Using CLP Simplifications to Improve Java Bytecode Termination Analysis

Fausto Spoto[1]

Dipartimento di Informatica
Università di Verona
Italy

Lunjin Lu[2]

Oakland University
USA

Fred Mesnard[3]

LIM IREMIA
Université de la Réunion
France

Abstract

In an earlier work, a termination analyzer for Java bytecode was developed that translates a Java bytecode program into a constraint logic program and then proves the termination of the latter. An efficiency bottleneck of the termination analyzer is the construction of a proof of termination for the generated constraint logic program, which is often very large in size. In this paper, a set of program simplifications are presented that reduce the size of the constraint logic program without changing its termination behavior. These simplifications remove program clauses and/or predicate arguments that do not affect the termination behavior of the constraint logic program. Their effect is to reduce significantly the time needed to build the termination proof for the constraint logic program, as our experiments show.

Keywords: Java, Java bytecode, static analysis, termination

1 Introduction

Termination analysis attempts to prove that programs terminate. Since termination of Turing-equivalent programming languages is undecidable [18], termination analysis only succeeds for a (hopefully large) class of programs, although many

[1] Email: fausto.spoto@univr.it
[2] Email: lunjin.lu@gmail.com
[3] Email: frederic.mesnard@univ-reunion.fr

terminating programs are not proved to terminate. Despite this limitation, it is increasingly important in software technology, since proofs of termination add value to software downloaded from insecure networks into computers or cellular phones: the user wants a proof that that software will actually terminate and yield a result or otherwise he will not use it and pay for it.

Termination analyses have been developed for logic [8,10,7], functional programs [14] and term rewrite systems [11], whose semantics is relatively simple and well understood. More recently, termination analysis has been applied to imperative programs, dealing with primitive values only [9,15], lists [13,6,5,4] or any dynamic data-structure [17]. In all cases, termination is typically proved by showing that some well-founded *measure* decreases along loops and recursion, so that divergence cannot occur. This measure can be the value of a variable of primitive type, the length of a list, the maximal path of pointers reachable from a given variable [16] or a mix of such values. When generic data structures are considered, the shape of the computer memory must be somehow approximated, since destructive updates mute dynamic data through shared pointers. Possibly cyclical data structures must be detected, since iterations over them might diverge.

In [17], a termination analysis is defined working for any sequential Java bytecode program [12], dealing with any dynamic data structure, possibly cyclical and shared. Since Java is compiled into Java bytecode, that technique can also be used for termination analysis of Java. It works by translating the Java bytecode program into a constraint logic program (*CLP*) expressing size relationships between program variables at different program points. It has been proved in [17] that if the *CLP* program terminates then the original Java bytecode program terminates. Hence all techniques for termination analysis of *CLP* can be used to prove termination of Java and Java bytecode. In [17], the BinTerm termination prover is used to that purpose. Experiments scale to programs of up to 1000 methods. Although this is already an impressive result, it must be acknowledged that the analysis is expensive in terms of the time needed to build the proof of termination.

In this paper we contribute to the termination analysis of Java and Java bytecode programs. Namely,

- we present a set of simplifications of the *CLP* programs generated by the termination analysis in [17]. They transform the program by removing clauses or variables, yet preserving its behaviour *w.r.t.* termination;
- we prove those transformations correct;
- we experiment with those transformations and show them effective: they reduce by orders of magnitude the cost of finding a termination proof for the *CLP* programs.

These techniques are now embedded in the termination prover for Java bytecode available at the address `http://julia.scienze.univr.it/termination`.

Although some of our simplifications are, often implicitly, used in the termination analysis of programs, this is not the case for others. Namely, the restriction to only those clauses that form a loop in the code (Subsection 4.1) cannot be applied to

```
public class List<X> {
  private X head; private List<X> tail;
  public List(X[] values) { this(values,0); }
  public List(X h, List<X> t) { head = h; tail = t; }
  private List(X[] values, int l) {
    while (l < values.length && values[l] == null) l++;
    if (l < values.length) {
      this.head = values[l];
      if (l + 1 < values.length)
        this.tail = new List<X>(values,l + 1);
    }
  }
  public List<X> append(List<X> other) {
    if (tail == null) return new List<X>(head,other);
    else return new List<X>(head,tail.append(other));
  }
  public void afterInteger() { afterIntegerAux(false); }
  private void afterIntegerAux(boolean wasInteger) {
    if (head instanceof Integer) {
      if (tail != null) tail.afterIntegerAux(true);
    } else {
      if (tail != null) tail.afterIntegerAux(false);
      if (wasInteger) head = null;
    }
  }
  public String toString() {
    if (tail == null) return "* ";
    else if (head instanceof Integer) return "* " + tail.tail.toString();
    else return "* " + tail.toString();
  }
  public static void main(String[] args) {
    Object[] vs = { new Object(),3,3.14,null,new List<Integer>(3,null) };
    List<Object> list1 = new List<Object>(vs);
    List<Object> list2 = new List<Object>(vs);
    list2.afterInteger();
    String s = list1.append(list2).toString();
  }
}
```

Fig. 1. An example Java program.

other frameworks, such as the termination analysis of logic programs, since one needs the removed clauses there, in order to take care of instantiation patterns due to the presence of logical variables (which do not exist in our setting). Also the simplifications based on removing variables which are irrelevant for termination are new (Subsection 4.4). Moreover, we present all such simplifications together and prove them correct in a uniform setting, which was not the case before. Furthermore, we experiment with their effects on the termination analysis of real, large software, which was never the case before; in particular, those simplifications have never been applied to the termination analysis of Java bytecode.

