HAL
open science

Towards a Framework for Algorithm Recognition in Binary Code

Frédéric Mesnard, Etienne Payet, Wim Vanhoof

To cite this version:

Frédéric Mesnard, Etienne Payet, Wim Vanhoof. Towards a Framework for Algorithm Recognition in Binary Code. 18th International Symposium on Principles and Practice of Declarative Programming (PPDP), Sep 2016, Edinburgh, United Kingdom. pp.202–213. hal-01451686

HAL Id: hal-01451686

https://hal.univ-reunion.fr/hal-01451686

Submitted on 5 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards a Framework for Algorithm Recognition in Binary Code

Frédéric Mesnard
LIM, université de la Réunion, France
frederic.mesnard@univ-reunion.fr

Étienne Payet
LIM, université de la Réunion, France
etienne.payet@univ-reunion.fr

Wim Vanhoof
Faculté d'informatique, université de Namur, Belgium
wim.vanhoof@unamur.be

Abstract

Algorithm recognition, which is the problem of verifying whether a program implements a given algorithm, is an important topic in program analysis. We propose an approach for algorithm recognition in binary code. For this paper, we have chosen the Dalvik Virtual Machine (DVM) bytecode. Given an algorithm A that is compiled into a DVM method M_0, and a DVM program P that includes a series of methods $\{M_1, \ldots, M_n\}$, the approach is able to identify those blocks M_i from P that essentially implement the algorithm A. The technique we propose first translates binary code into Horn clauses. Then we consider programs as implementing the same algorithm if their Horn clause representations can be reduced to a single common set of Horn clauses by means of a sequence of transformations.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal methods.

Keywords Algorithm recognition.

1. Introduction

Algorithm recognition – which we could intuitively define as verifying whether a program implements a given algorithm (Wills 1993) – is an important topic in program analysis. Applications are diverse and range from program comprehension (Storey 2005) over plagiarism detection (Zhang et al. 2012) and malware detection (Zhang et al. 2014) to advanced analyses and optimisations such as the automatic detection of the best parallelisation strategy for a given code fragment (Martino and Iannello 1996).

How to identify whether a given piece of code implements a particular algorithm is a challenging question that can be studied from different angles. While it can generally be assumed that the algorithm to be recognised is known in the form of source code, whether this is the case for the codebase in which the search has to take place depends on the desired application. Consider for example a system for assessing whether students in a programming course have correctly implemented a particular sorting algorithm (Taherkhani and Malmi 2013); in that case one can safely assume the student's source code to be available and one can resort to techniques for assessing the similarity between two algorithms represented in source code. However, as an alternative example, consider a company that wishes to verify whether a competitor's software program uses a (proprietary) algorithm. In this case, the competitor's software might be written in a different programming language and is presumably only available as binary code. Consequently, the verification must be performed between some model of the algorithm and that of the binary code under scrutiny (Zhang et al. 2012).

In this work, we propose an approach for algorithm recognition in binary code, namely Google's Dalvik Virtual Machine (DVM) bytecode, making the approach directly applicable to (compiled) Android programs. A direct application of our approach resides in algorithm plagiarism detection, a topic that has – in contrast with so-called software plagiarism – received relatively little attention (Zhang et al. 2012). While software plagiarism is mainly about unlawfully reusing existing source code or libraries, algorithm plagiarism is more about unlawfully copying the ideas behind how a certain computation is done. Let us consider computing the maximum of three integer values as an almost ridiculously simple example. Figure 1 shows two different Java source methods that perform this computation. While the code of the two methods is quite different (and will as such presumably not be recognised as software plagiarism or software clones), the algorithm that is implemented by both methods is essentially the same. Indeed, in both cases one compares `z` with the maximum of `x` and `y`. In the middle method code this is more explicit by the presence of the auxiliary variable, but the left-hand side method does essentially the same. While the example is, admittedly, too simple to speak of algorithm plagiarism, for several other applications (for example in the context of program understanding or program refactoring) it might nevertheless be desirable to recognize that the algorithm underlying both methods is the same.

While the notion of two algorithms being the same is not easily defined, and the mere existence of equivalence classes of programs implementing the same algorithm is even subject to debate (Blass et al. 2009), a pragmatic approach that is often taken in algorithm recognition (Metzger and Wen 2000) is to consider programs as implementing the same algorithm if they can be reduced to one another by means of a sequence of syntactical transformations. While our approach is based on program transformation and targets algorithm recognition in binary code, one of its cornerstones is the fact that we use Horn clauses to represent a model of the algorithm as well as of the compiled code under scrutiny. This has several advantages:

- Horn clauses are a suitable abstraction that is in between binary code and a more high-level programming language. This is in line with recent work (Gange et al. 2015) in which the use of Horn Clauses as a *universal* intermediate language has been advocated. In that work the general idea is to compile a program (written in an arbitrary language) first into Horn clauses, then

[Copyright notice will appear here once 'preprint' option is removed.]

```
int max(int x, int y, int z) {
  if ((x>=y) && (x>=z))
    return x;
  else if ((y>=x) && (y>=z))
    return y;
  else
    return z;
}
```

```
int max(int x, int y, int z) {
  int max_xy;
  if (x>=y)
    max_xy = x;
  else
    max_xy = y;
  if (max_xy>=z)
    return max_xy;
  else
    return z;
}
```

```
.method public max(III)I
    .registers 5
0: if-lt v2, v3, 4
1: move v0, v2
2: if-lt v0, v4, 6
3: return v0
4: move v0, v3
5: goto 2
6: move v0, v4
7: goto 3
.end method
```

Figure 1. Two different methods computing the maximum of three values, and the Dalvik code corresponding to the middle method.

analysing and optimizing this representation and, if needed, compile the result into executable code. In our work we take this idea the other way round and we present a scheme that allows us to decompile Dalvik binary code into Horn clauses.

- Horn clauses are a well-known formalism and have already been proven to be suitable for a lot of different program analyses. In this work, we resort to well-known transformations of Horn clauses in order to formally establish what it means for two algorithms, represented by a set of Horn clauses, to be the same.
- Even if we specifically target Android binary code in this work, the fact that our framework for algorithm recognition is expressed at the Horn clause level makes it a very general approach that can be readily ported to a multitude of programming languages and systems.

In our approach, we assume that we have an algorithm of interest A, for example the Java code at the left of Figure 1, and a series of DVM bytecode fragments $\{M_1, \ldots, M_n\}$ (a code fragment typically being a method but in principle any block having a clearly defined entry and exit point will do) where we suspect one of these bytecode fragments to be implementing the algorithm A. In the context of our example, one of the bytecode fragments could be the DVM code, *right*-hand side of Figure 1, resulting from compiling the method in the middle of Figure 1. The algorithm recognition process that we propose is schematically represented in Figure 2.

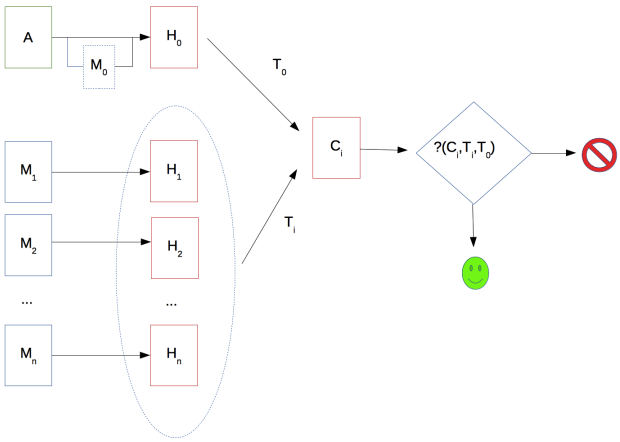

Figure 2. The algorithm recognition process.

