAYLENGTH$ is an array specific instruction that allows to get the size of a given array. This representation assumes that the reference of the array is on the top of the operand stack. The reference of the array R is retrieved via the *read* predicate. As the length of the the array is stored in a special field at index -1 at the creation of the array, the length L is simply retrieved via the $readMemory$ predicate. The length is popped on the top of the operand stack via the *write* predicate.

- ARRAYLOAD {iaload,laload,saload,faload,aaload,daload}

  Load a value from an array. The array reference and index of the element to retrieve must be at the top of the stack and are popped.

  $q\ : ARRAYLOAD\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP' = SP - 1\ \}\ ,$
  $read(SP', \tilde{IV}, R),$
  $read(SP, \tilde{IV}, I),$
  $readMemory(H, R, I, V),$
  $write(SP, \tilde{IV}, \tilde{IV}', V),$
  $p_{q+1}(\tilde{IV}', SP', H, HO, HS, HSO, R).$

  $\}$

  The $ARRAYLOAD$ meta-instruction group bytecode instructions that aim to retrieve a value located at a specified index in a given array. The retrieved value is pushed on the top of the operand stack. This operation assumes that the reference of the array R and the index I are on the top of the operand stack before the execution of the $ARRAYLOAD$ meta-instruction. The value of those two variables are popped via the *read* predicates. The value is retrieved via the $readMemory$ predicate and pushed on the operand stack via the *write* predicate. the stack pointer is decremented by one as we popped two R and I but pushed only the retrieved value from the array.

- ARRAYSTORE {iastore,lastore,sastore,fastore,aastore,dastore}

  Store a value into an array. The first three top stack values must be the array reference, index and the value to be stored at the given index and are popped during the operation.

  $q\ : ARRAYSTORE\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ SP''' = SP - 3,\ SP'' = SP - 2,\ SP' = SP - 1\ \}\ ,$
  $read(SP'', \tilde{IV}, R),$
  $read(SP', \tilde{IV}, I),$
  $read(SP, \tilde{IV}, V),$
  $writeMemory(H, H', R, I, V),$
  $p_{q+1}(\tilde{IV}, SP'''', H', HO, HS, HSO, R).$

  $\}$

The $ARRAYSTORE$ meta-instruction group bytecode instructions that aim to store a given value at a specified index of an array. This representation assumes that the references of the array, the index and the value to be stored are on the top of the operand stack. Those values are retrieved via $read$ predicates. The $writeMemory$ predicate is used to store assign the value to the given index of the array. The stack pointer is decremented by 3 as we popped the R,I and V values from the operand stack.

- MULTINEWARRAY{multinewarray}

  Create a new multidimensional array. The number of dimensions is determined via an instruction argument and the length of each dimension is from the n first popped top stack values. Reference of the array is pushed on the top of the stack.

  $MULTINEWARRAY$ allows to create multidimensional arrays. The number of dimension of the array is passed as an argument to the instruction whereas the length of the respective dimensions are assumed to be on the top of the operand stack. We illustrate the representation of the $MULTINEWARRAY$ for a two dimensions array.

  (Example for 2 dimensions array : d = 2)

  $q\ : MULTINEWARRAY\ d\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ D = d,\ SP' = SP{-}1,\ AR1 = HS{+}1,\ AR2 = HS{+}2,\ HS' = HS{+}3\}$,
  $read(SP', \tilde{IV}, V1)$,
  $read(SP, \tilde{IV}, V2)$,
  $write(SP', \tilde{IV}, \tilde{IV}', HS)$,
  $writeMemory(H, H', HS, -1, D)$,
  $writeMemory(H', H'', AR1, -1, V1)$,
  $writeMemory(H'', H''', AR2, -1, V2)$,
  $p_{q+1}(\tilde{IV}', SP', H''', HO, HS', HSO, R)$.

  $\}$

  The two lengths of the respective dimensions are retrieved in the variables $V1$ and $V2$ via the $read$ predicates. The reference of the multidimensional array is pushed on the top of the operand stack via the $write$ predicate. A multidimensional array of n dimensions is represented as an array of length n that contains references to other arrays. That's how a multidimensional array is represented as an array of array. Thus, the respective dimensions are stored into the -1 fields of each array via the $writeMemory$ predicates. The stack pointer is decremented by one as we popped the two dimensions values but pushed the reference of the new multidimensional array.

4.2 Method calls

We now describe how method calls are handled. This section also explains how the dynamic linking process works and how it is handled by our representation in CLP.

Java provides a great number of possibilities when it comes to method calls. The way methods are handled depends on their access and non-access modifiers. Moreover, the heritage mechanism provided by Java also elaborates the method call mechanism. There are two major types of methods. The most simple case are *static* methods, which do not require to know the concrete object instance on which they are executed to be executed. As a consequence, for static methods, the definition of the method to execute is always known at compile time. The second type of methods are called instance methods. Those methods require to know the concrete object on which they are executed to be executed and the definition of the method to be executed depends of the actual type of the instance (i.e. not the declared type). Hence, the definition of the method to be executed could change from one execution to another. Due to this, we need to provide a static lookup mechanism that is able to choose dynamically the right definition of method to execute, depending on the actual type of the instance at the execution time [16].

- INVOKE
  {invokestatic,invokespecial,invokeinterface,invokevirtual}

  Invoke a specified method. Arguments are popped from the stack.

  The *invoke* bytecode instructions all take a constant pool entry as argument. Those entries have to be symbolic references methods. As the *invoke* instructions are a bit more complex than the previous bytecode instructions we detail our $INVOKE$ meta-instruction for the different categories. The symbolic references have to be resolved differently depending on the invocation type.