2 Our Running Example

Consider the Java program in Figure 1. It implements a generic list of elements of type `X`. Two constructors are available. The first builds a list from head and tail; the second builds recursively a list from an array. The method `append` concatenates two lists `this` and `other`. The method `afterInteger` writes `null` after all elements of the lists of type `Integer`. Method `toString()` yields a `String` representing the list elements as asterisks, but does not represent the elements that follow an object of type `Integer`. All these methods are recursive. Method `main` builds some lists and calls the previous methods.

We compile this program into Java bytecode and analyse the bytecode as in [17].

Our system tells us that the program terminates. We refer to [17] for the detailed description of how our system works. Here, we briefly give an intuition. First, the Java bytecode is transformed into a graph of *basic blocks* [1], as done in Figure 2 for method `append`. Recursion is made explicit by linking each method call to the beginning of the called method(s), as we do for block 6560 in Figure 2. The makescope $\boldsymbol{\tau}$ pseudo-bytecode creates the activation stack for a method with arguments of type $\boldsymbol{\tau}$. The `catch` pseudo-bytecode marks the beginning of a default exception handler which throws back all exceptions to the caller. Bytecodes inside each block are abstracted into a linear constraint c over-approximating the path-length of each local variable and stack element at its beginning and at its end [16]. For instance, for block 6391 we have $c = \{\mathtt{IS0} - \mathtt{OS1} = 0, \mathtt{IL1} - \mathtt{OS4} = 0, \mathtt{IS0} - \mathtt{OS0} = 0, \mathtt{IL1} - \mathtt{OL1} = 0, \mathtt{IL0} - \mathtt{OL0} = 0, \mathtt{OS3} \geq 0, \mathtt{OS2} \geq 0, \mathtt{IL0} - \mathtt{OS3} \geq 1, \mathtt{IL0} - \mathtt{OS2} \geq 1\}$. The variables $\mathtt{IS}n$ stand for the path-length of the nth stack element at the beginning of the block; $\mathtt{OS}n$ for their path-length at the end of the block; $\mathtt{IL}n$ and $\mathtt{OL}n$ are the same for the nth local variable. This constraint is then used to build *CLP* clauses. In principle, there is a *CLP* clause for each arrow in the graph of basic blocks. Let $\mathtt{block}i$ be a predicate expressing the path-length of the variables in scope at the beginning of block i. Its arity depends on which local variables and stack elements are in scope at the beginning of block i. We build clauses

$$\begin{aligned} &\mathtt{block6391}(\mathtt{IL0}, \mathtt{IL1}, \mathtt{IS0}) : -c, \mathtt{block6392}(\mathtt{OL0}, \mathtt{OL1}, \mathtt{OS0}, \mathtt{OS1}, \mathtt{OS2}, \mathtt{OS3}, \mathtt{OS4}). \\ &\mathtt{block6391}(\mathtt{IL0}, \mathtt{IL1}, \mathtt{IS0}) : -c, \mathtt{block6560}(\mathtt{OL0}, \mathtt{OL1}, \mathtt{OS0}, \mathtt{OS1}, \mathtt{OS2}, \mathtt{OS3}, \mathtt{OS4}). \end{aligned} \tag{1}$$

since two arrows connect block 6391 with blocks 6392 and 6560. Two local variables `L0` and `L1` are in scope there (`L0` implements `this` and `L1` implements `other`). At the beginning of block 6391 there is only one stack element `S0`, while there are 5 at its end. Those clauses form a *CLP* program whose termination entails that of the original Java bytecode program [17]. The clauses of that program have exactly one predicate on their right.

Although the program in Figure 1 is relatively small, the number of arrows in its graph of basic blocks is quite large: the resulting *CLP* program consists of 297 clauses. The aim of the present paper is to introduce simplification techniques for such *CLP* programs which shorten the termination proofs. Next sections formalize our notion of *CLP* programs and show how these programs can be simplified.

3 CLP over Linear Integer Constraints

We formalise here the *CLP* programs of the previous section. Namely, they are sets of predicates, each defined by a set of clauses. We require that predicates are named $\mathtt{block}x$ or $\mathtt{entry}x$. Predicates are not distinguished by their arity. That is, two different predicates must be distinct identifiers. For our purposes, clauses arise from arrows in the graph of basic blocks, so we can assume them to have the form $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o})$, where $\boldsymbol{i}$ and $\boldsymbol{o}$ are disjoint sequences of distinct variables and c is a linear integer constraint on $\boldsymbol{i}$ and $\boldsymbol{o}$. This is similar to [8] and more general

