

# Non-termination of Dalvik bytecode *via* compilation to CLP

Étienne Payet and Fred Mesnard

Université de La Réunion, EA2525-LIM  
Saint-Denis de La Réunion, F-97490, France  
{etienne.payet,frederic.mesnard}@univ-reunion.fr

---

## Abstract

We present a set of rules for compiling a Dalvik bytecode program into a logic program with array constraints. Non-termination of the resulting program entails that of the original one, hence the techniques we have presented before for proving non-termination of constraint logic programs can be used for proving non-termination of Dalvik programs.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification (Formal methods), F.3.1 Specifying and Verifying and Reasoning about Programs (Mechanical verification), F.3.2 Semantics of Programming Languages (Program analysis)

**Keywords and phrases** Non-Termination, Android, Dalvik, Constraint Logic Programming

## 1 Introduction

Android is currently the most widespread operating system for mobile devices. Applications running on this system can be downloaded from anywhere, hence reliability is a major concern for its users. In this paper, we consider applications that may run into an infinite loop, which may cause a resource exhaustion, for instance the battery if the loop continuously uses a sensor as the GPS. Android programs are written in Java and compiled to the Google's Dalvik Virtual Machine (DVM) bytecode format [3] before installation on a device. We provide a set of rules for compiling a Dalvik bytecode program into a constraint logic program [5]. Non-termination of the resulting program entails that of the original one, hence the technique we have presented before [6] for proving non-termination of constraint logic programs can be used for proving non-termination of Dalvik programs. We model the memory and the objects it contains with *arrays*, so we compile Dalvik programs to logic programs with array constraints and we consider the theory of arrays presented in [1].

## 2 The Dalvik Virtual Machine

We briefly describe the operational semantics of the DVM (see [3] for a complete description). Unlike the JVM which is stack-based, the DVM is register-based. Each method uses its own array of registers and invoked methods do not affect the registers of invoking methods. The number of registers used by a method is statically known. At the beginning of an execution, the  $N$  arguments to a method land in its last  $N$  registers and the other registers are initialized to 0. Many Dalvik bytecode instructions are similar, so we concentrate on a restricted set which exemplifies the operations that the DVM performs.

- *const*  $d, c$  Move constant  $c$  into register  $d$  (*i.e.*, the register at index  $d$  in the array of registers of the method where this instruction occurs).
- *move*  $d, s$  Move the content of register  $s$  into register  $d$ .
- *add*  $d, s, c$  Store the sum of the content of register  $s$  and constant  $c$  into register  $d$ .

- *if-lt*  $i, j, q$  If the content of register  $i$  is less than the content of register  $j$  then jump to program point  $q$ , otherwise execute the immediately following instruction.
- *goto*  $q$  Jump to program point  $q$ .
- *invoke*  $S, m$  where  $S = s_0, s_1, \dots, s_p$  is a sequence of register indexes and  $m$  is a method. The content  $r^{s_0}$  of register  $s_0, \dots, r^{s_p}$  of register  $s_p$  are the *actual parameters* of the call. Value  $r^{s_0}$  is called *receiver* of the call and must be 0 (the equivalent of `null` in Java) or a reference to 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 with the same signature as  $m$ . That method is run from a state where its last registers are bound to  $r^{s_0}, r^{s_1}, \dots, r^{s_p}$ .
- *return* Return from a void method.
- *new-instance*  $d, \kappa$  Move a reference to a new object of class  $\kappa$  into register  $d$ .
- *iget*  $d, i, f$  (resp. *iput*  $s, i, f$ ) The content  $r^i$  of register  $i$  must be 0 or a reference to an object  $o$ . If  $r^i$  is 0, the computation stops with an exception. Otherwise,  $o(f)$  (the value of field  $f$  of  $o$ ) is stored into register  $d$  (resp. the content of register  $s$  is stored into  $o(f)$ ).

### 3 Compilation to CLP clauses

We model a *memory* as a pair  $(a, i)$  where  $a$  is an array of *objects* and  $i$  is the index into this array where the next insertion will take place. An *object*  $o$  is an array of terms of the form  $[w, f_1(v_1), \dots, f_n(v_n)]$  where  $w$  is the name of the class of  $o$ ,  $f_1, \dots, f_n$  are the names of the fields defined in this class and  $v_1, \dots, v_n$  are the current values of these fields in  $o$ . So, the first component of a memory is an array of arrays of terms and a memory location is an index into this array. Memory locations start at 1 and 0 corresponds to the `null` value.