The algorithm A is transformed (either directly or via its DVM implementation M_0) into a Horn clause representation H_0. Each bytecode fragment M_i is likewise decompiled into its Horn clause equivalent H_i. Subsequently, for each fragment M_i ($1 \leq i \leq n$), we try to establish two transformation sequences, T_0 and T_i, that transform respectively H_0 and H_i into a single Horn clause representation C_i that has, at least partially, the same semantics as H_0 and H_i (and thus as A and M_i). If such a common representation C_i can be found, we conclude that M_i implements the same algorithm as does A, at least with respect to the part of the semantics that has been preserved by the transformations. Note that termination of the process depends on the algorithm that is used when searching for a common representation. Since, as we will explain further, the search is based on repeatedly applying a set of given program transformations, the process can be made terminating at the cost of not necessarily finding such a common representation. In that sense, the outcome of the process is either that the code fragments are recognised as implementing the same algorithm, or that the search is inconclusive. Returning to our running example, Figure 3 represents H_0 and H_1 that is, the Horn-clause equivalent of, respectively, the left-hand and right-hand sides of Figure 1 after some initial semantics-preserving transformations (basically unfolding all non-recursive predicates). The precise description of our Horn-clause language will be defined in Section 2, but let us note that it is essentially a constraint logic language. The last two arguments of each predicate correspond to the Dalvik memory before and after the call, modeled as a zero-based array where its first element is the return value of the corresponding method.

Even if the predicates $p_0/7$ and $p_1/8$ depicted in the figure are different, it is not hard to see that they can be transformed into a common definition. Indeed, unfolding $V_4 \geq V_5$ in the second clause of p_1 into $(V_4 > V_5) \vee (V_4 = V_5)$, unfolding likewise $V_3 \geq V_4$ in the third clause, and removing from the definition the three first (unused) arguments, one obtains

$$
\begin{array}{l}
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_3 \geq V_4, V_3 \geq V_5, A' = A\{0 \leftarrow V_3\}\}. \\
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_4 > V_5, V_3 < V_4, A' = A\{0 \leftarrow V_4\}\}. \\
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_4 = V_5, V_3 < V_4, A' = A\{0 \leftarrow V_4\}\}. \\
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_3 = V_4, V_3 < V_5, A' = A\{0 \leftarrow V_5\}\}. \\
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_3 > V_4, V_3 < V_5, A' = A\{0 \leftarrow V_5\}\}. \\
p_1(V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_4 < V_5, V_3 < V_4, A' = A\{0 \leftarrow V_5\}\}.
\end{array}
$$

It can be easily observed that the same code (apart from a renaming of the arguments and a reordering of the constraints) can be obtained from the definition of p_0 by unfolding the atom $V_3 \geq V_4$ in the second clause and eliminating from the definition the first two (unused) arguments. Since unfolding and removal of unused arguments preserves the semantics, we can conclude that both fragments implement the same algorithm.

The remainder of the paper is organised as follows. Section 2 describes our decompilation schema from Dalvik code to Horn

$$
\begin{array}{l}
p_0(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_2 \geq V_4, V_2 \geq V_3, A' = A\{0 \leftarrow V_2\}\}. \\
p_0(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_2 < V_3, V_3 \geq V_4, A' = A\{0 \leftarrow V_3\}\}. \\
p_0(V_0, V_1, V_2, V_2, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_2 < V_4, A' = A\{0 \leftarrow V_4\}\}. \\
p_0(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_2 < V_4, V_3 < V_2, A' = A\{0 \leftarrow V_4\}\}. \\
p_0(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_2 < V_3, V_3 < V_4, A' = A\{0 \leftarrow V_4\}\}.
\end{array}
$$

$$
\begin{array}{l}
p_1(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_3 \geq V_4, V_3 \geq V_5, A' = A\{0 \leftarrow V_3\}\}. \\
p_1(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_4 \geq V_5, V_3 < V_4, A' = A\{0 \leftarrow V_4\}\}. \\
p_1(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_3 \geq V_4, V_3 < V_5, A' = A\{0 \leftarrow V_5\}\}. \\
p_1(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \\
\quad \{V_4 < V_5, V_3 < V_4, A' = A\{0 \leftarrow V_5\}\}.
\end{array}
$$

Figure 3. Horn clause representation of the methods of Figure 1, obtained by decompilation.

clauses. Section 3 provides the formal framework enabling the algorithm recognition process as described, and the formal definition of what it means, within this framework, when two program fragments implement the same algorithm. Section 4 summarizes related work and Section 5 concludes with a discussion of future work.

2. From Binary Code to Clauses

Android programs are written in Java. They are compiled to the Google's Dalvik Virtual Machine (DVM) bytecode format before installation on a device. We assume the reader familiar with the basic concepts of object-oriented programming and the Android platform (Android developers). Here, we briefly describe the DVM, see (ART and Dalvik) for a complete presentation.

Unlike the Java Virtual Machine (Lindholm and Yellin 1999) which is stack-based, the DVM is register-based. It runs a Dalvik bytecode program by keeping an activation stack of *frames*. Each frame is created by a method call, survives until the end of the call and uses its own registers. An invoked method cannot affect the registers in the frame of the invoking method. Any call to a same method always produces a frame with a same number of registers. For each method ω, this number is statically known. We denote it as $reg(\omega)$ and we refer to it as the *number of registers used by* ω.

The memory of the system contains objects, connected through pointers. To simplify the presentation, we do not consider array nor interface types and only allow integers as values of basic types.

Definition 1. *The set of* values *is* $\mathbb{Z} \cup \mathbb{L}$*, where* $\mathbb{Z}$ *is the set of integers and* $\mathbb{L}$ *is the set of* memory locations*. A* frame *of the DVM is a pair* $\langle v, \mu\rangle$ *where* v *is a sequence of values, called* registers*, numbered from* 0 *upwards and* μ *is a* memory*, or* heap*, that maps locations into* objects*. An object is a pair* $\langle \kappa, f\rangle$ *where* κ *is a class identifier i.e., an index (integer) in the class definition list of the program, and* f *is a sequence of values, called* fields*, numbered from* 0 *upwards; we say that it* belongs to *class* κ *or that it is* an instance of *class* κ *or* has class κ*. We let* $o.\kappa$ *and* $o.f$ *respectively denote the class and fields of an object* o*. We require that there are no dangling pointers i.e.,* $v \cap \mathbb{L} \subseteq dom(\mu)$ *and* $\mu(\ell).f \cap \mathbb{L} \subseteq dom(\mu)$ *for every* $\ell \in dom(\mu)$*. The set of all classes is denoted by* $\mathbb{K}$ *and it is partially ordered by the* subclass *relation (we consider that a class is a subclass of itself).*

The Dalvik bytecode is strongly typed. Each value has a type and registers are statically typed.

Definition 2. *The set of* types *of our simplified DVM is* $\mathbb{T} = \mathbb{K} \cup \{\texttt{int}, \texttt{void}\}$*. The* `void` *type can only be used as the return type of methods. A method signature is denoted by* $\kappa.m(t_1, \ldots, t_p)t$ *standing for a method named* m*, defined in class* κ*, expecting* p *explicit parameters of type, respectively,* $t_1, \ldots, t_p$ *and returning a value of type* t*, or returning no value when* $t = \texttt{void}$*.*

A non-static method $\kappa.m(t_1, \ldots, t_p)t$ also has an *implicit* parameter of type κ called `this` in the code of the method. So the actual number of parameters is $p + 1$. We do not distinguish between methods and constructors. A constructor is just a method named `<init>` and returning `void`. By a *void method* (resp. *non-void method*) we mean a method whose return type is `void` (resp. is not `void`). We do not consider static fields and methods. The extension of our definitions to them is not difficult.

Dalvik bytecode instructions work over frames and their execution affects the registers or the memory in the frames. Many are similar or only differ in the type or size of their operands. So we concentrate on a restricted set which exemplifies the operations that the DVM performs.

- $const\ d, c$ Writes constant c into register d.
- $move\ d, s$ Writes the value of register s into register d.
- $add\ d, s, c$ Writes the sum of the value of register s and constant c into register d.
- $if\text{-}lt\ i, j, q$ If the value of register i is less than the value of register j then jumps to program point q, otherwise executes the immediately following instruction.
- $goto\ q$ Jumps to program point q.
- $invoke\ S, \kappa.m(t_1, \ldots, t_p)t$ where $S = s_0, s_1, \ldots, s_p$ is a sequence of register indexes. The value v_{s_0} of register s_0, $\ldots$, v_{s_p} of register s_p are the *actual parameters* of the call. Value v_{s_0} is called *receiver* of the call and must be 0 (the equivalent of `null` in Java) or the memory location of an object o. In the former case, the computation stops with an exception. Otherwise, a lookup procedure is started from the class of o upwards along the superclass chain, looking for a method called m expecting p *formal parameters* of type $t_1, \ldots, t_p$, respectively, and returning a value of type t. It is guaranteed that such a method is found in a subclass of κ. That method is run from a new frame where the last $p + 1$ registers are bound to $v_{s_0}, v_{s_1}, \ldots, v_{s_p}$, respectively, and the other ones to 0.
- $return$ Returns from a void method.
- $return\ s$ Returns from a non-void method with the value of register s as result.
- $move\text{-}result\ d$ Writes the result of the most recent called method into register d. This instruction must immediately follow an $invoke$ instruction.
- $new\text{-}instance\ d, \kappa$ Writes the memory location of a new, properly initialised, object of class κ into register d.
- $iget\ d, i, j$ (resp. $iput\ s, i, j$) The value v_i of register i must be 0 or the memory location of an object o. If v_i is 0, the computation stops with an exception. Otherwise, the value of field j of o is written into register d (resp. the value of register s is written into the field j of o).