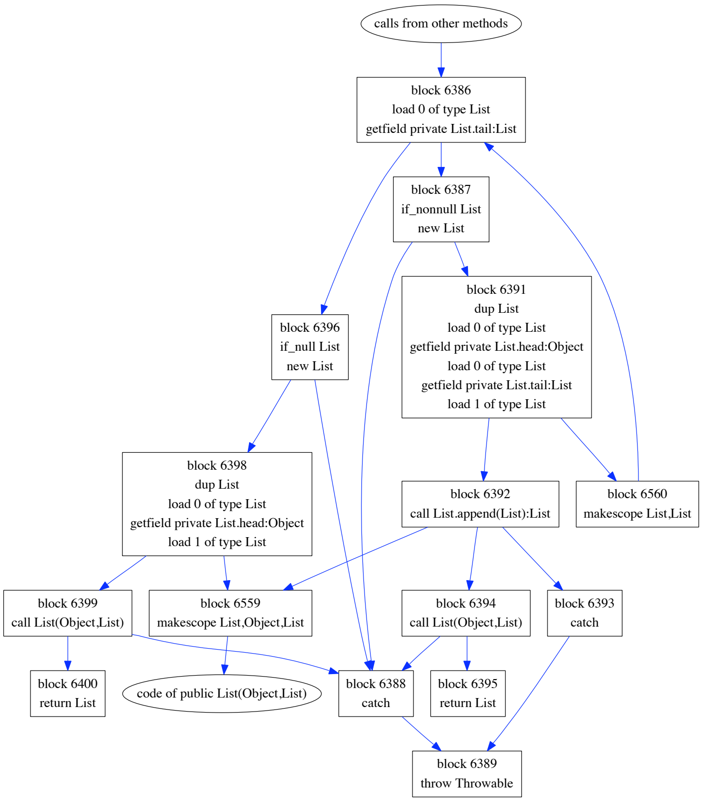

Fig. 2. The basic blocks for the method `append` in Figure 1.

than [3], where binary clauses express size-change graphs, although a more limited form of constraints is used there. Each local variable or stack element v in the bytecode program induces an input variable iv and an output variable ov in the *CLP* program. The sequence $\boldsymbol{i}$ consists of only input variables and $\boldsymbol{o}$ of only output variables. For each clause in the program, we refer to three sets of variables V, I and O; they are the sets of bytecode variables, induced input variable and induced output variables, respectively.

Definition 3.1 (Valuation) *A* valuation θ *is a map from a finite set of variables into integers. Let* $\boldsymbol{v} = v_1 v_2 \cdots v_k$ *be a sequence of variables and* $\boldsymbol{val} = val_1 val_2 \cdots val_k \in \mathbb{Z}^k$. *We write* $[v_1 \mapsto val_1, \ldots, v_k \mapsto val_k]$ *or* $[\boldsymbol{v} \mapsto \boldsymbol{val}]$ *for* *the valuation θ which is such that $\theta(v_i) = val_i$ for all $i = 1, \ldots, k$ and is undefined elsewhere. Let c be a constraint; then $c\theta$ is c where each variable v is replaced by $\theta(v)$. This notation is extended to any syntactical object, such as sequences of variables and predicates. The valuation θ is a* solution *of c if $c\theta$ is equivalent to true. Let p be a predicate; then $c[p(\boldsymbol{v}) \mapsto p(\boldsymbol{val})]$ stands for $c[\boldsymbol{v} \mapsto \boldsymbol{val}]$.* □

We define now the operational semantics for *CLP* over linear integer constraints. It expresses the fact that variables stand for the path-length of concrete data structures in the memory of the system and hence can be undefined but not free, in the sense of logic programming.

Definition 3.2 (Operational Semantics of our *CLP* Language) *Let p, q be predicates and $\boldsymbol{m}, \boldsymbol{n} \in \mathbb{Z}^*$. We say that $q(\boldsymbol{n})$* is derived from *$p(\boldsymbol{m})$ using clause $C = (p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o}))$, written $p(\boldsymbol{m}) \to^C q(\boldsymbol{n})$, if there is a solution θ of $c[\boldsymbol{i} \mapsto \boldsymbol{m}]$ such that $q(\boldsymbol{n}) = q(\boldsymbol{o})\theta$. Clause C in $p(\boldsymbol{m}) \to^C q(\boldsymbol{n})$ is often omitted unless necessary. A* derivation *of $p_0(\boldsymbol{n}_0)$ is $p_0(\boldsymbol{n}_0) \to p_1(\boldsymbol{n}_1) \to \cdots \to p_k(\boldsymbol{n}_k)$ such that $p_{i+1}(\boldsymbol{n}_{i+1})$ is derived from $p_i(\boldsymbol{n}_i)$ for all $0 \le i < k$. A* resolution *is a maximal derivation.* □

The above operational semantics lets us formalise the notion of *termination*. It uses a partition of the predicates of the program in strongly-connected components. Namely, for every clause $p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o})$, we let $p \le q$. Then predicates p_0 and p_1 belong to the same strongly-connected component if and only if $p_0 \le^* p_1$ and $p_1 \le^* p_0$ where $\le^*$ is the reflexive and transitive closure of $\le$. This means that they are part of the same loop. A predicate q is an *entry* if it occurs in a clause $q(\boldsymbol{n}) \text{ :- } c, s(\boldsymbol{m})$ with q and s in the same strongly-connected component (*i.e.*, in a loop) and also in a clause $t(\boldsymbol{v}) \text{ :- } c, q(\boldsymbol{w})$ with q and t in different strongly-connected components. We assume that entries are named `entry`x. From now on, when we say that a predicate is an entry of a *CLP* program, we mean that its name is `entry`x for some x.

Definition 3.3 (Termination) *An entry p* terminates *in a program P if, for every $\boldsymbol{n} \in \mathbb{Z}^*$, all resolutions of $p(\boldsymbol{n})$ by using the clauses of P, with predicates in the strongly-connected component of p, are finite. Otherwise, p is said to* diverge. *Let P_1 and P_2 be programs. P_1* terminates more *than P_2, and we write $P_1 \sqsupseteq P_2$, if whenever an entry of P_1 terminates in P_1, it also terminates in P_2. They are* termination-equivalent*, and we write $P_1 \equiv P_2$, if P_1 terminates more than P_2 and vice versa.* □

Note that if p is not defined in P then it terminates in P since its derivations have length 1. The notion of *terminating more* entails that a proof of termination for the predicates of P_2 is also a proof of termination for the predicates of P_1.