Our compilation rules are given in Fig. 1–3. We associate a predicate symbol  $p_q$  to each program point  $q$  of the Dalvik program  $P$  under consideration. We generate clauses with constraints on integer and array terms. Our constraint theory combines the theory of integers with that of arrays defined in [1]. Our CLP domain of computation  $\mathcal{D}$  (values interpreting constraints) is the union of  $\mathbb{Z}$  with the set  $Obj$  of arrays of terms of the form  $f(i)$  where  $i$  is an integer and with the set of arrays of elements of  $Obj$ . 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, we abbreviate  $a[i] \dots [j]$  with  $a[i, \dots, j]$ .

Each rule considers an instruction *ins* occurring at a program point  $q$ . We let  $\tilde{V} = V_0, \dots, V_{r-1}$  and  $\tilde{V}' = V'_0, \dots, V'_{r-1}$  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.  $V'_i$ ) models the content of register  $i$  before (resp. after) executing *ins*. We let  $M$  denote the input memory and  $M'$  the output memory. So,  $\tilde{V}$  and  $M$  (or  $[A, I]$ ) in the head of the clauses are input parameters while  $M'$  is an output parameter. We let  $id$  denote the sequence  $(V'_0 = V_0, \dots, V'_{r-1} = V_{r-1})$  and  $id_{-i}$  (where  $i \in [0, r-1]$ ) the sequence  $(V'_0 = V_0, \dots, V'_{i-1} = V_{i-1}, V'_{i+1} = V_{i+1}, \dots, V'_{r-1} = V_{r-1})$ . By  $|\tilde{X}|$  we mean the length of sequence  $\tilde{X}$ . For any method  $m$ ,  $q_m$  is the program point where  $m$  starts,  $reg(m)$  is the number of registers used by  $m$  and  $sign(m)$  is the set of all the methods with the same signature as  $m$ .

Some compilation rules are rather straightforward. For instance, *const*  $d, c$  moves constant  $c$  into register  $d$ , so in Fig. 1 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. In Fig. 2, we consider method calls. The instruction *invoke*  $s_0, \dots, s_p, m$  is compiled into a set of clauses (one for each method with the same signature as  $m$ ) which

$$\frac{\text{const } d, c}{p_q(\tilde{V}, M, M') \leftarrow \{V'_d = c\} \cup id_{-d}, p_{q+1}(\tilde{V}', M, M')} \quad (1a)$$

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

■ **Figure 1** Compilation of some simple Dalvik instructions.

$$\frac{\text{invoke } s_0, \dots, s_p, m}{\left\{ \begin{array}{l} p_q(\tilde{V}, M, M') \leftarrow \{V_{s_0} > 0\} \cup id, \\ \text{lookup}_P(M, V_{s_0}, m, q_{m'}), \\ p_{q_{m'}}(\tilde{X}_{m'}, M, M_1), \\ p_{q+1}(\tilde{V}', M_1, M') \end{array} \right\} \left| \begin{array}{l} m' \in \text{sign}(m) \\ \text{and } \tilde{X}_{m'} = 0, \dots, 0, V_{s_0}, \dots, V_{s_p} \\ \text{with } |\tilde{X}_{m'}| = \text{reg}(m') \end{array} \right.} \quad (2a)$$

$$\frac{\text{return}}{p_q(\tilde{V}, M, M') \leftarrow \{M' = M\}} \quad (2b)$$

■ **Figure 2** Compilation of some Dalvik instructions related to method calls.

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. If  $V_{s_0} > 0$ , the lookup procedure begins. For each  $m' \in \text{sign}(m)$ , this is modelled with the call  $\text{lookup}_P(M, V_{s_0}, m, q_{m'})$  which starts from the class of the object at location  $V_{s_0}$  in memory  $M$  and searches for the closest method  $m''$  with the same signature as  $m$  upwards along the superclass chain. If  $m'' = m'$ , this call succeeds, otherwise it fails. Then,  $m'$  is executed, modelled with  $p_{q_{m'}}(\tilde{X}_{m'}, M, M_1)$ , with some registers  $\tilde{X}_{m'}$  initialized as expected. When the execution of  $m'$  has finished, control jumps to the following instruction (*i.e.*,  $p_{q+1}(\tilde{V}', M_1, M')$ ). In Fig. 3, we consider some memory-related instructions that we compile to clauses with array constraints.

