Experiments with Non-Termination Analysis for Java Bytecode

Étienne Payet[1]

IREMIA
Université de La Réunion
France

Fausto Spoto[2]

Dipartimento di Informatica
Università di Verona
Italy

Abstract

Non-termination analysis proves that programs, or parts of a program, do not terminate. This is important since non-termination is often an unexpected behaviour of computer programs and exposes a bug in their code. While research has found ways of proving non-termination of logic programs and of term rewriting systems, this is hardly the case for imperative programs. In this paper, we describe and experiment with a technique for proving non-termination of imperative, bytecode programs by relating their non-termination to that of a (constraint) logic program. Moreover, we show that our non-termination test effectively helps a termination test, by avoiding expensive search for termination proofs of those portions of the code where such proofs do not exist.

Keywords: Java, Java bytecode, static analysis, termination, non-termination

1 Introduction

Java bytecode [8] is the result of the compilation of Java, as well as of other programming languages. It is a low-level, object-oriented, type-safe language which is distributed in a machine-independent format, hence executable on different architectures. It is the target of choice for the compilation of applications that must be downloaded from the net into client computers or mobile phones. The recent Android system by Google [1] uses the Java bytecode as the target of the compilation of Android programs, before translating it into a machine-centered lower-level bytecode.

[1] Email: epayet@univ-reunion.fr

[2] Email: fausto.spoto@univr.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

As a consequence of the wide use of Java bytecode, research is increasingly focused on checking, in an automatic way, that Java bytecode applications are not harmful. This includes the proof that, for instance, they do not overuse the resources of the system. One such resource is time. In particular, proofs of termination of Java bytecode programs guarantee that they will actually terminate. Such proofs are important for the software developer, since they support the quality standards of his product. Nevertheless, termination of computer programs being an undecidable property, the termination of many methods remains unproved and such methods might hence be potentially non-terminating. A direct proof of their non-termination becomes desirable, since it exhibits an actual, typically unexpected behaviour of the program and often means that the non-terminating methods contain a bug. Currently, no system exists to prove the non-termination of Java bytecode methods, since research has mainly been focused on proofs of non-termination for logic programs [4,11,10,2,16,15] and term rewriting systems [21,5,24,22,23,9]. In the recent paper [7], the authors consider non-termination of C programs and [6,14] provide some techniques for testing C programs that detect errors such as program crashes, assertion violation and non-termination. In [20], an approach to automatically check non-termination of imperative programs is introduced; it is based on the generation of invariants that are used to prove that some potential loops are never exited; the technique is experimented on a set of programs written in a fragment of Java and does not consider heap data structures. In this paper, we provide an example where our approach successfully proves the non-termination of a program where a data structure is defined.

This paper provides a first experimentation with the automatic derivation of non-termination proofs for Java bytecode programs. We start from our previous work on a tool *Julia+BinTerm* for the termination analysis of Java bytecode [19]. There, we translated the original Java bytecode program P into a constraint logic program P_{CLP} whose termination entails that of P. Here, we show how, in those cases when the approximation of the bytecodes is *exact*, the non-termination of P_{CLP} entails that of P. Hence, we use the same tool as in [19] to prove the non-termination of Java bytecode programs by exploiting previous results from non-termination analysis of logic programs [10]; namely, we prove the non-termination of P_{CLP} and hence infer, when possible, that of P. Although these results are far from being a definite solution to the problem of non-termination analysis of Java bytecode programs, they represent a first step in that direction and highlight weaknesses of the current approach, that must be solved if non-termination analysis must be applied to real Java and Java bytecode software. Note that, while a notion of *existential* non-termination for C is considered in [7], we instead consider a notion of *universal* non-termination here for the *CLP* program derived from the Java bytecode program.

This paper also shows that our non-termination test effectively helps the termination test defined in [19]. Namely, we use our non-termination test to signal to the termination prover in [19] that some clauses in P_{CLP} diverge, so that it is useless to look for an (often expensive) termination proof for them. Note that this technique is applicable and profitable for all Java bytecode programs, also when the approximation of their bytecodes is not exact or when all their methods actually terminate. Our termination test is applied, indeed, to the *CLP* program, whose clauses might not terminate because of the approximations induced by the abstraction from P to P_{CLP}.