  The *invokestatic* bytecode instruction is used to invoke static methods, their definition is always known at compile time. The mechanism used to resolve the symbolic reference in this case is called *static binding*. The symbolic reference describes the name of the class in which the method to invoke is located and the signature of the method. As it is a static method, information is sufficient to know which definition a method must be called [16].

The *invokespecial* instruction is particular invocation statement for some instance methods. *Invokespecial* is used to call constructors, private methods and methods that are called via the *super* keyword. In those cases, even if those methods are instance methods, they require a static binding mechanism. A dynamic binding applied on those cases would lead to wrong results.

The $invokeinterface$ and *invokevirtual* instructions are used to invoke the other instance methods. For those cases, the symbolic reference contains the name of the declared type of the instance and the signature of the method. This is where the dynamic linking mechanism process operates. The symbolic references point out to the declared type of the instance, however it is the actual type of the instance that determines the definition of method to invoke. In order to resolve this reference a lookup mechanism is established. A clause is generated for the declared type and each of its subclasses.Allowing to cover every potential types of the instance. for each subclass, the definition of the method to be called is statically defined by pointing to the first method that matches the signature when going upwards in the class hierarchy. We now give our representation of the *invokevirtual* instruction when invoking a non void method m from an instance declared of type C :

$$q \ : \ invoke \rightarrow \begin{cases} p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow \\ \{ \\ SP_i = SP - (n - i), \\ SP\_CALL = m, \\ CLASS\_ID = class\_id \\ \}, \\ read(SP_i, \tilde{IV}, A_i), \\ read(SP, \tilde{IV}, REF), \\ readMemory(H, REF, 0, CLASS\_ID), \\ p_{x_c}(REF, \tilde{A}, SP\_CALL, H, HO_{CALL}, HS, HSO_{CALL}, R'), \\ write(SP_n, \tilde{IV}, \tilde{IV}', R'), \\ p_{q+1}(\tilde{IV}', SP_n, HO_{CALL}, HO, HSO_{CALL}, HSO, R). \end{cases}$$

Where

$$\begin{array}{l} n = nb\ args\ of\ m \\ m = nb\ of\ local\ variables\ of\ m \\ \forall\ class\_id\ is\ a\ subclass\ id\ of\ C \\ \forall\ i\ \in\ [0..n] \end{array}$$

The representation when invoking a void method is similar except we don't push the returned R variable to the operand stack of the calling method.

- RETURN{iretrun,dreturn,lreturn,sreturn,areturn}

  Return a value from non-void method or return from method.

  The representation of the $RETURN$ meta-instruction is pretty straightforward. We assume that the value to be returned by the method is on the top of the operand stack. This value is simply written in variable R.

  $q\ : RETURN\ \rightarrow \{$
  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ HSO = HS, unify(H, HO)\}\ ,$
  $read(SP, \tilde{IV}, R).$

  $\}$

Our $DEMO$ example can now be modified to include some object manipulations and method calls. We introduce a new class $ARIT$ that holds 3 fields a,b and $result$. We also add two methods that make a computation over the variables a and b and stores the result in the $result$ variable.

```
public class DEMO {

public static void main(String[] args) {

    int sum_res = 0;

    ARIT o = new ARIT();
    o.sum(5,3);
    sum_res = o.result;
    }
}

public class ARIT {
    int result;

    public void sum(int a, int b){
        result = a + b;
    }
}
```

(a) Source code version

```
//DEMO.MAIN
  0:iconst_0
  1:istore_1
  2:new #2; //ARIT
  5:dup
  6:invokespecial #3; //ARIT."<init>":()V
  9:astore_2
 10:aload_2
 11:iconst_5
 12:iconst_3
 13:invokevirtual #4; //ARIT.sum:(II)V
 16:aload_2
 17:getfield #5; //ARIT.result:I
 20:istore_1
 21:return

//ARIT.SUM
 0:aload_0
 1:iload_1
 2:iload_2
 3:iadd
 4:putfield      #2; //result:I
 7:return
```

(b) Bytecode version (main)

Figure 4.1: A Java program with methods

The application of our decompilation rules over the Java bytecode sequence described by the Figure 4.1b result in the following CLP program :