Definition 3.3 formalizes a *loop-local* termination. This means that an entry terminates if it terminates by using the predicates of the loop where it occurs. This is importamnt to report a feedback to the user about which loop of which method might introduce the non-termination, without considering entries that diverge just because the computation, after execution the loop where the entry occurs, continues into another loop that diverges. Entries can also be used to improve the precision of the analysis by computing *call-patterns* from them to the other blocks [17]. We do not discuss this optimization here.

Next section presents a set of program transformations that simplify a *CLP* program P into a smaller program P_s. It will always be the case that P and P_s are termination-equivalent.

4 Program Simplifications

4.1 Removing clauses outside loops

In graphs such as that in Figure 2, arrows outside loops cannot be executed during a divergent computation, which stays inside the same strongly-connected component of the entry where it is started (Definition 3.3). Hence it seems reasonable to remove clauses that are not part of a loop *i.e.,* such that their head and tail do not belong to the same strongly-connected component of blocks. For instance, only the second clause in (1) is generated.

The following result formalizes of well-known technique used in many termination analyzers. It allows us to prove termination for the loops of the program. A clause $p(\boldsymbol{n})$:- $c, q(\boldsymbol{m})$ *occurs in a loop* if p and q are inside the same strongly-connected component of predicates.

Proposition 4.1 (Correctness of clauses outside loops removal) *Let P be a program and P_s be the same program deprived of those clauses that do not occur in a loop. Then $P \equiv P_s$.* $\square$

Proof: Programs P and P_s have the same set of entries. Let q be an entry. If q terminates in P then it terminates in P_s since the latter has less clauses than P. If q diverges in P then there is an infinite derivation using only predicates inside the strongly-connected component of q. Hence only clauses in P_s are used by that derivation, so that q diverges in P_s. $\square$

If we apply this simplification to the CLP program derived from the Java program in Figure 1, the number of clauses decreases from 297 to 12 and the time needed to prove all the entries terminating is 2.72 seconds.

Because of this simplification, from now on we assume that each predicate is only used in its strongly-connected component. Hence termination according to Definition 3.3 corresponds, from now on, to termination by using all the clauses of the program.

4.2 Removing clauses by unfolding

If a program contains clauses $p(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ and $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$, we can *unfold* them into the clause $p(\boldsymbol{m})$:- $c_1 \wedge c_2 \wedge \boldsymbol{n} = \boldsymbol{v}, s(\boldsymbol{w})$ (we assume without loss of generality that clauses are renamed so that they do not share variable). If this is done systematically, for all occurrences of q on the right of the clauses of P, and later the clauses defining q are removed, we say that we *unfold q away from P.*

The result is a program with less predicates but potentially more clauses than P. However, subsequent simplifications will usually remove most of them, so that this simplification is useful in practice.

Proposition 4.2 (Correctness of unfolding away of a predicate) *Let P be a program and q a predicate, not an entry, in P with no clause of the form $q(\boldsymbol{n})$:- $c, q(\boldsymbol{m})$. Let P_s be P where q has been unfolded away. Then $P \equiv P_s$.* □

Proof: Programs P and P_s have the same set of entries. Let p be an entry of P_s. If p diverges in P_s then there is an infinite derivation d for p in P_s. Some steps of this derivation might use clauses derived from unfolding together clauses $r(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ and $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$. We can replace those steps in d with two steps using those two clauses instead. The result is an infinite derivation for p that uses clauses of P. Hence p diverges in P. Conversely, if p diverges in P then there is an infinite derivation d for p in P. If a clause such as $r(\boldsymbol{m})$:- $c_1, q(\boldsymbol{n})$ is used during that derivation, then the subsequent step must use a clause of the form $q(\boldsymbol{v})$:- $c_2, s(\boldsymbol{w})$. Hence those two steps can be merged in d into a unique step that uses the unfolded clause $r(\boldsymbol{m})$:- $c_1 \wedge c_2 \wedge \boldsymbol{n} = \boldsymbol{v}, s(\boldsymbol{w})$. The resulting infinite derivation does not refer to q anymore and uses clauses in P_s. Hence p diverges in P_s. □

Note that Proposition 4.2 does not allow us to unfold away the entries to loops, whose termination is used to tell if each given loop terminates.

If we apply this simplification to the CLP program obtained at the end of Subsection 4.1, the number of clauses decreases from 12 to 8 and the time needed to prove all the entries terminating goes down from 2.72 to 1.48 seconds (including the time for unfolding).

4.3 Removing unsupported or subsumed clauses

By removing unsupported clauses *i.e.,* clauses that call undefined predicates, we maintain the termination-equivalence of programs, since unsupported clauses cannot be used to build an infinite derivation.