We suppose that the Dalvik program P under consideration consists of these instructions and that it is well-formed. For instance, each $move\text{-}result$ immediately follows an instruction of the form

$$q : const\ d, c \mapsto \big\{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = c\} \cup id_{-d},\ p_{q+1}(\tilde{V}', M, M')\big\}$$

$$q : move\ d, s \mapsto \big\{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = V_s\} \cup id_{-d},\ p_{q+1}(\tilde{V}', M, M')\big\}$$

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

$$q : goto\ q' \mapsto \big\{p_q(\tilde{V}, M, M') \leftarrow id,\ p_{q'}(\tilde{V}', M, M')\big\}$$

$$q : if\text{-}lt\ i, j, q' \mapsto \left\{ \begin{array}{l} p_q(\tilde{V}, M, M') \leftarrow \{V_i < V_j\} \cup id,\ p_{q'}(\tilde{V}', M, M') \\ p_q(\tilde{V}, M, M') \leftarrow \{V_i \geq V_j\} \cup id,\ p_{q+1}(\tilde{V}', M, M') \end{array} \right\}$$

Figure 4. Compilation of simple instructions.

$$\frac{ins = invoke\ s_0, \ldots, s_p, \kappa.m(t_1, \ldots, t_p)t}{q : ins \mapsto \left\{ \begin{array}{l|l} p_q(\tilde{V}, \langle A, I\rangle, M') \leftarrow \big\{V_{s_0} > 0,\ \kappa' = A[V_{s_0}, 0]\big\} \cup id, & \kappa' \text{ is a subclass of } \kappa \\ \qquad p_{q_{\omega'}}(\tilde{W}, \langle A, I\rangle, M_1), & \omega' = lookup(\kappa.m(t_1, \ldots, t_p)t, \kappa') \\ \qquad p_{q+1}(\tilde{V}', M_1, M') & \tilde{W} = 0, \ldots, 0, V_{s_0}, \ldots, V_{s_p} \text{ with } |\tilde{W}| = reg(\omega') \end{array} \right\}}$$

$$q : move\text{-}result\ d \mapsto \big\{p_q(\tilde{V}, \langle A, I\rangle, M') \leftarrow \big\{V'_d = A[0]\big\} \cup id_{-d},\ p_{q+1}(\tilde{V}', \langle A, I\rangle, M')\big\}$$

$$q : return\ s \mapsto \big\{p_q(\tilde{V}, \langle A, I\rangle, \langle A', I'\rangle) \leftarrow \{A' = A\{0 \leftarrow V_s\},\ I' = I\}\big\}$$

$$q : return \mapsto \big\{p_q(\tilde{V}, \langle A, I\rangle, \langle A', I'\rangle) \leftarrow \{A' = A,\ I' = I\}\big\}$$

Figure 5. Compilation of instructions related to method calls.

$$\frac{ins = new\text{-}instance\ d, \kappa \quad \text{and} \quad \text{objects of class } \kappa \text{ have } n \text{ fields}}{q : ins \mapsto \left\{ \begin{array}{r} p_q(\tilde{V}, \langle A, I\rangle, M') \leftarrow \big\{O[0] = \kappa,\ O[1] = 0,\ \ldots,\ O[n] = 0,\ A_1 = A\{I \leftarrow O\}, \\ V'_d = I,\ I_1 = I + 1\big\} \cup id_{-d},\ p_{q+1}(\tilde{V}', \langle A_1, I_1\rangle, M') \end{array} \right\}}$$

$$q : iget\ d, i, j \mapsto \big\{p_q(\tilde{V}, \langle A, I\rangle, M') \leftarrow \big\{V_i > 0,\ V'_d = A[V_i, j + 1]\big\} \cup id_{-d},\ p_{q+1}(\tilde{V}', \langle A, I\rangle, M')\big\}$$

$$q : iput\ s, i, j \mapsto \left\{ \begin{array}{r} p_q(\tilde{V}, \langle A, I\rangle, M') \leftarrow \big\{V_i > 0,\ O = A[V_i],\ O_1 = O\{j + 1 \leftarrow V_s\}, \\ A_1 = A\{V_i \leftarrow O_1\}\big\} \cup id,\ p_{q+1}(\tilde{V}', \langle A_1, I\rangle, M') \end{array} \right\}$$

Figure 6. Compilation of memory-related instructions.

$invoke\ S, \kappa.m(t_1, \ldots, t_p)t$ where $t \neq$ `void`. As for Java bytecode, real Dalvik bytecode must pass a verification check before being run on a device. Program points of P are denoted by $q, q', \ldots$ and we let $q + 1$ denote the program point immediately following q.

Our rules for compiling the instructions of P into clauses are given in Fig. 4–6. Some of them have already been presented in (Payet and Mesnard 2014). They have the form $q : ins \mapsto E$ where E is the set of clauses resulting from the compilation of instruction ins occurring at q. We sometimes write $\frac{\psi}{q : ins \mapsto E}$ meaning that the rule only applies when condition ψ holds.

We assign a predicate symbol p_q to each program point q of P. The arity of p_q is $r + 2$ where $r = reg(\omega)$ and ω is the method where q occurs. We assign the following meaning to the parameters of p_q. In an atom of the form

$$p_q(V_0, \ldots, V_{r-1}, M, M')$$

the first $r + 1$ parameters correspond to the state of the current frame just *before* executing the instruction at q: $V_0, \ldots, V_{r-1}$ are the values of the registers and M is the memory. The last parameter M' is the memory upon termination of ω. It is used for handling method calls; it is instantiated in the clauses generated for $return$ and its value is used in the clauses generated for $invoke$ (Fig. 5).

We generate clauses with constraints on integer and array terms. Our constraint theory combines the theory of integers with that of arrays defined in (Bradley et al. 2006). We borrow the following notations from (Bradley et al. 2006): the read $a[i]$ returns the value stored at position i of the array a and the write $a\{i \leftarrow e\}$ is a modified so that position i has value e; for multidimensional arrays, $a[i] \cdots [j]$ is abbreviated with $a[i, \ldots, j]$. We model a memory as a pair $\langle a, i\rangle$ where a is an array, called *memory content*, indexed from 0 upwards and i is the index where the next insertion in a will take place. We do not model garbage collection and assume that the memory is unbounded. Memory locations are indexes into a; they start at 1 and 0 corresponds to the `null` value. We model an object as an array of integers $[\kappa, x_0, \ldots, x_n]$, indexed from 0 upwards, where κ is the class identifier and $x_0, \ldots, x_n$ are the current values of the fields. Note that the value x_j of a field j is located at index $j + 1$. A memory content has the form $[x, o_0, \ldots, o_n]$ where x, an integer, is the result of the most recent called, non-void, method and $o_0, \ldots, o_n$ are objects.