Listing 4.1: A Java program with methods (Decompiled)

```
%METHOD ARIT.SUM
%    0: aload_0
p1_1_0(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP=2,SP'=SP+1},
          read(0,V0,V1,V2,V3,V4,V5,V),
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_1(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    1: iload_1
p1_1_1(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          read(1,V0,V1,V2,V3,V4,V5,V),
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_2(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    1: iload_2
p1_1_2(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          read(2,V0,V1,V2,V3,V4,V5,V),
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_2(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    3: iadd
p1_1_3(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1,VR=V1+V2},
          read(SP,V0,V1,V2,V3,V4,V5,V1),
          read(SP',V0,V1,V2,V3,V4,V5,V2),
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,VR),
          p1_1_4(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    4: putfield 2
p1_1_4(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1, SP''=SP-2},
          read(SP,V0,V1,V2,V3,V4,V5,V1),
          read(SP',V0,V1,V2,V3,V4,V5,V2),
          writeMemory(H,H',V2,1,V1),
          p1_1_5(V0,V1,V2,V3,V4,V5,SP'',H',HO,HS,HSO,R).
%    7: return
p1_1_5(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {HS = HSO, unify(H,HO)}.

%METHOD DEMO.MAIN
%    0: iconst_0
p2_1_0(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP=2,SP'=SP+1},
          write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,0),
          p2_1_1(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    1: istore_1
p2_1_1(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1},
          read(SP,V0,V1,V2,V3,V4,V5,V),
          write(1,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p2_1_2(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    2: new 2
p2_1_2(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1,S'=HS+1},
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,HS),
          writeMemory(H,H',HS,0,1),
          writeMemory(H',H'',HS',1,0),
          p2_1_3(W0,W1,W2,W3,W4,W5,SP',H'',HO,HS',HSO,R).
%    5: dup
p2_1_3(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          read(SP,V0,V1,V2,V3,V4,V5,V),
          write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
```

```
            p2_1_4(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    6: invokespecial 3
p2_1_4(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {},
            read(SP,V0,V1,V2,V3,V4,V5,V),
            p1_0_0(V,0,_,H,H_CALL,HS,HS_CALL,R),
            p2_1_5(V0,V1,V2,V3,V4,V5,SP,H_CALL,HO,HS_CALL,HSO,R).
%    9: astore_2
p2_1_5(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP'=SP-1},
            read(SP,V0,V1,V2,V3,V4,V5,V),
            write(2,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
            p2_1_6(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   10: aload_2
p2_1_6(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP=2,SP'=SP+1},
            read(2,V0,V1,V2,V3,V4,V5,V),
            write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
            p1_1_7(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   11: iconst_5
p2_1_7(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP=2,SP'=SP+1},
            write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,5),
            p2_1_8(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   12: iconst_3
p2_1_8(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP=2,SP'=SP+1},
            write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,3),
            p2_1_9(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   13: invokevirtual 4
p2_1_9(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP'=SP-2,SP''=SP-1,ClassID = 1},
            read(SP',V0,V1,V2,V3,V4,V5,V1),
            read(SP'',V0,V1,V2,V3,V4,V5,V2),
            read(SP,V0,V1,V2,V3,V4,V5,V3),
            readMemory(H,V1,0,ClassID),
            p1_1_0(V1,V2,V3,0,0,0,SP_CALL,H,H_CALL,HS,HS_CALL,R),
            p2_1_10(V0,V1,V2,V3,V4,V5,H_CALL,HO,HS_CALL,HSO,R).
%   16: aload_2
p2_1_10(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP'=SP+1},
            read(2,V0,V1,V2,V3,V4,V5,V),
            write(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
            p2_1_11(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   17: getfield 5
p2_1_11(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {},
            read(SP,V0,V1,V2,V3,V4,V5,V),
            write(SP,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,FV),
            readMemory(H,V,1,FV),
            p2_1_12(W0,W1,W2,W3,W4,W5,SP,H,HO,HS,HSO,R).
%   20: istore_1
p2_1_12(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP'=SP-1},
            read(SP,V0,V1,V2,V3,V4,V5,V),
            write(1,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
            p2_1_13(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%   21: return
p2_1_13(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {HS = HSO, unify(H,HO)}.
```

Chapter 5

Optimisations

As mentioned in chapter 3 our default representation is pretty verbose and the one-to-one mapping between bytecode instructions and CLP clauses leads to Prolog programs that are not particularly efficient. Even if efficiency is not our main interest for this work, in order to reduce the amount of code processed by analysis tools we developed an optimization phase in our decompilation process. This chapter aims to describe the applied optimisations.

Virtual machines (VM) allow developers to create program executables that are not directly linked with the physical architecture of the processor. Thus, Virtual Machines allows their programs to be easily interpreted or compiled on different platforms as they make few assumptions about the target hardware(registers, CPU features). The Java Virtual Machine discussed in this work, uses a virtual stack architecture, rather than the register architecture that dominates in real processors. The stack-based architecture allows smaller VM code so less code must be fetched inside the Virtual Machine by VM instruction executed [22]. In addition, stack-code is gradually easier to generate in the compiler than register code and eliminates the need for a complicated register allocator [9] [4] [22].

On the other hand, stack machines require more VM instructions for a given algorithm to be expressed. In general, a given computation can often be expressed using fewer register machine instructions than stack instructions. As an example, the local variable assignment $b = a + b$ would be translated in the following pseudo Java bytecode sequence :

```
iload a
iload b
iadd
istore b
```

Where a register based machines could directly put the sum of two registers in the desired destination register.

From those observations, we decided to provide some optimisation mechanisms in order to reduce the amount of code (and reduce its complexity) produced by our decompiler for a given program. For this purpose we developed an approach based on the detection and transformation of Java bytecode patterns during the sequential decompilation. The following sections explain this process and associated results.

5.1 pattern detection

In the stack-based JVM, operands are pushed from local variables onto the operand stack before they can be used, and results must be retrieved from the stack to local variables. However, in our CLP representation, the local variable table and the operand stack are both represented as a set of Prolog variables. Due to this representation, most of these stack push and pop operations are redundant in our CLP representation, instructions can directly use local variables with no additional cost.

As our default decompilation process is a one-to-one mapping between bytecode instructions and CLP clauses, our CLP representations are basically the declarative representation of each instruction. Hence, reducing the number of unnecessary push and pop operations would directly reduce the number of CLP clauses for a given program. This optimisation is fairly interesting as it is assumed that more than 40% of executed instructions in common Java benchmarks consists of loads and stores between local variables and the stack [22].

Moreover, our CLP representation models access and modification of the stack by the *read* and *write* predicates presented in chapter 3. Indeed, we need to provide a dynamic stack pointer mechanism to access and store values of the operand stack. However, as described in section 3.3, this dynamic mechanism implies to copy the $\tilde{IV}$ set each time an element is pushed on the top of the stack or in the local variable table. Numerous calls to those predicates could then present a problem of efficiency for analysis tools.