2 Compilation of Java bytecode into constraint logic programs

Java bytecode is a low-level object-oriented type-safe language. Its static analysis is complicated by the fact that it has no explicit structure, differently from high-level languages, and that it uses a stack of temporary variables. Hence the number and type of the variables are different at different program points inside the same method.

We have recently developed a static analysis of Java bytecode programs (and hence of Java programs) that proves the termination of most methods of a program [19]. The idea is that the Java bytecode program is first translated into its *basic blocks* and then an abstract interpretation [3], based on a denotational semantics over those blocks, is applied by using different abstract domains of analysis. The latter provide a conservative approximation of the numerical and structural constraints on the numbers or data structure used by the program: a first domain, for *sharing* [13], determines when data structures bound to program variables might share locations on the heap, so that an update of one variable might also affect the others. This information is exploited in the second domain, for *cyclicity* [12], which determines when the data structure bound to a program variable might contain loops of locations, so that an iteration over that data structure might not necessarily terminate. Both kinds of information are then used in a *path-length* domain [17,19], that computes the relationship between the *size* of program variables before and after the execution of each instruction in the bytecode: the size or path-length of a variable bound to a data structure is the maximal length of pointers that one can follow from that variable; the path-length of a variable bound to an array is the length of the array; the path-length of a numerical variable is its value; the path-length of a Boolean variable is 0 for *false* and 1 for *true*. The result of the path-length is finally used to express the relationship between the size of the variables at the beginning and at the end of each basic block of the program. This is written in terms of a constraint logic program P_{CLP} over linear constraints, whose predicates $b(vars)$ correspond to each basic block b of P and *vars* are the variables at the beginning of the execution of b. These approximations build constraints that are later used in order to derive bounds on the values of variables in programs, which is crucial for termination and non-termination analyses to work. The main result proved in [19], *wrt.* termination analysis, is the following:

Theorem 2.1 *Let P be a Java bytecode program and b a basic block of P. If the query $b(vars)$ has only terminating computations in P_{CLP}, for all fixed integer values for vars, then all executions of a Java Virtual Machine started at b terminate.* $\square$

The converse, however, does not hold in general: we can find programs P and a basic block b of P such that, in the translation P_{CLP}, predicate $b(vars)$ does not terminate for some fixed initial integer values for *vars*, although all executions of P starting at b do terminate. This is due to the approximations done during the translation of P into P_{CLP}: both sharing and cyclicity analyses are approximated, so that, for instance, the analyser might not necessarily prove that a non-cyclical list is actually non-cyclical. Moreover, some bytecodes have an inherently non-linear behaviour, such as multiplications and divisions, and cannot hence be approximated by using the linear constraints available for the path-length.

The translation from Java or Java bytecode to *CLP* makes it uniform the treatment of any kind of loops: `for`, `while` loops, recursion, loops having exit conditions depending on numerical, reference or Boolean variables, loops exiting become of the `break` statement, all become a loop in the graph of blocks of P_{CLP}. The termination of P_{CLP} can hence be established in a uniform way, also in the presence of Boolean variable assigned inside an `if` statement and hence making a loop exit.

An important point about the program P_{CLP} is that its termination is meaningful for *ground* inputs only, where all variables have been bound to their integer path-length (Theorem 2.1). Moreover, the clauses of P_{CLP} are *binary*, that is, they have the form $p(\tilde{X}) \leftarrow c, q(\tilde{Y})$, with only one predicate on the right.

The termination of P_{CLP} is proved by the *BinTerm* tool by F. Mesnard, that finds decreasing measures across iterations of most loops in P_{CLP}. The computational cost of the tool decreases by reducing the number of clauses in P_{CLP}: namely, only clauses in a loop are considered, since they correspond to loops or recursion in the original program P and are those that determine the termination or non-termination of the program. Moreover, its cost is reduced also by decreasing the arity of the predicates, when it is clear that the removed arguments are irrelevant for the termination of the predicates. These optimisations are defined and proved correct in [18]. As a consequence, in all our examples, the *CLP* program will express the path-length relationships for the loops of the program only.