Definition 3. *Our CLP domain of computation (values interpreting constraints) is $\mathcal{D} = \mathbb{Z} \cup \mathcal{O} \cup \mathcal{A}$ where $\mathcal{O}$ is the set of objects and $\mathcal{A}$ is the set of memory contents.*

Each rule of Fig. 4–6 considers an instruction ins occurring at a program point q. Uppercase letters denote variables. We let $\tilde{V} = V_0, \ldots, V_{r-1}$ and $\tilde{V}' = V'_0, \ldots, V'_{r-1}$ be sequences of distinct variables where r is the number of registers used by the method where ins occurs. For each $i \in [0, r-1]$, variable V_i (resp. V'_i) denotes the value of register i before (resp. after) executing ins. We use variables $M, M', \ldots$ or pairs of variables $\langle A, I\rangle, \langle A', I'\rangle, \ldots$ for denoting the memory. We let id denote the sequence $(V'_0 = V_0, \ldots, V'_{r-1} = V_{r-1})$ and id_{-i} (where $i \in [0, r-1]$) the sequence $(V'_0 = V_0, \ldots, V'_{i-1} = V_{i-1}, V'_{i+1} = V_{i+1} \ldots, V'_{r-1} = V_{r-1})$. By $|\tilde{W}|$ we mean the length of sequence $\tilde{W}$. For any method $\omega = \kappa.m(t_1, \ldots, t_p)t$, q_ω is the program point where ω starts. Moreover, for any subclass κ' of κ, we let $lookup(\omega, \kappa')$ denote the closest method in the superclass chain of κ' that has name m, expects p formal parameters of type $t_1, \ldots, t_p$, respectively, and returns a value of type t. As κ' is a subclass of κ and $\omega = \kappa.m(t_1, \ldots, t_p)t$, it is guaranteed that such a method exists.

Some compilation rules are rather straightforward. For instance, $const\ d, c$ writes constant c into register d, so in Fig. 4 the output register variable V'_d is set to c while the other register variables remain unchanged (modelled with id_{-d}). Rules for $move$, add and $goto$ are similar. The rule for $if\text{-}lt\ i, j, q'$ generates two clauses expressing that when the test is true execution jumps to program point q' otherwise it jumps to the next instruction *i.e.,* to program point $q + 1$.

In Fig. 5 we consider method calls. The rule for instruction $invoke\ s_0, \ldots, s_p, \kappa.m(t_1, \ldots, t_p)t$ works as follows. We impose that V_{s_0} (the receiver of the call) is a non-`null` location (*i.e.,* $V_{s_0} > 0$). Therefore, if $V_{s_0} \leq 0$, the execution of the generated CLP program fails, as the original Dalvik program. Moreover, we statically express the Dalvik dynamic lookup of the method to invoke. The method that will effectively be invoked at runtime is necessarily defined in a subclass of κ. Hence, for each subclass κ' of κ, we generate a clause in which we specify that if the class of object $A[V_{s_0}]$ is κ' (*i.e.,* $\kappa' = A[V_{s_0}, 0]$), then we invoke the closest method ω' in the superclass chain of κ' that has name m, expects p formal parameters of type $t_1, \ldots, t_p$ and returns a value of type t. This invocation is modelled by a call to $p_{q_{\omega'}}$ with a set of registers initialised as needed *i.e.,* the arguments of the call in the last registers and the other registers set to 0. This call potentially modifies the memory, which yields the new memory M_1. Finally, when the execution of the invoked method terminates, we transfer control to the program point following the invocation point. This is modelled by a call to p_{q+1} with input memory M_1. The rule for $move\text{-}result$ produces a clause that writes the value returned by the most recent called method (*i.e.,* $A[0]$) into register d. Instructions $return$ and $return\ s$ stop the execution of the current method, hence the body of the generated clauses contains no call. Moreover, the clause generated for $return\ s$ writes the value of register s at index 0 of the memory content *i.e.,* it sets the value returned by the most recent called method to that of register s.

Example 4. *Let us consider the program in Fig. 7. In method* `f`*, the variable* `a` *has type* `A`*, hence it may store a reference to an instance of any subclass of* `A`*. Therefore, when compiling the call* `a.mm()` *to clauses, we have to consider all the subclasses of* `A` *i.e.,* `A` *itself and* `B`*. For each subclass, we determine the method that would be invoked if a reference to an instance of it was stored in* `a`*. If* `a` *stores a reference to an instance of* `A`*, then the method* `mm` *of* `A` *is invoked. If* `a` *stores a reference to an instance of* `B`*, then the method* `mm` *of* `B` *is invoked. So, the instruction at line 14 is compiled to:*

$$\begin{array}{l} p_{14}(V_0, V_1, V_2, V_3, \langle A, I\rangle, M') \leftarrow \{V_0 > 0,\ \texttt{A} = A[V_0, 0], \\ \quad V'_0 = V_0,\ V'_1 = V_1,\ V'_2 = V_2,\ V'_3 = V_3\}, \\ \quad p_4(0, V_0, \langle A, I\rangle, M_1),\ p_{15}(V'_0, V'_1, V'_2, V'_3, M_1, M'). \end{array}$$

$$\begin{array}{l} p_{14}(V_0, V_1, V_2, V_3, \langle A, I\rangle, M') \leftarrow \{V_0 > 0,\ \texttt{B} = A[V_0, 0], \\ \quad V'_0 = V_0,\ V'_1 = V_1,\ V'_2 = V_2,\ V'_3 = V_3\}, \\ \quad p_7(0, V_0, \langle A, I\rangle, M_1),\ p_{15}(V'_0, V'_1, V'_2, V'_3, M_1, M'). \end{array}$$

Now, consider the call `a.m()`*. As there is no implementation of* `m` *in* `B`*, the method* `m` *of* `A` *is always invoked here. So, the instruction at line 13 is compiled to:*

$$\begin{array}{l} p_{13}(V_0, V_1, V_2, V_3, \langle A, I\rangle, M') \leftarrow \{V_0 > 0,\ \texttt{A} = A[V_0, 0], \\ \quad V'_0 = V_0,\ V'_1 = V_1,\ V'_2 = V_2,\ V'_3 = V_3\}, \\ \quad p_0(0, V_0, \langle A, I\rangle, M_1),\ p_{14}(V'_0, V'_1, V'_2, V'_3, M_1, M'). \end{array}$$

$$\begin{array}{l} p_{13}(V_0, V_1, V_2, V_3, \langle A, I\rangle, M') \leftarrow \{V_0 > 0,\ \texttt{B} = A[V_0, 0], \\ \quad V'_0 = V_0,\ V'_1 = V_1,\ V'_2 = V_2,\ V'_3 = V_3\}, \\ \quad p_0(0, V_0, \langle A, I\rangle, M_1),\ p_{14}(V'_0, V'_1, V'_2, V'_3, M_1, M'). \end{array}$$

□

Finally, in Fig. 6 we consider the memory-related instructions. The rule for $new\text{-}instance\ d, \kappa$ builds a new object O which is properly initialised, stores O in memory at location I (*i.e.,* $A_1 = A\{I \leftarrow O\}$), writes the location of O into register d (*i.e.,* $V'_d = I$) and sets the next insertion index to $I + 1$ (*i.e.,* $I_1 = I + 1$). In the clauses generated for $iget\ d, i, j$ and $iput\ s, i, j$ we specify that the value of register i is a non-`null` location (*i.e.,* $V_i > 0$). Therefore, if $V_i \leq 0$, the execution of the generated CLP program fails, as the original Dalvik program. The rule for $iget$ considers the object whose location is in register i (*i.e.,* $A[V_i]$) and writes the value of field j of this object into register d *i.e.,* $V'_d = A[V_i, j + 1]$. We write $A[V_i, j + 1]$ for accessing field j because the array $A[V_i]$ starts with the class identifier of the object, hence field j is located at index $j + 1$. The rule for $iput$ considers the object O whose location is in register i (*i.e.,* $O = A[V_i]$) and replaces it in memory with a new object O_1 (*i.e.,* $A_1 = A\{V_i \leftarrow O_1\}$) which is the same as O up to the value of field j, which is set from the value of register s (*i.e.,* $O_1 = O\{j + 1 \leftarrow V_s\}$).

Example 5. *In Fig. 7, objects of class* `A` *only have one field (*`n`*). Hence, the* $new\text{-}instance$ *instruction at line 11 of the Dalvik program is compiled into the clause:*

$$\begin{array}{l} p_{11}(V_0, V_1, V_2, V_3, \langle A, I\rangle, M') \leftarrow \{O[0] = \texttt{A}, O[1] = 0, \\ \quad A_1 = A\{I \leftarrow O\},\ V'_0 = I,\ I_1 = I + 1, \\ \quad V'_1 = V_1,\ V'_2 = V_2,\ V'_3 = V_3\}, \\ \quad p_{12}(V'_0, V'_1, V'_2, V'_3, \langle A_1, I_1\rangle, M'). \end{array}$$

□

3. Algorithm Recognition in Horn Clauses

In this section we will define what it means for two (decompiled) programs to be algorithmically equivalent. Proving algorithmic equivalence boils down (Metzger and Wen 2000) to proving semantic equivalence on the one hand (the two program fragments compute the same results and/or exhibit the same behaviour) and some sort of structural equivalence on the other hand, meaning that the two program fragments are similar in their (algorithmic) structure. We will first define what semantic equivalence means in our setting, and we will focus on structural equivalence afterwards.