Example 4.3 *Let $P = \{C_1, C_2, C_3\}$ with $C_1 = (entry1(ix) := ix = ox, q(ox))$, $C_2 = (q(ix)$:- $ix = ox{+}1, entry1(ox))$ and $C_3 = (q(ix)$:- $ix \geq ox, r(ox))$. Predicate r is not defined in P and hence clause C_3 is unsupported. Thus P is termination-equivalent to $P' = \{C_1, C_2\}$.* □

Proposition 4.4 (Correctness of unsupported clause removal) *Let P be a program and P_s be P deprived of unsupported clauses. Then $P \equiv P_s$.* □
Proof: Any divergent resolution in P_s is also a divergent resolution in P since P_s has less clauses than P. Any divergent resolution in P is also a divergent resolution in P_s since a divergent resolution in P cannot use any unsupported clause, or otherwise it would be finite. □

Another simplification consists in removing subsumed clauses (see also [8]). Let for instance $C_1 = (p(\boldsymbol{i})$:- $c_1, q(\boldsymbol{o}))$ and $C_2 = (p(\boldsymbol{i})$:- $c_2, q(\boldsymbol{o}))$. We say that C_2 *subsumes* C_1 iff $c_1 \models c_2$ (c_1 entails c_2). Note that C_1 and C_2 only differ in the constraint part.

Example 4.5 *The program obtained at the end of Subsection 4.2 contains clauses*

```
entry3899(IL0):-OL0 >= 0,IL0 - OL0 >= 1,IL0 >= 2,entry3899(OL0).
entry3899(IL0):-OL0 >= 0,IL0 - OL0 >= 2,entry3899(OL0).
```

The second clause subsumes the first which can hence be removed. □

Proposition 4.6 (Correctness of subsumed clause removal) *Let P be a program and P_s be P deprived of subsumed clauses. Then $P \equiv P_s$.* □

Proof: Any divergent resolution in P_s is also a divergent resolution in P since P_s has less clauses than P. Hence it is enough to prove that for any divergent resolution in P there is a divergent resolution in P_s. To that purpose, we prove that if $C_1 = (p(\boldsymbol{i}) \text{ :- } c_1, q(\boldsymbol{o}))$ is subsumed by $C_2 = (p(\boldsymbol{i}) \text{ :- } c_2, q(\boldsymbol{o}))$ then $p(\boldsymbol{m}) \to^{C_1} q(\boldsymbol{n})$ implies $p(\boldsymbol{m}) \to^{C_2} q(\boldsymbol{n})$ for any $p, q, \boldsymbol{m}$ and $\boldsymbol{n}$, which entails that any derivation step using C_1 can be replicated by using C_2. Assume hence that $p(\boldsymbol{m}) \to^{C_1} q(\boldsymbol{n})$. Then there is a solution θ of $c_1[\boldsymbol{i} \mapsto \boldsymbol{m}]$ such that $\boldsymbol{n} = \boldsymbol{o}\theta$. Since $c_1 \models c_2$, θ is also a solution of $c_2[\boldsymbol{i} \mapsto \boldsymbol{m}]$ and hence $p(\boldsymbol{m}) \to^{C_2} q(\boldsymbol{n})$. □

If we apply these simplifications to the CLP program obtained at the end of Subsection 4.2, the number of clauses decreases from 8 to 7 and the time needed to prove all the entries terminating goes down from 1.48 to 1.25 seconds (including the time to apply all the simplifications discussed up to now).

4.4 Removing variables

By removing an argument from the clauses of a CLP program, the time needed to build a termination proof of the program decreases, since less arguments means less variables in the data structure implementing the linear constraints and hence better efficiency. Moreover, by removing variables there are chances that distinct clauses get merged because one subsumes another (Subsection 4.3).

Let c be a constraint and let $c^v = \exists_{-\{iv,ov\}}.c$ and $c^{-v} = \exists_{\{iv,ov\}}.c$. The constraint c^v is the *v-dedicated part* of c since it constrains variables iv and ov only; the constraint c^{-v} is the *v-independent part* of c since it does not constrain iv nor ov but only the other variables. Let us define an operation that removes a variable from a predicate, thus reducing its arity:

$$p(iv_1, \ldots, iv_n) \ominus v = \begin{cases} p(iv_1, \ldots, iv_{i-1}, iv_{i+1}, \ldots, iv_n) & \text{if } v \equiv v_i \\ p(iv_1, \ldots, iv_n) & \text{otherwise.} \end{cases}$$

Let us define $p(ov_1, \ldots, ov_n) \ominus v$ similarly. The transformation

$$Comp^{-v} = \{p(\boldsymbol{i}) \ominus v \text{ :- } c^{-v}, q(\boldsymbol{o}) \ominus v \mid p(\boldsymbol{i}) \text{ :- } c, q(\boldsymbol{o}) \in Comp\}$$

removes v from a strongly-connected component $Comp$.

Removal of a variable from a strongly-connected component preserves divergent entries but might introduce more divergent entries.

Proposition 4.7 *Let p_0 be an entry diverging in Comp. Then p_0 also diverges in $Comp^{-v}$.* $\square$

Proof: Since p_0 diverges in *Comp*, there is an infinite resolution

$$p_0(\boldsymbol{n}_0) \to p_1(\boldsymbol{n}_1) \to p_2(\boldsymbol{n}_2) \to \cdots \to p_k(\boldsymbol{n}_k) \to \cdots$$

with $p_j(\boldsymbol{i}_j)$:- $c_j, p_{j+1}(\boldsymbol{o}_j) \in Comp$, $p_{j+1}(\boldsymbol{n}_{j+1}) = p_{j+1}(\boldsymbol{o}_j)\theta_j$ and θ_j solution of $c_j[\boldsymbol{i}_j \mapsto \boldsymbol{n}_j]$. Hence $p_j(\boldsymbol{i}_j) \ominus v$:- $c_j^{-v}, p_{j+1}(\boldsymbol{o}_j) \ominus v \in Comp^{-v}$ and θ_j is a solution of $c_j^{-v}[\boldsymbol{i}_j \mapsto \boldsymbol{n}_j]$ since $c_j \models c_j^{-v}$. Then θ_j is a solution of $c_j^{-v}[p_j(\boldsymbol{i}_j) \ominus v \mapsto p_j(\boldsymbol{n}_j) \ominus v]$ since c_j^{-v} is v-independent. Thus,