Although the converse of Theorem 2.1 does not hold in general, there are many cases when the approximation of the original program P into path-length is *exact*, in the sense that all denotations represented by the P_{CLP} program are actual denotations that represent real, concrete executions of P. This is the case, for instance, of the approximations of the instructions dealing with integer values, with the notable exception of multiplications and divisions; as well as of instructions dealing with data structures that have been successfully proved to be non-cyclical by the cyclicity analysis. In those frequent cases, a proof of *non*-termination for the *CLP* program induces a proof of *non*-termination for the original Java bytecode program. In the following, we discuss how proofs of non-termination for *CLP* programs can be constructed and exemplify many cases when we can conclude (or not) that the original Java bytecode program does not terminate either.

3 Proving non-termination of constraint logic programs

A non-termination criterion is provided in [10] for the standard operational semantics of constraint logic programming, where free variables may occur in a call to a predicate. The specialisation of this criterion to the semantics we consider in this paper (free variables are not allowed in a call to a predicate) is briefly described in this section.

We consider constraint logic programs over path-length polyhedra ($CLP(\mathbb{PL})$).

We let $\tilde{t}$ denote a sequence of terms, $\tilde{X}$ and $\tilde{Y}$ denote sequences of distinct variables, p and q denote predicate symbols and c denote a path-length constraint. An *atom* has the form $p(\tilde{t})$ where the length of $\tilde{t}$ equals the arity of p. A *query* has the form $\langle p(\tilde{X}) \,|\, c\rangle$. A *clause* has the form $p(\tilde{X}) \leftarrow c, q(\tilde{Y})$ where $\tilde{X}$ and $\tilde{Y}$ are disjoint and the variables occurring in c necessarily occur in $\tilde{X} \cup \tilde{Y}$. A *CLP*($\mathbb{PL}$) *program* is a finite set of clauses. We use $\exists_{\tilde{X}} c$ as a shortcut for $\exists X_1 \dots \exists X_n c$ where $X_1, \dots, X_n := \tilde{X}$. The *projection* of c onto the sequence $\tilde{X}$ is denoted by $\overline{\exists}_{\tilde{X}} c$ and is the constraint $\exists_{\mathit{Var}(c)\setminus\tilde{X}} c$, where $\mathit{Var}(c)$ is the set of variables occurring in c. The *set described by a query* $Q := \langle p(\tilde{X}) \,|\, c\rangle$ is denoted by $\mathit{Set}(Q)$; it consists of all the atoms of the form $p(v(X_1), \dots, v(X_n))$ where $X_1, \dots, X_n := \tilde{X}$ and v is a ground solution of c. We say that $\mathit{Set}(Q)$ is *non-terminating wrt.* a *CLP*($\mathbb{PL}$) program P when for all $p(v(X_1), \dots, v(X_n)) \in \mathit{Set}(Q)$, the query

$$\langle p(X_1, \dots, X_n) \,|\, X_1 = v(X_1), \dots, X_n = v(X_n)\rangle$$

is non-terminating *wrt.* P by using the standard semantics of constraint logic programs. This means that an infinite computation can be built for that query in the program P. Note that we do not consider any precedence between the clauses of P, that is, we assume a non-deterministic resolution of a predicate with all the clauses that define that predicate. The following results provide simple non-termination conditions for constraint logic programs.

Theorem 3.1 ([10]) *Let $p(\tilde{X}) \leftarrow c, p(\tilde{Y})$ be a recursive clause in a CLP($\mathbb{PL}$) program P. If $\mathit{Set}(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq \mathit{Set}(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ then $\mathit{Set}(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ is non-terminating wrt. P.* □

Theorem 3.1 means that if the set of values assigned to $\tilde{Y}$ by all the solutions of c is included in the set of values assigned to $\tilde{X}$ by all the solutions of c, then any value assigned to $\tilde{X}$ by a solution of c provides a non-terminating ground query. Indeed, intuitively, the constraint $\overline{\exists}_{\tilde{X}} c$ is the guard of the clause and $\mathit{Set}(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq \mathit{Set}(\langle p(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ means that every output value of the clause satisfies this guard. Hence, if a value satisfies the guard, then it enters the clause and the corresponding output satisfies the guard, so this output can also enter the clause and the next output satisfies the guard, and so on. Notice that the converse of the implication in Theorem 3.1 does not always hold: consider for instance the recursive clause $p(X) \leftarrow X \geq 3, p(Y)$; we have that $\mathit{Set}(\langle p(X) \,|\, \overline{\exists}_X X \geq 3\rangle)$, *i.e.* $\mathit{Set}(\langle p(X) \,|\, X \geq 3\rangle)$, is non-terminating wrt. this clause although $\mathit{Set}(\langle p(Y) \,|\, \overline{\exists}_Y X \geq 3\rangle)$, *i.e.* $\mathit{Set}(\langle p(Y) \,|\, \mathit{true}\rangle)$, is not included in $\mathit{Set}(\langle p(X) \,|\, X \geq 3\rangle)$.