3.1 Semantic equivalence of code fragments

A decompiled Dalvik program is represented by a set of predicate definitions of the form $H \leftarrow \{C\}, \overline{B}$ where H is the head atom, C is a set of constraints and $\overline{B}$ a conjunction of atoms (most notably

```
public class A {
    private int n = 5;

    public void m() { n++; }

    public int mm() { return n + 2; }
}

public class B extends A {
    public int mm() { return 10; }
}

public class MyActivity extends Activity {
    ...
    public int f(int i) {
        A a = (i >= 1 ? new A() : new B());
        a.m();
        return a.mm();
    }
    ...
}
```

```
.method public m()V
    .registers 2
0:  iget v0, v1, 0
1:  add v0, v0, 1
2:  iput v0, v1, 0
3:  return
.end method

.method public mm()I
    .registers 2
7:  const v0, 10
8:  return v0
.end method

.method public f(I)I
    .registers 4
 9: const v1, 1
10: if-lt v3, v1, 17
11: new-instance v0, A
12: invoke v0, A.<init>()V
13: invoke v0, A.m()V
14: invoke v0, A.mm()I
15: move-result v1
16: return v1
17: new-instance v0, B
18: invoke v0, B.<init>()V
19: goto 13
.end method

.method public mm()I
    .registers 2
4:  iget v0, v1, 0
5:  add v0, v0, 2
6:  return v0
.end method
```

Figure 7. A part of an Android program with simple instructions, method calls and memory accesses (Java code on the left and corresponding Dalvik bytecode on the right). The Dalvik bytecode is written in a simplified Smali's syntax (Smali assembler for the dex format). The `.registers` directive at the beginning of each method indicates the number of registers used by the method. Moreover, `v0` and `v1` denote registers 0 and 1 respectively and `V` (resp. `I`) denotes the type `void` (resp. `int`). At lines 0, 2 and 4, value 0 corresponds to the field `n` defined in `A`.

calls to other predicates). Although in our case $\overline{B}$ is a conjunction of 0, 1 or 2 atoms, the results of this section remain valid for the general case where $\overline{B}$ is any conjunction of atoms. While different semantics have been defined for CLP programs (Jaffar and Lassez 1987; Jaffar et al. 1998), given the simplicity of the clauses that are generated by the compilation scheme, we can stick to the basic computed answer semantics as it is known from Prolog (Lloyd 1987). As usual, we will use Greek letters to denote substitutions (mapping from variables to data terms). A call to one of the generated predicates having as head $p_q(V_0, \ldots, V_{n-1}, M, M')$ is thus represented by an atom $p_q(V_0, \ldots, V_{n-1}, M, M')\theta$ with θ a substitution. Given the nature of the generated clauses, θ will map the predicate's input arguments ($V_0, \ldots, V_n$ and M) to ground terms and, given the deterministic behaviour of the generated clauses, the call will result in a single answer substitution θ' mapping the predicate's single output argument M' to a ground value.

While the simple computed answer semantics is sufficient to model the behaviour of our generated CLP programs, we need a somewhat more sophisticated measure in order to compare the values that are manipulated by these programs. Since objects are essentially represented by locations into a memory, we will in the following often refer to a value-memory pair, represented by (v, μ), where v is a value from $\mathbb{Z} \cup \mathbb{L}$ and μ is the memory potentially referred to by v. In order to compare value-memory pairs while making abstraction of the actual locations, we define the notion of value equivalence as follows:

Definition 6. *Two value-memory pairs (v, μ) (v', μ') are* value equivalent*, which we denote by $(v, \mu) \approx (v', \mu')$, when the following conditions are met:*

$$\begin{cases} v = v' & \text{if } v \text{ and } v' \text{ are basic values from } \mathbb{Z} \\ true & \text{if } v \text{ and } v' \text{ are locations, } \mu = \langle a, i \rangle \text{ and} \\ & \mu' = \langle a', i' \rangle, a[v] = \langle \kappa, x_0, \ldots, x_n \rangle \\ & \text{and } a'[v'] = \langle \kappa, x'_0, \ldots, x'_n \rangle, \\ & \text{and } \forall j : 0 \leq j \leq n : (x_j, \mu) \approx (x'_j, \mu') \\ false & \text{otherwise.} \end{cases}$$

Intuitively, two value-memory pairs are value equivalent if their value parts either represent the same basic (integer) value, or if they both refer to an object (each in their respective memory) belonging to the same class and the corresponding fields are, in turn, value equivalent.

Now, in order to formalise semantic equivalence of decompiled Dalvik programs, let us define what it means for two such programs to compute the same result. Since the CLP predicates we wish to relate result from compiling different sources, they potentially have a different number of arguments (reflecting a different number of used registers) and, even if the predicates basically compute the same results, they may use different registers (and thus argument positions) for storing what may essentially be the same values. The following definition captures what it means for two such predicates to compute the same result. It states that both predicates must have a subsequence of their argument positions (both sequences having the same size but containing possibly different argument positions and not necessarily in the same order) such that when the predicates are invoked with the corresponding arguments initialised with the same values, then each predicate computes the same result. This means that for each pair of arguments representing two corresponding registers, the value-memory pairs referred to by these arguments must be value equivalent both at the moment the predicates are invoked (condition 1 in the definition) and at the moment the predicates return (condition 2 in the definition). Note that while the arguments (being ground values) will not have changed over the execution of the predicate, the memories *will* have: the initial memories represented by $\theta(M)$ and $\sigma(M)$, the final memories by $\theta'(M')$ and $\sigma'(M')$. Finally, the values returned by each of the methods must also be value equivalent (condition 3 in the definition). As for notation, given a sequence R, we denote by R_i the i'th element of R.

Definition 7. *Given CLP programs P_1 and P_2 representing decompiled Dalvik programs, let p_s/n_s and p_q/n_q denote predicates in, respectively, P_1 and P_2 and let R and R' denote sequences of argument positions from respectively $\{1, \ldots n_s\}$ and $\{1, \ldots n_q\}$*

such that $|R| = |R'| = n$. *We say that* (p_s, R) computes in P_1 a subset of (p_q, R') in P_2 *if and only if for each call of the form* $p_s(V_0, \ldots, V_{n_s-3}, M, M')\theta$ *with computed answer substitution* θ', *there also exists a call* $p_q(V_0, \ldots, V_{n_q-3}, M, M')\sigma$ *with computed answer substitution* σ' *such that the following holds for all* $k \in 0 \ldots n-1$:

1. $(\theta(V_{R_k}), \mu_{in}) \approx (\sigma(V_{R'_k}), \mu'_{in})$
2. $(\theta(V_{R_k}), \mu_{out}) \approx (\sigma(V_{R'_k}), \mu'_{out})$
3. $(a[0], \mu_{out}) \approx (a'[0], \mu'_{out})$

where $\mu_{in} = \theta(M)$, $\mu'_{in} = \sigma(M)$, $\mu_{out} = \theta'(M') = \langle a, i\rangle$, *and* $\mu'_{out} = \sigma'(M') = \langle a', i'\rangle$. *Moreover, we say that* (p_s, R) computes the same in P_1 *as does* (p_q, R') *in* P_2 *if and only if* (p_s, R) *computes a subset of* (p_q, R') *and vice versa in their respective programs.*

The above definition allows us to characterise predicates as computing the same results, even if these predicates only *partially* exhibit the same behaviour. Indeed, what matters is that they compute the same return value and update those parts of the memory pointed to by arguments in R, respectively R', in the same way. The parts of the memory that are pointed to by arguments *not* comprised in either R or R' may be updated differently. Note that at the CLP level, a predicate such as p_s or p_q will *always* be called with all but the last argument (representing the output memory) a ground value. Consequently, the computed answer for the call contains a single binding, binding the output memory argument to a ground term. That explains why in condition 2 of the definition we compare $\theta(V_{R_k})$ and $\sigma(V_{R'_k})$ (the argument values at the time of the call) with respect to the output memories (representing the memory states at the time of the return of the call).