$$(p_{j+1}(\boldsymbol{o}_j) \ominus v)\theta_j = p_{j+1}(\boldsymbol{o}_j)\theta_j \ominus v = p_{j+1}(\boldsymbol{n}_{j+1}) \ominus v$$

and we can build the following infinite resolution of $p_0(\boldsymbol{n}_0) \ominus v$ in $Comp^{-v}$

$$p_0(\boldsymbol{n}_0) \ominus v \to p_1(\boldsymbol{n}_1) \ominus v \to p_2(\boldsymbol{n}_2) \ominus v \to \cdots \to p_k(\boldsymbol{n}_k) \ominus v \to \cdots$$

so that p_0 diverges in $Comp^{-v}$. $\square$

In general, *Comp* is not termination-equivalent to $Comp^{-v}$.

Example 4.8 *Consider the strongly-connected component*

$$Comp = \left\{ \begin{array}{c} entry1(ix, iy) \text{ :- } ix \geq 0, oy = ix, ox = iy, q(ox, oy) \\ q(ix, iy) \text{ :- } ox = iy - 1, oy = ix, entry1(ox, oy) \end{array} \right\}$$

The entry entry1 terminates in Comp since the value of x decreases in every two other step and is bounded from below by 0. By removing x from Comp we get

$$Comp^{-x} = \left\{ \begin{array}{l} entry1(iy) \text{ :- } true, q(oy) \\ q(iy) \text{ :- } true, entry1(oy) \end{array} \right\}$$

Now entry1 does not terminate in $Comp^{-x}$. $\square$

The following subsections identify special cases when removal of a variable maintains the termination-equivalence. A common condition is that the variable is *isolated* from other variables.

Definition 4.9 *A variable v is* isolated *in a strongly-connected component Comp if, for every clause $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$, we have $c = c^v \wedge c^{-v}$.* $\square$

Example 4.10 *Neither x nor y is isolated in the component Comp of Example 4.8. Instead, both x and y are isolated in the component*

$$Comp = \left\{ \begin{array}{c} entry1(ix, iy) \text{ :- } ix \geq 0, ox = ix, oy = iy - 1, q(ox, oy) \\ q(ix, iy) \text{ :- } ox = ix - 1, oy = iy, entry1(ox, oy) \end{array} \right\}$$

$\square$

4.5 Removing right-open/left-open variables

In this subsection we show a first example of a removal of variables for which the converse of Proposition 4.7 holds.

Definition 4.11 (Right or left-open variable) *An isolated variable v in a strongly-connected component Comp is* right-open *if, for every $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$, we have that c^v is either true or $iv = ov$, or $ov \geq const$, $ov = const$ or $ov \leq const$ (or equivalent) , where const is an integer constant.* Left-openness *is defined analogously by switching ov with iv in the definition of right-openness.* □

Example 4.12 *The program obtained at the end of Subsection 4.3 contains the component*

```
entry3880(IL0,IL1):-IL1 - OL1 = 0,OL0 >= 0,IL0 >= 2,IL0 - OL0 >= 1,
                    entry3880(OL0,OL1).
```

where variable `L1` is both left- and right-open and can hence be removed obtaining the component

```
entry3880(IL0):-OL0 >= 0,IL0 >= 2,IL0 - OL0 >= 1,entry3880(OL0).
```

`L1` would still be left-open if there were an extra constraint `IL1 >= 3`. It would not be left-open anymore if there were also an extra constraint `OL1 >= 7`. □

Consider a resolution of $p_0(\boldsymbol{n}_0)$ in a strongly-connected component $Comp$ where v is right-open. Let $p(\boldsymbol{i}_j)$:- $c_j, q(\boldsymbol{o}_j)$ be the clause used at the j^{th} resolution step. If c^v_j is $iv = ov$ then the j^{th} step simply copies the value of v from p_j to p_{j+1}. Otherwise, the value of v in p_j is not related to that in p_{j+1}: any value satisfying the v-dedicated part c^v_j of c_j may be picked up for v in p_{j+1}; such a value exists always due to the limited form of c^v_j. This means that v does not contribute to the termination of the predicates in $Comp$ and can hence be removed. This is formally proved below.

Proposition 4.13 (Correctness of left- or right-open variable removal) *Let v be right- or left-open in a strongly-connected component Comp. If an entry diverges in $Comp^{-v}$ then it diverges in Comp.* □