Even if we can't dispense with the *read* and *write* predicates, it is possible to reduce their use. After having inquired some Java bytecode program examples, one can rapidly observe that recurrent patterns are used to accomplish some manipulations. Furthermore, for some of those patterns, it is possible to statically know where a given value will be written in the stack. Thus, for those special cases we can totally bypass the *read* and *write* dynamic predicates mechanisms.

Within the context of our work, we provided two optimisations related to pattern detection to our decompiler.

- Initialisation of a variable.
- Arithmetic operation with two operands.

We now describe the mechanisms behind each of those pattern based optimisations. The goal here is to detect some patterns in a sequence of bytecode instructions. As our decompilation process is sequential, the default process is to take instructions one by one in the order of occurence and to translate each instruction in its CLP form. However, within the context of our optimisation module we put in place a kind of sliding window mechanism in order to analyse the n following instructions to come. The size of the sliding windows should be the number of instruction of the longest defined pattern in order to detect all developed patterns. However, allowing the user to define the size of the sliding window via some optimisation levels can be useful in certain cases in order to allow or exclude some optimisations.

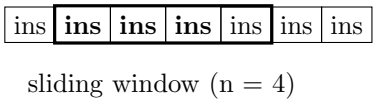

Figure 5.1: Illustration of the sliding windows mechanism

The mechanism behind the sliding window is pretty simple. During the sequential translation of a given sequence of bytecode instructions, the decompiler analyses the n following instructions to come. At a given instruction i, if a pattern of length x is detected from instruction i in the sliding window, the x following instructions are translated into their respective pattern representation. Figure 5.1 represents the case where a pattern consisting of 3 instructions has been detected in a sliding window of size 4.

5.1.1 Initialisation of a variable

Variable initialisation operations are quite frequent in any programming language. In Java bytecode, initialisations of variables are always performed in the same way. First, a constant is pushed on the top of the stack. Then the value that was just pushed on the top of the stack is copied into a specified slot of the local variable table.

Even if a great number of combinations are possible when it comes to initialize a variable in Java bytecode (i.e. iconst_5 → istore_2, bipush 10 → istore_3, fconst_3 → fstore 10, ..), all those combinations follow the same pattern in our CLP representation thanks to our well defined meta-instruction groups. Those combinations are all defined as a $CONST$ meta-instruction directly followed by a $STORE$ meta-instruction.

Figure 5.2: Initialisation of a variable pattern

The pattern allows us to not work with the operand stack any more to perform this operation. Indeed, as we see those two instructions as a whole, we simply have to put a given value in a specified slot of the local variable table. Thus, when representing this pattern by a single CLP clause, we don't need to deal with dynamic *read* and *write* predicates at all. We now give the definition of our initialisation variable pattern :

- INITVAR {CONST c → STORE i}

  $q\ : INITVAR\, c,\ i\ \rightarrow \{$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ V_i' = c\} \ \cup\ id_{-d},$
  $p_{q+2}(\tilde{IV'}, SP, H, HO, HS, HSO, R).$

  $\}$

Besides grouping the two meta-instructions $CONST$ and $STORE$ into a single CLP clause, this optimised representation dispenses us from making 3 calls to *read* and *write* predicates for each variable initialisation.

5.1.2 Arithmetic operation with two operands

The way arithmetic operations are performed in Java bytecode is pretty heavy weight. Each basic arithmetic operation (i.e. addition, subtraction, modulo, etc..) requires at least four bytecode instructions. Two load instructions are first required to push the two operands of the operation on the top of the stack. Then the operation is actually performed with a corresponding instruction and the result is stored into the local variable table via a store instruction.

Once again, the number of possible combinations is even more important than for the initialisation of a variable. But once more, our CLP representation allows us to take in account all those combinations by a simple sequence of meta-instructions. Considering that the meta-instructions ADD, SUB, DIV, $MODULO$, MUL could be regrouped under a same meta-meta-instruction $ARIT\_OP$.
The arithmetic operations are represented by the
$LOAD \rightarrow LOAD \rightarrow ARIT\_OP \rightarrow STORE$ sequence.

Figure 5.3: Arithmetic operation with two operands

By grouping those four instructions into a single CLP clause, this pattern prevents us from using *read* and *write* dynamic predicates. When considering the sequence of instructions a whole, we get the two operand locations. There is no need to push operands on the top of the operand stack as the operation is also performed in the CLP clause. Finally we need to store the result of the operation in a specified field of the local variable table. As our representation allows us to directly manipulate values from the local variable table, we don't need to interact with the operand stack. The CLP representation of our arithmetic operation pattern is defined as follows :

- ARIT_OP_PATTERN {LOAD v → LOAD w → ARIT_OP → STORE i}

  $q\ : ARIT\_OP\_PATTERN\ c,\ i\ \rightarrow \{$

  $p_q(\tilde{IV}, SP, H, HO, HS, HSO, R) \leftarrow$
  $\{\ V_i' = V_v\ op\ V_w\} \ \cup\ id_{-d},$
  $p_{q+4}(\tilde{IV'}, SP, H, HO, HS, HSO, R).$

  $\}$

Where *op* is the operation sign depending on the arithmetic operation represented by $ARIT\_OP$. Once again, besides grouping four CLP clauses into a single clause we save 9 calls to *read* and *write* predicate each time an arithmetic operation of this type is performed.