Example 8. *Figures 8 and 9 represent two Java methods for computing* x^n *(where* x *and* n *are arguments of the method) and the CLP code that was generated from the corresponding Dalvik code (not displayed). In order to make the CLP code more readable, all intermediate non-recursive predicates have been unfolded.*[1] *If we call the CLP program of Figure 8* P_{exp1} *and that of Figure 9* P_{exp2}, *it is not hard to see that* $(p_{1_0}, \langle 5, 4\rangle)$ *computes the same result in* P_{exp1} *as does* $(p_{2_0}, \langle 6, 5\rangle)$ *in* P_{exp2}. *Indeed, a closer look at the code makes it clear that the arguments* V_4 *and* V_3 *in* p_{1_0} *represent, respectively, the arguments* n *and* x *from the original method whereas in* p_{2_0} *this role is played by the arguments* V_5 *and* V_4. *Moreover, from the original Java code it can be clearly seen that both methods compute the same return value for all possible values of the arguments.*

The following proposition is trivial to prove:

Proposition 9. *The "computes the same" relation defined in Definition 7 is an equivalence relation.*

As Example 8 illustrates, it is essential that we consider a *subset* of arguments when comparing predicates since some of the arguments, introduced by the compilation, are not (or, due to the CLP transformations, no longer) used. This is the case for V_2 in p_{1_0} and both V_2 and V_3 in p_{2_0}. Moreover, selecting a subset of arguments allows us to focus on a (sub)computation of interest when considering semantic equivalence of predicates. As a technical note, the programs P_{exp1} and P_{exp2} both use two helper arguments representing the auxiliary variables k and w (represented in both p_{1_0} and p_{2_0} by the arguments V_0 and V_1); therefore we have also that they both compute the same result with respect to an extended set of arguments, notably we have that $(p_{1_0}, \langle 1, 2, 5, 4\rangle)$ computes the same as $(p_{2_0}, \langle 1, 2, 6, 5\rangle)$.

[1] Unfolding is one of the main transformations that we will consider in our transformation-based approach towards structural equivalence, see Section 3.2.

3.2 Structural equivalence

As is common practice in the literature on algorithm recognition (see e.g. (Metzger and Wen 2000)), we will define algorithmic equivalence using the notion of program transformation, the basic and intuitive idea being that two programs are algorithmically equivalent if one can be transformed into the other by a series of (semantic-preserving) transformations. However, the fact that we use CLP as the representation language for the algorithms allows us to restrict our attention to a limited number of nonetheless powerful transformations (such as slicing and unfolding) whereas more traditional approaches (Metzger and Wen 2000) usually consider a wide variety of more low-level transformations as they are working on the program's source code (such as renaming variables, loop unrolling, array manipulations, etc.)

Let us consider a given set $\mathcal{R}$ of available program transformations. We will in a moment provide examples of concrete transformations that might be considered in this set but define first the notion of an $\mathcal{R}$-transformation sequence as follows, based on (Pettorossi and Proietti 1998).

Definition 10. *Let* $\mathcal{R}$ *be a set of program transformations and* P *a CLP program. Then an* $\mathcal{R}$-transformation sequence *of* P *is a finite sequence of CLP programs, denoted* $\langle P_0, P_1, \ldots, P_n\rangle$, *where* $P_0 = P$ *and* $\forall i\ (0 < i \leq n) : P_i$ *is obtained by the application of one transformation from* $\mathcal{R}$ *on* P_{i-1}.

Given a predefined set of program transformations $\mathcal{R}$ and CLP programs P and Q, we will often use $P \rightsquigarrow^*_{\mathcal{R}} Q$ to represent the fact that there exists an $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$ with $P_0 = P$ and $P_n = Q$. Two transformations of particular interest for our purpose are the *unfolding* (Pettorossi and Proietti 1998) and *slicing* (Szilágyi et al. 2002) transformations.

Definition 11. *Given a program* P, *let* c *be a clause* $A \leftarrow \{C\}, \overline{B}$ *in* P, B_s *one of the atoms in* $\overline{B}$, *and*

$$\begin{cases} H_1 \leftarrow \{C_1\}, \overline{L}_1 \\ \vdots \\ H_n \leftarrow \{C_n\}, \overline{L}_n \end{cases}$$

the (renamed apart) set of clauses in P *such that* $C \wedge C_i \wedge (B_s = H_i)$ *is satisfiable for all* $1 \leq i \leq n$. *Then* unfolding *the atom* B_s *in the clause* c *consists in replacing* c *by the set of clauses*

$$\{A \leftarrow \{C \wedge C_i \wedge B_s = H_i)\}, \overline{B'}_i | 1 \leq i \leq n\}$$

where $\overline{B'}_i$ *represents the conjunction obtained by replacing, in* $\overline{B}$, *the atom* B_s *by the conjunction* $\overline{L}_i$.

Example 12. *Reconsider the program* P_{exp1} *defined at the right hand side in Figure 8. Unfolding the single body atom in the clause defining* p_{1_0} *results in the program defined in part (a) of Figure 11.*

Definition 13. *Given the definition of a predicate* p *in a program* P. *A* slice *of* p *is a predicate* p' *that is obtained from* p *by removing a (possibly empty) subset of its clauses and removing, for each remaining clause, a (possibly empty) subset of the arguments, constraints and atoms therein.*

Example 14. *Consider the program* P_{exp2} *depicted on the right-hand side of Figure 9. We can compute the slice given in Figure 10 by removing the third and fourth arguments (*V_2 *and* V_3*) from each clause and from each call therein, as well as removing the superfluous atom* $V'_2 = 1$ *from the last clause's body.*

While unfolding preserves the computed answer semantics of a program (Pettorossi and Proietti 1998), slicing obviously does not.

```
int exp1(int x, int n) {
   int w = 1;
   int k = n;
   while (k>0) {
      w = w*x;
      k--;
   }
   return w;
}
```

$$p_{1_0}(V_0, V_1, V_2, V_3, V_4, M, M') \leftarrow \{V_1' = 1\}, p_{1_2}(V_4, V_1', V_2, V_3, V_4, M, M').$$

$$p_{1_2}(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 0, A' = A\{0 \leftarrow V_1\}\}.$$

$$p_{1_2}(V_0, V_1, V_2, V_3, V_4, M, M') \leftarrow \{V_0 > 0, V_0' = V_0 - 1, V_1' = V_1 * V_3\}, p_{1_2}(V_0', V_1', V_2, V_3, V_4, M, M').$$

Figure 8. The exponentiation method (version 1) and its translation into CLP, P_{exp1}.

```
int exp2(int x, int n) {
   int w ;
   if (n<=0)
     w = 1;
   else {
     w = x;
     int k = n;
     while (k>1) {
        w = w*x;
        k--;
     }
   }
   return w;
}
```

$$p_{2_0}(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_5 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$

$$p_{2_0}(V_0, V_1, V_2, V_3, V_4, V_5, M, M') \leftarrow \{V_5 > 0\}, p_{2_6}(V_5, V_4, V_2, V_3, V_4, V_5, M, M').$$

$$p_{2_6}(V_0, V_1, V_2, V_3, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 1, A' = A\{0 \leftarrow V_1\}\}.$$

$$p_{2_6}(V_0, V_1, V_2, V_3, V_4, V_5, M, M') \leftarrow \{V_0 > 1, V_0' = V_0 - 1, V_1' = V_1 * V_4, V_2' = 1\}, p_{2_6}(V_0', V_1', V_2', V_3, V_4, V_5, M, M').$$

Figure 9. The exponentiation method (version 2) and its translation into CLP, P_{exp2}.

$$p_{2_0}(V_0, V_1, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_5 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$

$$p_{2_0}(V_0, V_1, V_4, V_5, M, M') \leftarrow \{V_5 > 0\}, p_{2_6}(V_5, V_4, V_4, V_5, M, M').$$

$$p_{2_6}(V_0, V_1, V_4, V_5, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 1, A' = A\{0 \leftarrow V_1\}\}.$$

$$p_{2_6}(V_0, V_1, V_4, V_5, M, M') \leftarrow \{V_0 > 1, V_0' = V_0 - 1, V_1' = V_1 * V_4\}, p_{2_6}(V_0', V_1', V_4, V_5, M, M').$$

Figure 10. A slice of P_{exp2}.