Proof: We only prove the case when v is right-open. The case when v is left-open is symmetrical. Let hence p_0 be a divergent entry in $Comp^{-v}$. Then there is $\boldsymbol{m}_0 \in \boldsymbol{Z}$ and an infinite resolution of $p_0(\boldsymbol{m}_0)$ in $Comp^{-v}$, which we write as

$$d_0 \to^{C_0} d_1 \to^{C_1} d_2 \to^{C_2} \cdots \to d_\ell \to^{C_\ell} d_{\ell+1} \cdots$$

where every clause $p(\boldsymbol{i}) \ominus v$:- $c^{-v}, q(\boldsymbol{o}) \ominus v$ used in each portion d_ℓ, for $\ell \geq 0$, is obtained from a clause $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in Comp$ with $c^v = (iv = ov)$ and each C_ℓ is obtained from a clause $C'_\ell = (p_\ell(\boldsymbol{i}_\ell)$:- $c_\ell, q_\ell(\boldsymbol{o}_\ell)) \in Comp$ with c^v_ℓ different from $iv = ov$. Let $x_\ell \in \mathbb{Z}$ be such that, for every $\ell > 0$, $\{ov \mapsto x_{\ell+1}\}$ is a solution of c^v_ℓ (hence x_0 is completely free). Let $p(\boldsymbol{m})$ be a call in $Comp^{-v}$ and $x \in \mathbb{Z}$. Then we define $p(\boldsymbol{m}) \oplus_v [x]$ as the call in $Comp$ obtained from $p(\boldsymbol{m})$ by putting x at the position for v in the predicate p of $Comp$. It suffices to prove that there is an infinite resolution of $p_0(\boldsymbol{m}_0) \oplus_v [x_0]$ in *Comp*. Assume that

$$d_\ell = (p_{\ell,0}(\boldsymbol{m}_{\ell,0}) \rightarrow \cdots p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) \cdots \rightarrow p_{\ell,f_\ell}(\boldsymbol{m}_{\ell,f_\ell}))$$

with $p_{0,0} = p_0$ and $\boldsymbol{m}_{0,0} = \boldsymbol{m}_0$. Let $p_{\ell,j}(\boldsymbol{n}_{\ell,j}) = p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \oplus_v [x_\ell]$ for each $0 \leq j \leq f_\ell$. Since $p_{\ell,j}(\boldsymbol{m}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1})$, there is $p_{\ell,j}(\boldsymbol{i})$:- $(iv = ov) \wedge c, p_{\ell,j+1}(\boldsymbol{o}) \in Comp$ such that $p_{\ell,j}(\boldsymbol{i}) \ominus v$:- $c, p_{\ell,j+1}(\boldsymbol{o}) \ominus v \in Comp^{-v}$ and there is a solution θ of $c[p_{\ell,j}(\boldsymbol{i}) \ominus v \mapsto p_{\ell,j}(\boldsymbol{m}_{\ell,j})]$ such that $p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) = (p_{\ell,j+1}(\boldsymbol{o}) \ominus v)\theta$. Since c is v-independent, $\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_\ell\}$ is a solution of $(iv = ov) \wedge c[\boldsymbol{i} \mapsto \boldsymbol{n}_{\ell,j}]$ and $(\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_\ell\})(p_{\ell,j+1}(\boldsymbol{o})) = (p_{\ell,j+1}(\boldsymbol{o}) \ominus v)\theta \oplus_v [x_\ell] = p_{\ell,j+1}(\boldsymbol{m}_{\ell,j+1}) \oplus_v [x_\ell] = p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1})$. Thus, $p_{\ell,j}(\boldsymbol{n}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1})$ for $0 \leq j \leq f_\ell - 1$ and

$$d'_\ell = (p_{\ell,0}(\boldsymbol{n}_{\ell,0}) \rightarrow \cdots p_{\ell,j}(\boldsymbol{n}_{\ell,j}) \rightarrow p_{\ell,j+1}(\boldsymbol{n}_{\ell,j+1}) \cdots \rightarrow p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}))$$

is a derivation in *Comp*. We show now that $p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}) \rightarrow^{C'_\ell} p_{\ell+1,0}(\boldsymbol{n}_{\ell+1,0})$ so that we obtain an infinite resolution $d'_0 \rightarrow d'_1 \rightarrow \cdots d'_\ell \rightarrow d'_{\ell+1} \rightarrow \cdots$ in *Comp*. Since $p_{\ell,f_\ell}(\boldsymbol{m}_{\ell,f_\ell}) \rightarrow^{C_\ell} p_{\ell+1,0}(\boldsymbol{m}_{\ell+1,0})$, we know that $p_\ell = p_{\ell,f_\ell}$, $q_\ell = p_{\ell+1,0}$ and there is a solution θ of $c_\ell^{-v}[p_\ell(\boldsymbol{i}_\ell) \ominus v \mapsto \boldsymbol{m}_{\ell,f_\ell}]$ such that $q_\ell(\boldsymbol{m}_{\ell+1,0}) = (q_\ell(\boldsymbol{o}_\ell) \ominus v)\theta$. Then $\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_{\ell+1}\}$ is a solution of $c_\ell^{-v}[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$, since c_ℓ^{-v} is v-independent, and it is also a solution of $c_\ell^{v}[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$ and hence of $c_\ell[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}]$ since $c_\ell^{v}[\boldsymbol{i}_\ell \mapsto \boldsymbol{n}_{\ell,f_\ell}] = c_\ell^{v}[iv \mapsto x_\ell] = c_\ell^{v}$ and c_ℓ^{v} contains only ov and $\{ov \mapsto x_{\ell+1}\}$ is a solution of c_ℓ^{v}. Also, $q_\ell(\boldsymbol{o}_\ell)(\theta \cup \{iv \mapsto x_\ell, ov \mapsto x_{\ell+1}\}) = (q_\ell(\boldsymbol{o}_\ell) \ominus v)\theta \oplus_v [x_{\ell+1}] = q_\ell(\boldsymbol{m}_{\ell+1,0}) \oplus_v [x_{\ell+1}] = q_\ell(\boldsymbol{n}_{\ell+1,0})$. Hence $p_{\ell,f_\ell}(\boldsymbol{n}_{\ell,f_\ell}) \rightarrow^{C'_\ell} p_{\ell+1,0}(\boldsymbol{n}_{\ell+1,0})$. □

If we apply this simplification to the CLP program obtained at the end of Subsection 4.3, the number of clauses goes down from 7 to 6 (because of entailment checks) and there are less arguments in predicates. The time needed to prove all the entries terminating goes down from 1.25 to 1.02 seconds (including the time to apply all the simplifications discussed up to now).