5.1.3 Some results

We now present an example from our decompiler that aims to illustrate how the optimisations presented in the previous sections can reduce the complexity of a given program. Our example consists of a method that simply initialises two variables and assigns the sum of those two variables to the first variable. The following figure illustrates the original Java program and its corresponding bytecode sequence.

```java
public class DEMO {

public static void main(String[] args) {
    int a = 18;
    int b = 2;
    a = a + b;
    }
}
```

(a) Source code version

```
0:    bipush  18
2:    istore_1
3:    iconst_2
4:    istore_2
5:    iload_1
6:    iload_2
7:    iadd
8:    istore_1
9:    return
```

(b) Bytecode version (main)

Figure 5.4: A last Java program

The application of our default decompilation rules over the Java bytecode sequence described by the Figure 5.4b result in the following CLP program; represented in Listing 5.1 :

Listing 5.1: A last Java program with no optimisations (Decompiled)

```
%    0: bipush 18
p1_1_0(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1,SP=2},
          write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,18),
          p1_1_1(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    2: istore_1
p1_1_1(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1},
          read_1(SP,IV0,IV1,IV2,IV3,IV4,V),
          write_1(1,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_2(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    3: iconst_2
p1_1_2(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,2),
          p1_1_3(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    4: istore_2
p1_1_3(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1},
          read_1(SP,IV0,IV1,IV2,IV3,IV4,V),
          write_1(2,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_4(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    5: iload_1
p1_1_4(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          read_1(1,IV0,IV1,IV2,IV3,IV4,V),
          write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_5(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    6: iload_2
p1_1_5(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP+1},
          read_1(2,IV0,IV1,IV2,IV3,IV4,V),
          write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
          p1_1_6(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    7: iadd
p1_1_6(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
          {SP'=SP-1,VR=V1+V2},
          read_1(SP,IV0,IV1,IV2,IV3,IV4,V1),
```

```
            read_1(SP',IV0,IV1,IV2,IV3,IV4,V2),
            write_1(SP',V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,VR),
            p1_1_7(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    8: istore_1
p1_1_7(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {SP'=SP-1},
            read_1(SP,IV0,IV1,IV2,IV3,IV4,V),
            write_1(1,V0,V1,V2,V3,V4,V5,W0,W1,W2,W3,W4,W5,V),
            p1_1_8(W0,W1,W2,W3,W4,W5,SP',H,HO,HS,HSO,R).
%    9: return
p1_1_8(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {HSO = HS, unify(H,HO)}.
```

The application of our optimised decompilation rules on the same program result in the CLP program given in Listing 5.2:

Listing 5.2: A last Java program with optimisations (Decompiled)

```
%VARIABLE INITIALISATION \index{pattern}pattern {0: bipush 18 ->    2: istore_1}
p1_1_0(V0,V1,V2,V3,V4,V5,SP,H,HS,R) :-
            {SP = 2},
            p1_1_2(V0,18,V2,V3,V4,SP',H,HO,HS,HSO,R).
%VARIABLE INITIALISATION \index{pattern}pattern {3: iconst_2 ->    4: istore_2}
p1_1_2(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {},
            p1_1_4(V0,V1,2,V3,V4,SP,H,HO,HS,HSO,R).
%BINARY OPERATOR \index{pattern}pattern {5: iload_1 ->    6: iload_2 ->
7: iadd ->    8: istore_1}
p1_1_4(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {RES = V1+V2},
            p1_1_8(V0,RES,V2,V3,V4,SP,H,HO,HS,HSO,R).
%    9: return
p1_1_8(V0,V1,V2,V3,V4,V5,SP,H,HO,HS,HSO,R) :-
            {HSO = HS, unify(H,HO)}.
```

The reader should note that the optimised program is much more efficient and concise than the original one.

5.2 Some performance results

We now illustrate some performance results for the set of programs given as examples in chapters 3 and 4. This section aims to compare execution times and the number of lines for the different decompiled programs with or without the optimisations. In order to simplify the presentation we named the programs as follow :

- **prog1**

  Denotes the Java program illustrated in Figure 3.5.
  It is a very simple Java program that initialises two integers. This program was introduced in order to illustrate stack manipulation instructions.

- **prog2**

  Denotes the Java program illustrated in Figure 3.6.
  It is a simple Java program that performs some additions. This program was introduced in order to illustrate arithmetic related instructions.

- **prog3**

  Denotes the Java program illustrated in Figure 3.7.
  It is a Java program that was introduced in order to illustrate some control flow instructions. The body of the program is composed of an if-then-else structure.

- **prog4**

  Denotes the Java program illustrated in Figure 4.1.
  This Java program was introduced in order to illustrate object-related instructions and method calls. This program is composed of two classes. A first class initialize an object of the type of the second class and call some methods on that object.

5.2.1 Execution time

One first thing to be considered when discussing the performance results is the execution time of the respective program representations. Figure 5.5 and 5.6 presents the execution times for JVM and CLP representations respectively.