Theorem 3.2 ([10]) *Let $q(\tilde{X}) \leftarrow c, p(\tilde{Y})$ be a clause in a CLP($\mathbb{PL}$) program P and Q be a query such that $\mathit{Set}(Q)$ is non-terminating wrt. P. If $\mathit{Set}(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq \mathit{Set}(Q)$ then $\mathit{Set}(\langle q(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ is non-terminating wrt. P.* □

The intuition of Theorem 3.2 is that any value $q(\tilde{x})$ in $\mathit{Set}(\langle q(\tilde{X}) \,|\, \overline{\exists}_{\tilde{X}} c\rangle)$ satisfies $\overline{\exists}_{\tilde{X}} c$, the guard of the clause, and the corresponding output $p(\tilde{y})$ is included in $\mathit{Set}(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle)$. As $\mathit{Set}(\langle p(\tilde{Y}) \,|\, \overline{\exists}_{\tilde{Y}} c\rangle) \subseteq \mathit{Set}(Q)$ and $\mathit{Set}(Q)$ is non-terminating *wrt.* P, then $p(\tilde{y})$ does not terminate *wrt.* P, so $q(\tilde{x})$ does not terminate also.

These theorems provide a simple mechanism to infer ground non-terminating queries: first, use Theorem 3.1 to infer a set of non-terminating queries from the recursive clauses of the program and then complete this set with the help of Theorem 3.2.

4 Proving non-termination of Java bytecode programs

In this section, we give several examples of situations where we can conclude the non-termination of the original program from that of the *CLP* program, as well as examples where instead this is not possible.

4.1 Exact approximations with iterations

When the approximation into a path-length constraint of the Java bytecode program P under analysis is exact, a proof of non-termination of P_{CLP} is also a proof of non-termination of P. The formal definition of *exact* requires the bytecodes to have a concrete behaviour which is exactly matched by their numerical abstraction, that is, every pair of states satisfying the input/output abstraction of the bytecode must correspond to an actual, concrete behaviour of the bytecode. Note that the converse must always hold by the correctness of the abstraction.

Definition 4.1 [Exact Abstraction] Let *ins* be a bytecode instruction, formalised as an input/output map on concrete JVM states, as in [19], and let $ins^{\mathbb{PL}}$ be a correct approximation of its behaviour, *i.e.*, a constraint over input variables $\check{v}$ and output variables $\hat{v}$. This approximation is *exact* if and only if, for all input states $\check{\sigma}$ and output variable $\hat{\sigma}$ satisfying the static information at *ins* (number and type of local variables and stack elements), whenever $\{\check{v} \mapsto \mathit{pathlength}(\check{\sigma}(v))\} \cup \{\hat{v} \mapsto \mathit{pathlength}(\hat{\sigma}(v))\} \models ins^{\mathbb{PL}}$ then $\sigma(\check{\sigma}) = \hat{\sigma}$. □

Consider for instance the program *Add1*:

```
public class Add1 {
    public static void main(String args[]) {
        int k = 3;
        for(int i = 2; i < 2 + k; i++);
    }
}
```

The approximation of the bytecode program corresponding to *Add1* is exact: the loop guard involves the *add* bytecode instruction whose approximation, as provided in [19], is

$$add_q^{\mathbb{PL}} = \mathit{Unchanged}_q(\#l, \#s-2) \cup \{\check{s}^{\#s-2} + \check{s}^{\#s-1} = \hat{s}^{\#s-2}\}$$