However, slicing can be used to restrict the definition of a predicate to those computations that depend only on a given subset of the predicate's arguments. We therefore limit ourselves to slices that are *correct* in the sense that they preserve the predicate's semantics with respect to a given sequence of argument positions.

Definition 15. *Given a predicate p from a program P, a sequence R of argument positions, and a slice p' of p. We say that the slice p' is* correct w.r.t. *R if (p', R) computes the same in P as does (p, R).*

The above notion of correctness with respect to a sequence of argument positions is easily generalised to a transformation sequence as a whole:

Definition 16. *Given a set of program transformations $\mathcal{R}$, predicates p and p', and sequences of argument positions R and R'. A $\mathcal{R}$-transformation sequence $\langle P_0, P_1, \ldots, P_n\rangle$* correctly transforms *$(p, R)$ into (p', R') if and only if (p, R) computes the same result in P_0 as (p', R') in P_n.*

Definition 16 essentially defines what we will see as a correct transformation sequence: one that preserves the computation performed by a predicate of interest, at least with respect to a subset of its arguments. Note that the definition is parametrized with respect to the set $\mathcal{R}$ of allowed transformations. Also note that the definition is quite liberal, in the sense that it allows predicates to be renamed, arguments (and thus computations) to be left out of the equation, and arguments to be permuted. We are now in a position to define algorithmic equivalence, which we define with respect to the combination of a program, a predicate, and a sequence of argument positions. The definition is loosely based on the notion of a semantic clone pair (Dandois and Vanhoof 2012).

Definition 17. *Given predicates p_1 and p_2 defined in, respectively the programs P_1 and P_2, and sequences of argument positions R_1 and R_2. Then we define P_1 and P_2* algorithmically equivalent *for (p_1, R_1) and (p_2, R_2) if and only if there exists a program Q, predicate q and set of arguments R such that $P_1 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_1, R_1) into (q, R) and $P_2 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_2, R_2) into (q, R).*

Our approach towards defining algorithmic equivalence is somewhat different from other transformation-based approaches in the sense that we consider programs algorithmically equivalent if each of them can be transformed into a third, common, program while preserving the semantics (with respect to a subset of argument positions). As such, the third program captures the essence of the computations performed by the two given programs. Note that if all transformations from $\mathcal{R}$ are reversible, then this is equivalent to transforming P_1 into P_2 or vice versa, as is the more common approach towards defining algorithmic equivalence by transformation (Metzger and Wen 2000). Observe that our definition requires that the same predicate occurs in both program fragments; while this may seem strange, it is easily justified by the fact that renaming a predicate may be considered to be part of any suitable set of transformations $\mathcal{R}$.

Example 18. *Let us recall the programs P_{exp1} and P_{exp2} from Example 8. Figure 11 shows a possible transformation sequence starting from P_{exp1}. As a first step, the call to p_{1_2} in the definition of p_{1_0} is unfolded (part (a) of Figure 11), the third argument (V_2) is removed from each definition and call by slicing (part (b)),*

and then the functor $+1$ is introduced around the first argument of p_{1_2} both in the head of the predicate as in each call (part (c)). Note that, in contrast with the slice we computed for P_{exp2}, only a single argument is removed during the transformation of P_{exp1}. Subsequently, constraint simplification is applied to obtain the final result of the transformation, P_{exp} (Figure 12). Since each transformation (except the slicing operation) preserves the semantics of the predicate of interest, p_{1_0}, we trivially have that the transformation sequence $P_{exp1} \rightsquigarrow^*_{\mathcal{R}} P_{exp}$ correctly transforms $(p_{1_0}, \langle 4, 5\rangle)$ into $(p_{1_0}, \langle 3, 4\rangle)$. Note the shift in argument positions due to the argument that was removed in the process. Now, it is not hard to see that the slice we computed of P_{exp2} (see Figure 10) is nothing but a renaming of P_{exp} and we have thus a transformation sequence $P_{exp2} \rightsquigarrow^*_{\mathcal{R}} P_{exp}$ that correctly transforms $(p_{2_0}, \langle 5, 6\rangle)$ into $(p_{1_0}, \langle 3, 4\rangle)$.

The following easy to prove result justifies our definition for algorithmic equivalence by establishing a formal link between structural equivalence as defined by a transformation sequence and semantic equivalence of the involved programs.

Proposition 19. *Given programs P_1 and P_2, predicates p_1 and p_2 and sequences of argument positions R_1 and R_2. If P_1 and P_2 are algorithmically equivalent for (p_1, R_1) and (p_2, R_2), then (p_1, R_1) computes the same result in P_1 as does (p_2, R_2) in P_2.*

Proof. By algorithmic equivalence (Definition 17) there exist a program Q, predicate p and sequence of argument positions R such that $P_1 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_1, R_1) and $P_2 \rightsquigarrow^*_{\mathcal{R}} Q$ correctly transforms (p_2, R_2). By Definition 16, it follows that (p_1, R_1) computes in P_1 the same as does (q, R) in Q and, by the same definition and the symmetry of the "computes the same" relation (q, R) computes in Q the same as does (p_2, R_2) in P_2. Transitivity of the "computes the same" relation allows us to conclude the proof. □

We conclude this section with a discussion of our approach. Algorithmic equivalence can be seen as an approximation of semantic equivalence, but quite stronger as it incorporates a syntactical component: indeed, the considered algorithms should not only compute the same results, their syntactical representation (at the Horn clause level) should be related by means of the transformations in $\mathcal{R}$. The fact that our notion of algorithmic equivalence is parametrized with this set $\mathcal{R}$ is coherent with the fact that there is no single universally accepted definition for algorithmic equivalence (Blass et al. 2009) and essentially allows us to define a whole hierarchy of characterizations of $\mathcal{R}$-algorithmic equivalence, for different instantiations of $\mathcal{R}$. For instance, by instantiating $\mathcal{R}$ to just $\{id\}$ – with id being the identity transformation – we obtain a very strong characterization in which algorithmic equivalence is basically a synonym for having an identical Horn clause representation. Adding more transformations to $\mathcal{R}$ allows for more liberal characterizations of algorithmic equivalence. For instance, when $\mathcal{R}$ is instantiated with the unfolding rule, we obtain an $\mathcal{R}$-algorithmic equivalent criterion that characterises algorithms as equivalent if their Horn clause representation is identical modulo unfolding. Even with a limited set of transformations in $\mathcal{R}$, the definition allows for multiple degrees of liberty when considering algorithmic equivalence. In particular when a *slicing* transformation is present, algorithms could – in an extreme case – be characterised as equivalent even if they do not share any computation (i.e. when all computations are sliced away in the transformation sequence). This illustrates that the definition, even with a suitable incarnation for $\mathcal{R}$ must be tuned for the application at hand, in particular when the application of interest is related to plagiarism detection.

4. Related Work

The seminal idea of translating imperative programs into CLP for static analysis has been introduced in (Peralta et al. 1998), where a semantics-based interpreter of an imperative programming language is expressed as a CLP program. This interpreter together with a term representation of the imperative program to be analyzed is partially evaluated. The residual CLP program is statically analyzed and the results – invariants expressed as linear inequalities between program variables (Cousot and Halbwachs 1978) – are brought back to the initial imperative program. As another example, (De Angelis et al. 2015) proposes a method for automatically generating verification conditions for imperative programs by program specialization. The approach of (Peralta et al. 1998) has also been applied to Object-Oriented programs. For instance, the Java bytecode static analyzer Julia (Spoto et al. 2010) is able to generate a CLP program whose termination implies the termination of the initial Java bytecode program. As a last example, in (Albert et al. 2012) Java bytecode programs are rewritten into a rule based formalism similar to Horn clauses. Then, given a cost model, cost relations are derived. A cost analysis is automatically inferred by solving such cost relations with the help of a dedicated constraint solver.

Algorithm recognition is a well-established topic in program analysis. We present below some of the main existing works.

Two of the oldest related research projects are the MIT's *Programmer's Apprentice* (see e.g., (Rich et al. 1979)) and the *Knowledge Based Software Assistant*, a research program funded by the United States Air Force (see e.g., (Green et al. 1983)). The underlying idea was to adapt artificial intelligence techniques to help software development.