► **Theorem 1.** *Let  $P$  be a Dalvik bytecode program and  $P_{CLP}$  its CLP compilation. If there is a computation  $p_{q_0}p_{q_1} \dots$  in  $P_{CLP}$  then there is an execution  $q_0q_1 \dots$  of  $P$ .*

More precisely, if there is a finite (resp. infinite) computation in  $P_{CLP}$  starting from a query  $p_{q_0}(\tilde{v}, [a, i], M')$  (where  $\tilde{v}$ ,  $a$  and  $i$  are values in  $\mathcal{D}$  and  $M'$  is an output variable), then there is a finite (resp. infinite) execution of  $P$ , using the same program points, starting from values corresponding to  $\tilde{v}$  and  $a$  in the DVM registers and memory.

## 4 Non-termination inference

The following proposition is a CLP reformulation of a result presented in [4].

► **Proposition 2.** *Let  $r = p(\tilde{x}) \leftarrow c, p(\tilde{y})$  and  $r' = p'(\tilde{x}') \leftarrow c', p(\tilde{y}')$  be some clauses. Suppose there exists a set  $\mathcal{G}$  such that formulæ  $[\forall \tilde{x} \exists \tilde{y} \tilde{x} \in \mathcal{G} \Rightarrow (c \wedge \tilde{y} \in \mathcal{G})]$  and  $[\exists \tilde{x}' \exists \tilde{y}' c' \wedge \tilde{y}' \in \mathcal{G}]$  are true. Then,  $p'$  has an infinite computation in  $\{r, r'\}$ .*

$$\frac{\text{new-instance } d, \kappa}{\begin{array}{l} w \text{ is the name of class } \kappa \text{ and } f_1, \dots, f_n \text{ are the names of the fields defined in } \kappa \\ p_q(\tilde{V}, [A, I], M') \leftarrow \{O[0] = w, O[1] = f_1(0), \dots, O[n] = f_n(0), \\ A_1 = A\{I \leftarrow O\}, V'_d = I, I_1 = I + 1\} \cup id_{-d}, p_{q+1}(\tilde{V}', [A_1, I_1], M') \end{array}} \quad (3a)$$

$$\frac{\text{iget } d, i, f}{p_q(\tilde{V}, [A, I], M') \leftarrow \{V_i > 0, A[V_i, F] = f(V'_d)\} \cup id_{-d}, p_{q+1}(\tilde{V}', [A, I], M')} \quad (3b)$$

$$\frac{\text{iput } s, i, f}{\begin{array}{l} p_q(\tilde{V}, [A, I], M') \leftarrow \{V_i > 0, O = A[V_i], O[F] = f(X), O_1 = O\{F \leftarrow f(V_s)\}, \\ A_1 = A\{V_i \leftarrow O_1\}\} \cup id, p_{q+1}(\tilde{V}', [A_1, I], M') \end{array}} \quad (3c)$$

■ **Figure 3** Compilation of some memory-related instructions.

Consider the Android program in Fig. 4, with the Java syntax on the left and the corresponding Dalvik bytecode  $P$  on the right, where  $v_0, v_1, \dots$  denote registers  $0, 1, \dots$ . Method `loop` in class `MyActivity` is called when the user taps a button displayed by the application. Execution of this method does not terminate because in the call to `m`, the objects `o1` and `o2` are aliased and therefore by decrementing `x.i` we are also decrementing `this.i` in the loop of method `m`. We get the following clauses for program points 0 and 14:

$$\begin{array}{l} p_0(\tilde{V}, [A, I], M') \leftarrow \{A[V_1, F] = i(V'_0)\} \cup id_{-0}, p_1(\tilde{V}', [A, I], M') \\ p_{14}(\tilde{V}, M, M') \leftarrow \{V_0 > 0\} \cup id, \text{lookup}_P(M, V_0, \text{Loops} \rightarrow \mathbf{m}(\text{ILoops})V, 0), \\ p_0(0, V_0, V_2, V_1, M, M_1), p_{15}(\tilde{V}', M_1, M') \end{array}$$