where $\#l$ and $\#s$ are the number of local variables and stack elements at program point q where the instruction occurs; we distinguish between variables v at the beginning of the execution of the bytecode, written as $\check{v}$, and variables at its end, written as $\hat{v}$. The formula above means that *add* does not modify any local variable nor any stack element not involved in the addition; moreover, the new top of the stack ($\hat{s}^{\#s-2}$) holds a value which is equal the addition of the former two topmost stack elements ($\check{s}^{\#s-2}$ and $\check{s}^{\#s-1}$). This approximation is exact since, for every couple of input state $\check{\sigma}$ and output state $\hat{\sigma}$ satisfying the static information at this bytecode and the approximation above, we must have that the local variables have the same values in $\check{\sigma}$ and $\hat{\sigma}$ and the top of the stack of $\hat{\sigma}$ is the sum of the topmost two values on top of the stack of $\check{\sigma}$, so that those states are such that $add_q(\check{\sigma}) = \hat{\sigma}$. The corresponding $CLP(\mathbb{PL})$ program $Add1_{CLP}$ is:

$$entry(IL2) \leftarrow \{IL2 - OL2 = -1,\ -IL2 \geq -4,\ IL2 \geq 2\}, entry(OL2)$$

The predicate *entry* denotes the entry point of the loop of the program; local variable 2 implements `i` while variable `k` has been removed since it is irrelevant for the termination of the program. This *CLP* program has been derived by using the abstract interpretations cited in the introduction. Namely, we have used the path-length abstract analysis, which has derived the constraint $IL2 - OL2 = -1$ (that is, local variable 2, which is `i`, decreases along iterations of the loop) and the constraints $-IL2 \geq -4,\ IL2 \geq 2$, which provide bounds on the possible values of that variable inside the loop. That *CLP* program terminates. By Theorem 2.1 we conclude that *Add1* terminates also. If we turn *Add1* into the non-terminating program:

```
public class Add2 {
    public static void main(String args[]) {
        int k = 3;
        for(int i = 2; i < 2 + k; i--);
    }
}
```

we get the $CLP(\mathbb{PL})$ program $Add2_{CLP}$:

$$entry(IL2) \leftarrow \{IL2 - OL2 = 1,\ -IL2 \geq -2\}, entry(OL2)$$

which by Theorem 3.1 does not terminate because the projection of the constraint of its unique clause onto $IL2$ (resp. $OL2$) is $-IL2 \geq -2$ (resp. $-OL2 \geq -1$) and we have

$$Set(\langle entry(OL2) \mid -OL2 \geq -1\rangle) \subseteq Set(\langle entry(IL2) \mid -IL2 \geq -2\rangle)\ .$$

Here, we can safely conclude the non-termination of *Add2* from that of $Add2_{CLP}$.

Our technique is also able to handle more complicated situations. For instance, if we nest the non-terminating loop of program *Add2* into a terminating loop, we get:

```
public class Add3 {
    public static void main(String args[]) {
        int k = 3;
        for(int j = 0; j < 10; j++)
            for (int i = 2; i < 2 + k; i--);
    }
}
```

The corresponding $CLP(\mathbb{PL})$ program $Add3_{CLP}$:

$$entry(IL3) \leftarrow \{OL3 = 2\}, block(OL3)$$

$$block(IL3) \leftarrow \{IL3 - OL3 = 1,\ -IL3 \geq -2\}, block(OL3)$$

does not terminate. Note that the outer loop does not appear in the *CLP* program, since the exit condition $i \geq 2 + k$ of the inner loop is found to be false during the path-length analysis and no clause is generated with a *false* constraint. Indeed, such clause would not influence the termination or non-termination behaviour of the program, since it would just stop the *CLP* resolution process. Indeed, by applying Theorem 3.1 to the recursive clause we get that $Set(Q)$ is non-terminating *wrt.* $Add3_{CLP}$ where

$$Q := \langle block(IL3) \,|\, -IL3 \geq -2 \rangle \ .$$

Notice that we have to infer a non-terminating query of the form $\langle entry(\cdots) \,|\, \cdots \rangle$ to conclude the non-termination of $Add3_{CLP}$ because the entry point of the loops of the program is the predicate *entry*. The projection of the constraint of the first clause onto $OL3$ is $OL3 = 2$ and we have

$$Set(\langle block(OL3) \,|\, OL3 = 2 \rangle) \subseteq Set(Q) \ .$$

Hence, by Theorem 3.2 applied to the first clause of $Add3_{CLP}$ and to Q, we have that $Set(\langle entry(IL3) \,|\, true \rangle)$ is non-terminating *wrt.* $Add3_{CLP}$ (where *true* denotes the always satisfiable constraint). Therefore, $Add3_{CLP}$ does not terminate so we conclude that *Add3* does not terminate either.