As an offspring of the Programmer's Apprentice, Linda Mills describes an automated program recognition system in (Wills 1990, 1993). It aims at helping software maintenance, translation, and debugging. Given a program and a library of *clichés*, i.e., programming stereotypes and associated structures, the system builds a hierarchical description of the program in terms of the clichés found and their relationships.

In (Martino and Iannello 1996), the authors present a tool called *PAP Recognizer*, where PAP stands for *Parallelizable Algorithmic Patterns*. It implements a plan-based technique for the hierarchical recognition of concept-instances in the program. It aims at automatically parallelizing the code. Another approach to automatically replace the sequential parts of a program with their parallelized versions is described in (Metzger and Wen 2000). Although similar to the previous work, this approach focusses on the computationnally intensive parts of the program.

While most techniques for algorithm recognition are based on using some kind of pattern or template matching, others try to capture the essence of the algorithm at hand. In (Alias and Barthou 2003), for example, algorithms are converted into a system of recurrence equations. In (Taherkhani 2011; Taherkhani and Malmi 2013), Ahmad Taherkhani proposes to statically summarize programs by means of software metrics and program schemas. Then a decision tree classifier acts as an algorithm recognizer. The approach is evaluated by classifying sorting algorithms written by students.

More recently, (Zhang et al. 2012) specifically targets algorithm plagiarism detection. Their detection mechanism is based on abstracting the algorithm by a signature that is computed from a sequence of core values that should arise in any implementation of the algorithm. One of the advantages of such value based signature is the resilience with respect to several obfuscation techniques.

$$p_{1_0}(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle') \leftarrow \{V_4 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$
$$p_{1_0}(V_0, V_1, V_2, V_3, V_4, M, M') \leftarrow \{V_4 > 0, V_0' = V_4 - 1, V_1' = 1 * V_3\}, p_{1_2}(V_0', V_1', V_2, V_3, V_4, M, M').$$
$$p_{1_2}(V_0, V_1, V_2, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 0, A' = A\{0 \leftarrow V_1\}\}.$$
$$p_{1_2}(V_0, V_1, V_2, V_3, V_4, M, M') \leftarrow \{V_0 > 0, V_0' = V_0 - 1, V_1' = V_1 * V_3\}, p_{1_2}(V_0', V_1', V_2, V_3, V_4, M, M').$$

(a)

$$p_{1_0}(V_0, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle') \leftarrow \{V_4 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$
$$p_{1_0}(V_0, V_1, V_3, V_4, M, M') \leftarrow \{V_4 > 0, V_0' = V_4 - 1, V_1' = 1 * V_3\}, p_{1_2}(V_0', V_1', V_3, V_4, M, M').$$
$$p_{1_2}(V_0, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 0, A' = A\{0 \leftarrow V_1\}\}.$$
$$p_{1_2}(V_0, V_1, V_3, V_4, M, M') \leftarrow \{V_0 > 0, V_0' = V_0 - 1, V_1' = V_1 * V_3\}, p_{1_2}(V_0', V_1', V_3, V_4, M, M').$$

(b)

$$p_{1_0}(V_0, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle') \leftarrow \{V_4 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$
$$p_{1_0}(V_0, V_1, V_3, V_4, M, M') \leftarrow \{V_4 > 0, V_0' = V_4 - 1, V_1' = 1 * V_3\}, p_{1_2}(V_0' + 1, V_1', V_3, V_4, M, M').$$
$$p_{1_2}(V_0 + 1, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 0, A' = A\{0 \leftarrow V_1\}\}.$$
$$p_{1_2}(V_0 + 1, V_1, V_3, V_4, M, M') \leftarrow \{V_0 > 0, V_0' = V_0 - 1, V_1' = V_1 * V_3\}, p_{1_2}(V_0' + 1, V_1', V_3, V_4, M, M').$$

(c)

Figure 11. The transformation of program P_{exp1} from Example 8.

$$p_{1_0}(V_0, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle') \leftarrow \{V_4 \leq 0, A' = A\{0 \leftarrow 1\}\}.$$
$$p_{1_0}(V_0, V_1, V_3, V_4, M, M') \leftarrow \{V_4 > 0\}, p_{1_2}(V_4, V_3, V_3, V_4, M, M').$$
$$p_{1_2}(V_0, V_1, V_3, V_4, \langle A, I\rangle, \langle A', I\rangle) \leftarrow \{V_0 \leq 1, A' = A\{0 \leftarrow V_1\}\}.$$
$$p_{1_2}(V_0, V_1, V_3, V_4, M, M') \leftarrow \{V_0 > 1, V_0' = V_0 - 1, V_1' = V_1 * V_3\}, p_{1_2}(V_0', V_1', V_3, V_4, M, M').$$

Figure 12. The transformed program P_{exp}.

5. Conclusion

We have presented a *generic* approach to algorithm recognition in binary code. Its genericity stems from two points. On the one hand, the technique is generic with respect to the input language, as soon as we can translate it into an Horn-clause based representation mimicking the operational semantics of the original target processor. On the other hand, the approach is generic with respect to the notion of algorithmic equivalence, via its parametric set of program transformation rules.

One key aspect of our approach is the use of Horn clauses as a language for representing what is basically the model of the algorithms being compared. In addition to the before-mentioned genericity advantage, the use of Horn clauses allows one to instantiate $\mathcal{R}$ using a limited number of powerful, general, and well-understood transformations such as unfolding, folding and slicing without the need to resort to more low-level and less-general (or language-dependent) transformations. Nevertheless, the question remains about what *are* desirable incarnations of $\mathcal{R}$ and whether one should impose restrictions on the transformation sequences used in the proof of algorithmic equivalence.

While a general equivalence relation on algorithms might not exist (Blass et al. 2009), suitable incarnations of $\mathcal{R}$ will most probably depend on the particular application at hand. Defining such an incarnation (and the particular notion of algorithmic equivalence that comes with it) remains an open and challenging question, in particular when applications such as plagiarism detection are concerned.

Future work will focus on developing other front-ends dealing with various input languages as well as on investigating the feasibility of implementing the procedure described in Section 3 for a limited instantiation of $\mathcal{R}$. Even when only a limited number of transformations are present in $\mathcal{R}$, developing a search procedure trying to construct an $\mathcal{R}$-transformation sequence is a non-trivial and daunting task that might need guidance and global analysis of the program's at hand.

References

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.

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

Android developers. `http://developer.android.com`.

ART and Dalvik. `http://source.android.com/devices/tech/dalvik/`.

A. Blass, N. Dershowitz, and Y. Gurevich. When are two algorithms the same? *Bull. Symbolic Logic*, 15(2):145–168, 06 2009.

A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, POPL'78, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

C. Dandois and W. Vanhoof. Semantic code clones in logic programs. In E. Albert, editor, *Proc. of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'12)*, volume 7844 of *LNCS*, pages 35–50. Springer, 2012.

E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 91–102, 2015.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *TPLP*, 15(4-5):526–542, 2015.

C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. Report on a knowledge-based software assistant. Technical report, Kestrel Institute, 1983.

J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

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

R. Metzger and Z. Wen. *Automatic Algorithm Recognition and Replacement*. The MIT Press, 2000.

E. Payet and F. Mesnard. Non-termination of Dalvik bytecode via compilation to CLP. In C. Fuhs, editor, *Proc. of the 14th International Workshop on Termination (WST'14)*, pages 65–69, 2014.

J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, pages 246–261, 1998.

A. Pettorossi and M. Proietti. Transformation of logic programs. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.

C. Rich, H. E. Shrobe, and R. C. Waters. Overview of the programmer's apprentice. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 827–828, 1979.

Smali assembler for the dex format. `https://github.com/JesusFreke/smali`.

F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32 (3), 2010.

M. D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC)*, pages 181–191, 2005.

G. Szilágyi, T. Gyimóthy, and J. Małuszyński. Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, 9(1): 41–65, 2002.

A. Taherkhani. Using decision tree classifiers in source code analysis to recognize algorithms: An experiment with sorting algorithms. *Comput. J.*, 54(11):1845–1860, 2011.

A. Taherkhani and L. Malmi. Beacon- and schema-based method for recognizing algorithms from students' source code. *Journal of Educational Data Mining*, 5(2):69–101, 2013.

L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1-2):113–171, 1990.

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

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

F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '14, pages 25–36. ACM, 2014.