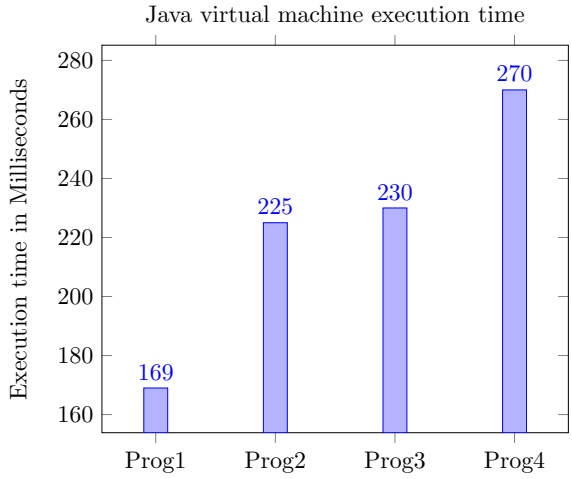

Figure 5.5: JVM executions times

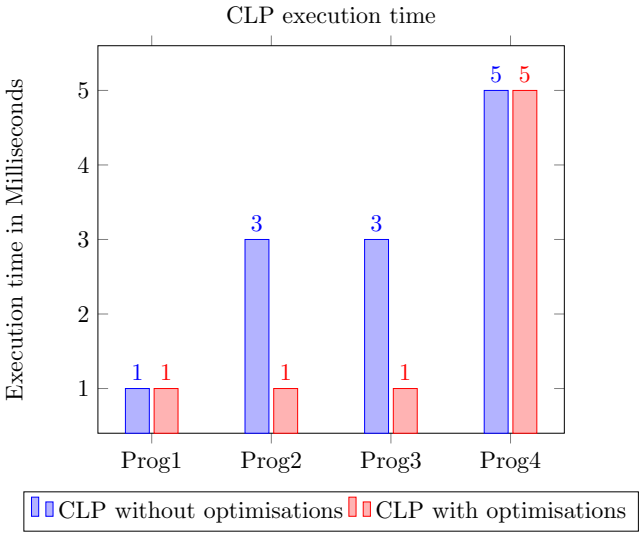

Figure 5.6: CLP executions times

One first thing that the reader should note by comparing Figures 5.5 and 5.6 is that there is a huge difference between the execution times of the bytecode representations and the CLP representations. Bytecode versions of programs take much more time to execute than the CLP representations. However, this great difference is caused by the loading of the JVM in order to execute the bytecode instructions.

However, a relevant comparison to analyse is the one of the Figure 5.6. The graph presented in this figure compares the execution times of the different programs in the CLP representation for the optimised and non-optimised versions. For *prog*1 and *prog*2 the execution times does not differ between the optimised and non-optimised version. However, a consequent improvement of the execution time can be observed for *prog*2 and *prog*3. This difference is due to the fact that the patterns used for our optimisations are less present in programs 1 and 2 than programs 2 and 3.

5.3 Number of lines

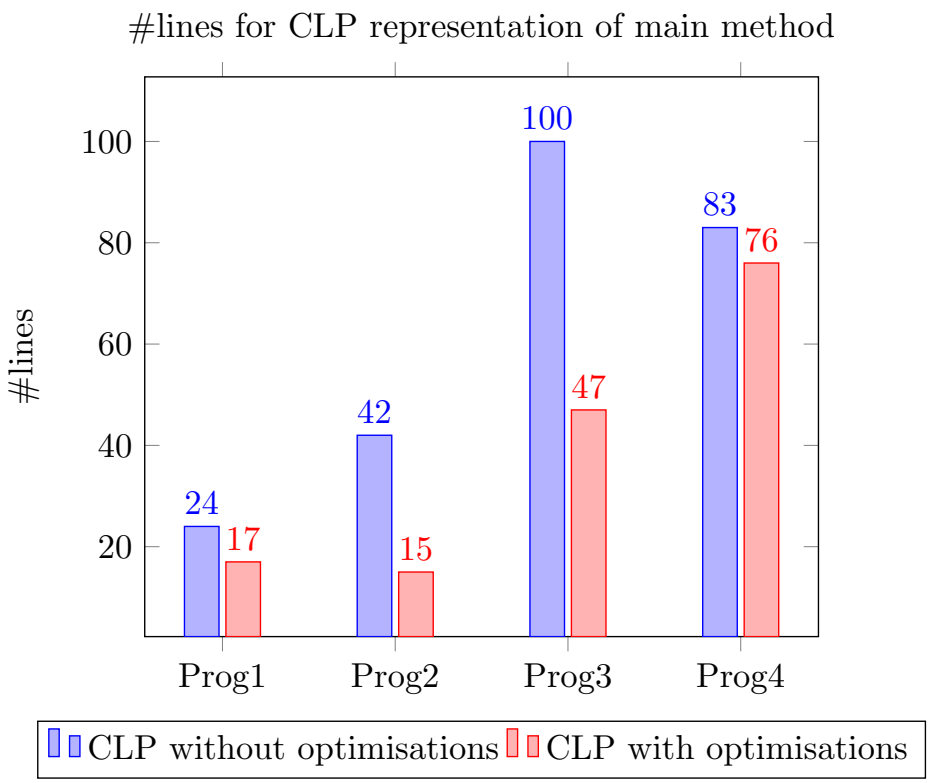

Figure 5.7: CLP executions times

Another relevant measure to compare the optimised and non-optimised versions of our CLP representations is the number of lines of the CLP program. Figure 5.7 illustrates the number of lines for the representation of the main method in CLP for each program. Even if for *prog*1 and *prog*2, the number of removed lines of the optimised version is not very satisfying, the number of lines of *prog*2 and *prog*3 are almost divided by two in their optimised versions.

*Prog*2 and *prog*3 give more interesting results than *prog*1 and *prog*2 because implemented bytecode patterns are more present in those two programs. It is important to note that those results are obtained with a set of only two optimisation patterns. The following section gives some ideas to extend the list of optimisation patterns.

5.4 Other possible optimisations

The ideas behind the pattern mechanism defined in sections 5.1.1 and 5.1.2 could easily be extended to other constructs. This would help to greatly reduce the number of calls to *read* and *write* predicates as well as unnecessary stack interactions. As an example we give some object-related patterns that could be implemented following the same mechanisms.