If we nest the non-terminating loop of program *Add2* into a separated method, such as in:

```
public class Add4 {
    public static void loop(int k) {
        for(int i = 2; i < 2 + k; i--);
    }
    public static void main(String args[]) {
        loop(3);
    }
}
```

we get the $CLP(\mathbb{PL})$ program $Add4_{CLP}$:

$$entry(IL1) \leftarrow \{IL1 - OL1 = 1,\ -IL1 \geq -2\}, entry(OL1)$$

which does not terminate (by Theorem 3.1). Hence we conclude that *Add4* does not terminate either.

4.2 Exact approximations with recursion

The following terminating Java program involves a recursive method:

```
public class Rec1 {
    public static int sum(int n) {
        if (n <= 0) return 0;
        else return n + sum(n-1);
    }
    public static void main(String args[]) {
        sum(2);
    }
}
```

The $CLP(\mathbb{PL})$ program $Rec1_{CLP}$:

$$entry(IL0) \leftarrow \{IL0 - OL0 = 1,\ IL0 \geq 1,\ -IL0 \geq -2\}, entry(OL0)$$

terminates, hence by Theorem 2.1 we conclude that *Rec1* terminates. If we turn *Rec1* into the following non-terminating program (where the programmer forgot the base case in the recursive method):

```
public class Rec2 {
    public static int sum(int n) {
        return n + sum(n-1);
    }
    public static void main(String args[]) {
        sum(2);
    }
}
```

we get the $CLP(\mathbb{PL})$ program $Rec2_{CLP}$:

$$entry(IL0) \leftarrow \{IL0 - OL0 = 1,\ -IL0 \geq -2\}, entry(OL0)$$

By Theorem 3.1, $Rec2_{CLP}$ does not terminate. As the approximation of the bytecode program corresponding to *Rec2* is exact, we can safely conclude that *Rec2* does not terminate either.

4.3 Exact approximations with data structures

All examples above deal with integer values only. Let us consider the following program now, where a list data structure is defined and recursively scanned:

```
public class List {
    private int head;
    private List tail;
    public List(int head, List tail) {
        this.head = head;
        this.tail = tail;
    }
    private void iter() {
        if (tail != null) iter();
    }
    public static void main(String args[]) {
        List l = new List(0, new List(1, null));
        l.iter();
    }
}
```

The method *iter* (intended to perform an iteration over a list) contains a bug since it recurs on the same list rather than on its tail (`iter()` instead of `tail.iter()`). The bytecode version of this program has an exact approximation as our cyclicity analysis correctly infers that the list l in the method *main* is not cyclical. The corresponding $CLP(\mathbb{PL})$ program $List_{CLP}$:

$$entry \leftarrow true,\ entry$$

($true$ denotes the always satisfiable constraint) does not terminate, hence we safely conclude that the program *List* does not terminate either.

4.4 Non-exact approximations

Consider the *mul* bytecode instruction that removes the two top operand stack elements and replaces them with the result of their multiplication. As there is no linear way of expressing a constraint on the result of the multiplication, we just set

$$mul_q^{\mathbb{PL}} = Unchanged_q(\#l, \#s - 2)$$

($\#l$ and $\#s$ are the number of local variables and stack elements at program point q where the instruction occurs) meaning that the instruction does not modify any local variables nor any stack element which are not its operands; however, no constraint on the new top of the stack (the result of the multiplication) is generated. The Java program:

```
public class Mul {
    public static void main(String args[]) {
        int k = 3;
        for(int i = 2; i < 2 * k; i++);
    }
}
```

terminates. Notice that the guard of the loop involves a multiplication. The corresponding $CLP(\mathbb{PL})$ program Mul_{CLP}:

$$entry(IL2) \leftarrow \{IL2 - OL2 = -1,\ IL2 \geq 2\}, entry(OL2)$$

does not terminate. Indeed, the projection of the constraint of the unique clause of Mul_{CLP} onto $IL2$ (resp. $OL2$) is $IL2 \geq 2$ (resp. $OL2 \geq 3$) and we have

$$Set(\langle entry(OL2) \mid OL2 \geq 3\rangle) \subseteq Set(\langle entry(IL2) \mid IL2 \geq 2\rangle) .$$

Therefore, by Theorem 3.1, the non-empty set $Set(\langle entry(IL2) \mid IL2 \geq 2\rangle)$ is non-terminating *wrt.* Mul_{CLP}. However, the non-termination of *Mul* does not follow from this result, since we are using approximated constraints.