Let  $P_{CLP}$  denote the CLP program resulting from the compilation of  $P$ . The set of *binary unfoldings* [2] of  $P_{CLP}$  contains the following clauses

$$\begin{array}{l} r: \quad p_0(\tilde{V}, [A, I], M') \leftarrow \{V_1 > 0, O = A[V_1], O[F] = i(X), X < V_2, \\ O_1 = O\{F \leftarrow i(X + 1)\}, A_1 = A\{V_1 \leftarrow O_1\}, \\ V_3 > 0, O' = A_1[V_3], O'[F'] = i(X'), V'_0 = X' - 1, \\ O'_1 = O'\{F' \leftarrow i(V'_0)\}, A_2 = A_1\{V_3 \leftarrow O'_1\}\} \cup id_{-0}, p_0(\tilde{V}', [A_2, I], M') \\ r': \quad p_{10}(\tilde{V}, [A, I], M') \leftarrow \{O[0] = \text{loops}, O[1] = i(0), A_1 = A\{I \leftarrow O\}, \\ I_1 = I + 1, I > 0\}, p_0(0, I, 2, I, [A_1, I_1], M_1) \end{array}$$

where  $r$  corresponds to the path  $0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow \dots \rightarrow 9 \rightarrow 0$  and  $r'$  to the path  $10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \rightarrow 14 \rightarrow 0$  in  $P$ . In  $r'$ ,  $O$  corresponds to both  $o_1$  and  $o_2$ , which expresses that  $o_1$  and  $o_2$  are aliased. Note that  $I$ , the address of  $O$ , is passed to  $p_0$  both as second and fourth parameter, which corresponds in  $r$  to  $V_1$  (`this` in method `m`) and  $V_3$  (`x` in `m`). Moreover, when  $V_1 = V_3$  in  $r$ , we have  $O' = O_1$ ,  $F' = F$  and  $X' = X + 1$ , hence  $V'_0 = X' - 1 = X$ . Therefore, we have  $O'_1 = O$ , so  $A_2 = A$ . The logical formulæ of Proposition 2 are true for the set  $\mathcal{G} = \{(\tilde{v}, \text{mem}, \text{mem}') \in \mathcal{D}^3 | v_1 = v_3\}$ . Hence,  $p_{10}$  has an infinite computation in  $\{r, r'\}$ , which implies [2] that  $p_{10}$  has an infinite computation in  $P_{CLP}$ . So by Theorem 1,  $P$  has an infinite execution from program point 10.

```

public class Loops {
    int i;
    public void m(int n, Loops x) {
        while (this.i < n) {
            this.i++;
            x.i--;
        }
    }
}

.method public m(ILoops)V
    .registers 4
0: iget v0, v1, Loops->i:I
1: if-lt v0, v2, 3
2: return-void
3: iget v0, v1, Loops->i:I
4: add-int/lit8 v0, v0, 0x1
5: iput v0, v1, Loops->i:I
6: iget v0, v3, Loops->i:I
7: add-int/lit8 v0, v0, -0x1
8: iput v0, v3, Loops->i:I
9: goto 0
.end method

public class MyActivity extends Activity {
    ...
    public void loop(View v) {
        Loops o1 = new Loops();
        Loops o2 = o1;
        o1.m(2, o2);
    }
    ...
}
.method public loop(Landroid/view/View;)V
    .registers 5
10: new-instance v0, Loops
11: invoke-direct {v0}, Loops-><init>()V
12: move-object v1, v0
13: const/16 v2, 0x2
14: invoke-virtual {v0, v2, v1}, Loops->m(ILoops)V
15: return-void
.end method

```

■ **Figure 4** The non-terminating method `loop` is called when the user taps a button.

## 5 Future Work

We plan to implement the technique described above and to write a solver for array constraints. Currently, our compilation rules only consider the operational semantics of Dalvik, a part of the Android platform. We also plan to extend them by considering the operational semantics of other components of Android, for instance *activities* that we have studied in [7].

---

### References

- 1 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.
- 2 M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
- 3 Dalvik docs mirror. <http://www.milk.com/kodebase/dalvik-docs-mirror/>.
- 4 A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In G. C. Necula and P. Wadler, editors, *Proc. of POPL’08*, pages 147–158. ACM Press, 2008.
- 5 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.
- 6 É. Payet and F. Mesnard. A non-termination criterion for binary constraint logic programs. *Theory and Practice of Logic Programming*, 9(2):145–164, 2009.
- 7 É. Payet and F. Spoto. An operational semantics for Android activities. In W.-N. Chin and J. Hage, editors, *Proc. of PEPM’14*, pages 121–132. ACM, 2014.