A first recurrent object-related pattern is the sequence of bytecode instructions that appear when creating an object. Indeed, objects are often created in Java and are always represented by the following sequence of meta-instructions :

```
NEW
DUP
INVOKESPECIAL
STORE
```

Figure 5.8: Sequence of bytecode instructions related to object creation

As illustrated by this sequence of instructions, in order to create an object, we first have to define a reference for the new object to be created and push this reference on the top of the operand stack. This is realised via the NEW meta-instruction. Next, the new object reference is always duplicated on the top of the stack. The reference is duplicated so that the two following meta-instructions (invokespecial and store) can pop this object reference. In the case of an object creation, the INVOKESPECIAL meta-instruction always calls a constructor of the object. A first copy of the new object reference is used by this meta-instruction. Finally, the second copy of the object reference is popped by the STORE meta-instruction in order to store the new object reference in the local variable table of the current method. This sequence of bytecode instructions could be translated by a single CLP predicate to extend our optimisations patterns.

Another object-related operation that leads to a recurrent pattern is the

assignment of a local variable with an object's field value. Figure 4.1 illustrates this operation with $sum\_res = o.result$; instruction. As this kind of operation is also represented by a same sequence of meta-instructions in Java bytecode it could also be part of our pattern optimisations. This operation is represented by the following sequence of bytecode instructions :

```
LOAD
GETFIELD
STORE
```

Figure 5.9: Sequence of bytecode instructions related to the affectation of a variable with an object field

This sequence of instruction illustrates the fact that in order to perform this operation, the reference of the object is first pushed on the top of the operand stack with the LOAD meta-instruction. Then the field of the object is retrieved via the GETFIELD meta-instruction and the value of the given field is stored in a variable via the STORE meta-instruction. As this sequence of meta-instructions is always the same when performing an operation of this type, it could have been included in our pattern optimisations.

A last example of recurrent patterns found in Java bytecode is the pattern that occurs when calling a method. As there are different types of method calls in Java, the related bytecode patterns obviously differs with the type of call. Here we illustrate the Java bytecode pattern that appears when calling a non-static void method with two constants passed as arguments. This type of operation is illustrated in Figure 4.1 with the operation $o.sum(5, 3);$. This type of call is represented by the instructions sequence :

```
LOAD
CONST
CONST
INVOKEVIRTUAL
```

Figure 5.10: Sequence of bytecode instructions related to the call of a non-static void method with two constants passed as arguments

As it is a non-static method, the reference of the object on which the method is called is first pushed on the top of the operand stack with LOAD.Next, the two arguments are passed to the method via two consecutive CONST meta-instruction that both push a constant on the top of the operand stack. It is clear that the number of CONST meta-instruction directly depends of the number of arguments of the method. Thus, the pattern could be easily generalized for the same type of call with an undefined number of arguments. Finally, the method is invoked via the INVOKEVIRTUAL meta-instruction. This type of sequence could be turned in a single CLP predicate that would directly load all necessary items on the top of the operand stack at once, thus avoiding unnecessary *read* and *write* calls.

It should be clear that those three examples are only illustrations of what future optimisation patterns could be. We think that there is a great number of optimisation patterns to be implemented, allowing to produce more efficient and more concise CLP programs.

Patterns are not the only optimisations that could be done, the stack representation presented in section 3.3 could be implemented more efficiently. Because this project has been developed in parallel with the similarity analysis tool presented in [27], we consciously chose to represent the operand stack as a sequence of variables. We restricted ourselves to this representation because it was not clear if the analysis tool could be able to handle Prolog's list mechanism or not. The Prolog list mechanism would certainly have provided us with a way to handle stack interactions more concisely than with the *read* and *write* predicates. However, we can now confirm that the similarity analysis tool could be able to handle Prolog lists [27]. Hence, a possible way to improve our CLP representation would be to add a stack representation with list mechanism.

The fact that certain analysis tools may have some trouble with Prolog lists is also the reason why we provide two representations for the heap (section 4.1). Working in this way grants us to be compatible with a greater number of analysis tools.

Chapter 6

Conclusions

The work presented in this document has been developed as a part of an algorithm recognition framework in binary code. The approach of the framework is to first translate binary code into Horn clauses. Then, for two given programs, they are considered as implementing the same algorithm if their Horn clause representations can be reduced to a single common set of clauses by means of a sequence of transformations. The main goal of our work was to develop a Java bytecode decompiler that translates Java bytecode programs into a CLP declarative representation that is compliant with the framework as well as with some other analysis tools. This decompiler acts as the front-end of the framework. Hence, in order to provide a universal representation, a major part of the work was dedicated to the adaptation of the CLP declarative representation presented in [25] for the Dalvik virtual machine.

Our work proposes some answers to the direct linking between Java bytecode and an executable declarative representation. We limited the scope of our work to Java programs that manipulate numbers only, in fact the only primitive type that we do not take in account is 'char'. The extension to this type is not difficult but is not relevant for the current version of the framework. Also, some of the Java bytecode instructions are not taken in account almost all of them are handled by our decompiler. We deliberately put aside thread and exception related instructions as well as bitwise operators to focus all the other constructs of Java.

The particularity of our work is that the transformation from bytecode to the declarative representation is the result of a direct mapping between Java bytecode and the declarative clauses. Other known approaches that transform Java bytecode to a declarative representation are not direct translations and pass through intermediate representations (section 1.2.1). However, those previous works have been very useful for the elaboration of our decompiler.