We are facing a similar situation when dealing with numeric fields. The *getfield* f instruction takes the reference to an object o located on top of the stack and replaces it with the value of $o.f$. In [19] we defined

$$getfield_q^{\mathbb{PL}}\ f = Unchanged_q(\#l, \#s - 1)$$

whenever the field f has integer type ($\#l$ and $\#s$ are the number of local variables and stack elements at program point q where the instruction occurs). No constraint is generated for the new top of the operand stack (the value of the field) since its path-length is unknown. The Java program:

```
public class Field {
    private int n = 6;
    public static void main(String args[]) {
        Field f = new Field();
        for(int i = 2; i < f.n; i++);
    }
```

```
}
```

terminates. The corresponding $CLP(\mathbb{PL})$ program $Field_{CLP}$:

$$entry(IL2) \leftarrow \{OL2 - IL2 = 1\}, entry(OL2)$$

does not terminate as the projection of the constraint of its clause onto $IL2$ or onto $OL2$ is the always satisfiable constraint $true$ and we have

$$Set(\langle entry(OL2) \mid true\rangle) \subseteq Set(\langle entry(IL2) \mid true\rangle)\ .$$

5 Using non-termination proofs to support termination analysis of Java bytecode

A completely different use of our non-termination tests consists in proving the non-termination of clauses of the P_{CLP} program generated during the termination analysis of a Java bytecode program P. By removing such clauses, which cannot have any termination proof, we help the termination checker by simplifying its task. Since our non-termination tests are extremely efficient, while a thorough quest for a termination proof is in general expensive, the trade-off is positive and we get a more efficient termination analysis still keeping the same precision.

In particular, we have implemented the non-termination tests of Section 3 to help the termination prover *BinTerm* used in the tool *Julia+BinTerm* [19]. Given a Java bytecode program P, our approach consists in a preliminary analysis which considers the strongly connected components (SCCs) of P_{CLP}; any SCC where a non-terminating ground query is found is removed from P_{CLP} and the resulting CLP program P'_{CLP} is analysed by *BinTerm*.

We have run *Julia+BinTerm* on the following Java bytecode programs using a Linux machine based on a 2.33GHz Intel Core 2 Duo with 2 gigabytes of RAM.

P	number of methods in P	number of clauses in P_{CLP}
JavaCup	270	170
JLex	137	356
Kitten	2149	1224

The next table summarizes the results. For each program P, it reports: the number of clauses removed from P_{CLP} by the non-termination analysis; the non-termination analysis time; the *BinTerm* running time on P'_{CLP}; the *BinTerm* running time on P_{CLP}. All the times are in seconds.

P	clauses removed	non-termination analysis	*BinTerm* on P'_{CLP}	*BinTerm* on P_{CLP}
JavaCup	113	0.09s	3.90s	5.66s
JLex	204	0.20s	21.20s	55.30s
Kitten	288	0.68s	99.52s	100.99s

In these experiments on large programs, the computational overhead of the non-termination analysis is not important and the running time of *BinTerm* is smaller on P'_{CLP} than on P_{CLP}. For JLex, *BinTerm* is more than twice faster on P'_{CLP} than on P_{CLP}, as the non-termination analysis removes 204 clauses from P_{CLP} out of 356; among the removed clauses, there is a huge SCC containing 122 clauses where the arity of the involved predicate symbols is 8, which explains the gain in efficiency. On the contrary, the clauses removed for Kitten are several but include relatively small components and have small arity, so that the gain in efficiency is not significant there. This is because the cost of the termination analysis increases significantly with the arity of the predicates and, by removing clauses with small arity, we do not affect very much the efficiency of the termination analysis.

6 Conclusion

In this paper, we have presented some experiments with the automatic derivation of non-termination proofs for Java bytecode programs. When the approximation of the bytecodes into a path-length constraint is exact, the non-termination of the original program can be deduced from that of its *CLP* translation. When the approximation is not exact, it may happen that the bytecode program terminates while its *CLP* version does not terminate (Section 4.4 illustrates this situation). As a future work, we plan to replace some non-exact approximations (such as that of the *getfield* instruction or of the non-linear arithmetic operations) with exact ones that are suitable for deriving non-termination proofs of Java bytecode programs. To that purpose, a possibility is that of finding specific executions that make the program diverge, instead of proving a universal non-termination. In that direction, we might make some program variables *ground*, hence linearising some operations. This would be similar to the technique used in [6].