The decompilation process that we put in place is pretty straightforward. The decompiler performs a sequential translation of a sequence of bytecode instructions that composes the program with respect to its context. Each of the proposed CLP representation for the bytecode instructions are defined around a same structure that allows to represent and manipulate the local variables, the operand stack and the heap as defined by the Java Virtual Machine. As our CLP representation is a level of abstraction higher than the Java bytecode instructions, a single CLP clause can generally translate a set of bytecode operations that belong to the same family (i.e. the four bytecode instructions that perform an addition are represented by the same CLP construct). Our decompiler implements all the decompilation rules that were described in this paper. The operand stacks and local variable table are represented via a set of variables. We also proposed two representations for the heap, a first approach based on assert-retract Prolog mechanisms, and a second approach based on Prolog lists.

The fact that the Java Virtual machine is a stack based machine typically implies a substantial number of bytecode instructions in order to model a given algorithm. This is a weakness of our default approach (i.e. without optimisations), as each bytecode instructions of a program is translated in its CLP equivalent, our resulting CLP programs contains at least as many clauses as the number of instructions of the original program. As our decompiler has been developed to produce CLP programs that could be easily handled by analysis tools, the produced output had to be suitable for transformations. Series of transformations often require a lot of CPU time, thus some optimisations have been developed over our default decompilation process. Those optimisations are based on bytecode patterns. We presented optimisations based on the pattern behind variable initialization and arithmetic operations. The idea behind those optimisations is that for certain constructs(i.e. sequence of bytecode instructions) it is possible to translate that sequence of bytecode instructions into a single CLP clause that is much more efficient that the sequential translation of each bytecode instruction of the sequence. The examples illustrated in the paper show that this type of optimisation can really reduce the complexity of the produced CLP programs.Those two optimisations are implemented by our decompiler and extension to other patterns is pretty easy.

As mentioned in chapter 5, other optimisations could have been made to reduce the complexity of our decompiler. Future work could consist in studying alternative representations of the operand stack and extend the list of pattern-based optimisations.

With respect to that, we think that future work should first focus on reducing the complexity of the produced CLP programs rather than extending the approach to non-numerical aspect of Java programs.

Index

Bibliography

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. *Proceedings of the 17th ACM SIGPLAN conference on Object oriented programming systems languages, and applications(ACM Press)*, pages 311–330, 2002.

[2] C. Alias and D. Barthou. Algorithm recognition based on demand-driven dataflow analysis. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 296–305, 2003.

[3] L. O Andersen. Program analysis and specialization for the c programming language. 1994.

[4] Martin Anton Ertl. Implementation of stack-based langages on register machines. 1996.

[5] Anton Arhipov. Mastering the java bytecode. 2012.

[6] K Ashok and David Harel. Horn clause queries and generalizations. *Journal of logic programming (1)*, pages 1–15, 1985.

[7] William C. Benton and Charles N. Fisher. Interactive, scalable, declarative program analysis : From prototype to implementation. 2007.

[8] A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same ? *Bull Symbolic Logic*, pages 145–168, 2009.

[9] Gregg David, Andrew Beatty, Kevin Casey, and Brian Davis. The case for virtual register machines. 2004.

[10] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Proving horn clause specifications of imperative programs. 2015.

[11] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Horn clause transformation for program verification. 2016.

[12] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyse and optimize object-oriented programs. *Programming Languages and Systems 23*, pages 30–72, 2001.

[13] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. pages 114–126.

[14] Marco Gavanelli and Rossi Francesca. Constraint logic programming. *25 years of logic programming*, pages 64–86, 2010.

[15] Joxan Jaffar and Michael J. Maher. Constraint logic programming : A survey. *J. logic programming (20)*, pages 503–581, 1994.

[16] Tim Lindholm, Franck Yellin, Gilad Bracha, and Alex Buckley. *The Java VirtualMachine Specification (Java SE 8 Edition)*. 2015.

[17] Kim Marriott and Peter J. Stuckey. Programming with constraints : An introduction. 1998.

[18] B. D. Martino and G. Iannello. Pap recognizer: A tool for automatic recognition of parallelizable patterns. *4th International Workshop on Program Comprehension (WPC)*, page 164, 1996.

[19] George C Necula, Scott McPeak, S. P. Rahul, and estley Weimer. Cil : Intermediate language and tools for analysis and transformation of c programs. 2002.

[20] Etienne Payet and Fausto Spoto. Experiments with non-termination analysis for java bytecode. 2009.

[21] A Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. pages 43–55.

[22] Yunhe Shi, David Gregg, Andrew Beatty, and Martin Anton Ertl. Virtual machine showndown : Stack versus registers. 2005.

[23] A. Taherkhani and L. Malmi. Schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining*, pages 69–101, 2013.

[24] R. Vallée-Rai, P. CO, E. Gagnon, L. Hendren, P. LAM, and V. Sundaresan. Scoot - a java bytecode optimization framework. *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (IBM Press)*, page 13, 1999.

[25] Wim Vanhoof, Étienne Payet, and Frédéric Mesnard. Towards a framework for algorithm recognition in binary code. 2016.

[26] L.M. Wills. Flexible control for program recognition. *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 134–143, 1993.

[27] Gonzague Yernaux. Horn clauses transformation framework for algorithm comparison. 2017.

[28] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobileapplication repackaging detection. *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 25–36, 2014.

[29] F Zhang, Y. C. Jhi, D. Wu, P. Liu, and Zhu. A. A first step towards algorithm plagiarism detection. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 111–121, 2012.