We have also implemented the non-termination tests of Section 3 in order to help the termination prover *BinTerm* used in the tool *Julia+BinTerm*. The results we have presented in Section 5 are encouraging; even for some large Java bytecode programs, the computational overhead of the non-termination analysis is unimportant; moreover, the termination prover *BinTerm* runs much faster when the components detected as non-terminating are removed from the *CLP* translation of the original bytecode program.

References

[1] Android - An Open Handset Alliance Project. `http://code.google.com/android/`.

[2] R. N. Bol, K. R. Apt, and J. W. Klop. An Analysis of Loop Checking Mechanisms for Logic Programs. *Theoretical Computer Science*, 86:35–79, 1991.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[4] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A Practical Technique for Detecting non-Terminating Queries for a Restricted Class of Horn Clauses, using Directed, Weighted Graphs. In *Proc. of ICLP'90*, pages 649–663. The MIT Press, 1990.

[5] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and Disproving Termination of Higher-order Functions. In Bernhard Gramlich, editor, *Proc. of the 5th International Workshop on Frontiers of Combining Systems (FroCoS'05)*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231. Springer-Verlag, 2005.

[6] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In V. Sarkar and M. W. Hall, editors, *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

[7] 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 the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 147–158. ACM, 2008.

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

[9] É. Payet. Loop Detection in Term Rewriting Using the Eliminating Unfoldings. *Theoretical Computer Science*, 403:307–327, 2008.

[10] É. Payet and F. Mesnard. A non-Termination Criterion for Binary Constraint Logic Programs. Technical report, IREMIA, Université de La Réunion, 2008. Available at `http://arxiv.org/abs/0807.3451`.

[11] É. Payet and F. Mesnard. Non-Termination Inference of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 28, Issue 2:256–289, March 2006.

[12] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'06)*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006. Springer-Verlag.

[13] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin and I. Siveroni, editors, *Proc. of Static Analysis Symposium (SAS'05)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, 2005.

[14] K. Sen, D. Marinov, and G. Agha. CUTE: a Concolic Unit Testing Engine for C. In M. Wermelinger and H. Gall, editors, *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005.

[15] Y-D. Shen, J-H. You, L-Y. Yuan, S. Shen, and Q. Yang. A Dynamic Approach to Characterizing Termination of General Logic Programs. *ACM Transactions on Computational Logic*, 4(4):417–434, 2003.

[16] Y-D. Shen, L-Y. Yuan, and J-H. You. Loops Checks for Logic Programs with Functions. *Theoretical Computer Science*, 266(1-2):441–461, 2001.

[17] F. Spoto, P. M. Hill, and É. Payet. Path-Length Analysis for Object-Oriented Programs. In *First International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, Vienna, Austria, March 2006. Available at the web address `http://profs.sci.univr.it/~spoto/papers.html`.

[18] F. Spoto, L. Lu, and F. Mesnard. Using CLP Simplifications to Improve Java Bytecode Termination Analysis. Submitted for publication to Bytecode'09.

[19] F. Spoto, F. Mesnard, and É. Payet. A Termination Analyser for Java Bytecode Based on Path-Length. Submitted for publication in August 2007.

[20] H. Velroyen and P. Rümmer. Non-Termination Checking for Imperative Programs. In B. Beckert and R. Hähnle, editors, *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966 of *Lecture Notes in Computer Science*, pages 154–170, Prato, Italy, April 2008. Springer.

[21] J. Waldmann. Matchbox: A Tool for Match-bounded String Rewriting. In Vincent van Oostrom, editor, *Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer-Verlag, 2004.

[22] J. Waldmann. Compressed Loops (draft). Available at `http://dfa.imn.htwk-leipzig.de/matchbox/methods/`, 2007.

[23] H. Zankl and A. Middeldorp. Nontermination of String Rewriting using SAT. In *Proc. of the 9th International Workshop on Termination (WST'07)*, pages 52–55, 2007.

[24] H. Zantema. Termination of String Rewriting Proved Automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.