4.6 Removing uniform variables

Even if an isolated variable is neither left-open nor right-open, it can still be removed when there is a fixed value that can be put in that variable throughout an infinite resolution. Such a variable is called *uniform*.

Definition 4.14 (Uniform variable) *An isolated variable v is* uniform *in a strongly-connected component Comp if there is $x \in \mathbb{Z}$ such that, for every $p(\boldsymbol{i})$:- $c, q(\boldsymbol{o}) \in$ Comp, the valuation $\{iv \mapsto x, ov \mapsto x\}$ is a solution of c^v (note that c^v may contain more than one constraint).* □

Example 4.15 *The program obtained at the end of Subsection 4.5 contains the component:*

```
block3853(IL0,IL1,IL2):-IL2 - OL2 = -1,IL1 - OL1 = 0,IL0 - OL0 = 0,
                        IL1 - IL2 >= 1,block3853(OL0,OL1,OL2).
block3853(IL0,IL1,IL2):-IL2 - OL2 = -1,IL1 - OL1 = 0,OL0 = 1,
                        IL1 - IL2 >= 2,entry3849(OL0,OL1,OL2).
```

```
entry3849(IL0,IL1,IL2):-IL2 - OL2 = 0,IL1 - OL1 = 0,IL0 - OL0 = 0,
                        IL0 >= 1,block3853(OL0,OL1,OL2).
```

By taking $x = 1$, we conclude that `L0` *is uniform.* □

Example 4.16 *Uniform variables and left- or right-open variables are different concepts. For instance, variable* `L0` *is uniform in the component of Example 4.15 but it is not left-open nor right-open. Conversely, variable x is left-open in the component*

$$entry1(ix, iy) \text{ :- } iy \geq 0, ox = ix, ix \geq 3, oy = iy - 1, p(ox, oy)$$
$$p(ix, iy) \text{ :- } ox = ix, ix \leq 0, oy = iy, entry1(ox, oy)$$

but it is not uniform there. □

This proposition justifies the removal of a uniform variable from a strongly-connected component.

Proposition 4.17 (Correctness of a uniform variable removal) *Let a variable v be uniform in a strongly-connected component $Comp$. If an entry diverges in $Comp^{-v}$ then it diverges $Comp$.* □

Proof: Let $x \in \mathbb{Z}$ as in Definition 4.14. From an infinite resolution in $Comp^{-v}$, we can construct an infinite resolution in $Comp$ by simply inserting x into each call at the position of variable v. □

If we apply this simplification to the CLP program obtained at the end of Subsection 4.5, the number of clauses remains 6 but there are less arguments in predicates. The time needed to prove all the entries terminating goes down from 1.02 to 0.67 seconds (including the time to apply all the simplifications).

5 Experiments

Figure 3 reports the results of our termination analysis and the effects of our simplifications on the time needed to build a proof of termination for the entries of the program. `Ackermann` is an implementation of the traditional Ackermann function. `BubbleSort` is an implementation of the bubblesort algorithm on arrays. `NQueens` is a program that solves the n-queens problem by using a library for binary decision diagrams, included in the analysis. `JLex` is a lexical analyzers generator. `Kitten` is a didactic compiler for simple object-oriented programs. Our experiments have been performed on a Linux machine based on a 64 bits dual core AMD Opteron processor 280 running at 2.4Ghz, with 2 gigabytes of RAM and 1 megabyte of cache, by using Sun Java Development Kit version 1.5 and SICStus Prolog version 3.12.8. For each program, we report the number of methods (without the Java libraries) and the time for building a proof of termination with the original, unlocalized technique of [17] and with the successive application of more and more simplifications, described in this paper (the time for the simplifications is included). The header of each column reports the subsection where the simplification is described. The original technique failed to conclude the analysis after 15 minutes for `NQueens`, `JLex`

program	meth.	original	4.1	4.2	4.3	4.5	4.6
`Ackermann`	5	7.11	0.21	0.21	0.21	0.21	0.21
precision		5	5	5	5	5	5
`BubbleSort`	5	19.07	1.55	0.71	0.71	0.49	0.49
precision		3	4	5	5	5	5
`NQueens`	222	-	210.31	156.32	92.29	47.77	34.34
precision		-	171	171	171	171	171
`JLex`	137	-	228.51	335.85	374.82	121.95	81.21
precision		-	84	87	102	102	102
`Kitten`	947	-	200.39	226.79	152.47	93.70	79.35
precision		-	811	827	827	827	827

Fig. 3. The termination analyses of some programs. Times are in seconds. The second line (*precision*), for each program, reports the number of methods proved to terminate. In the header, we refer to the subsection where the simplification is described.

and `Kitten`. In general, more simplifications means better efficiency. This relation is not always true. For instance, building a proof of termination for `JLex` takes 228.51 seconds if only the simplification of Subsection 4.1 is applied. If also the simplification of Subsection 4.2 is applied, this time increases to 335.85. We explain this behaviour with the fact that simplifications have a cost. Moreover, when the program is too complex, BinTerm uses timeouts, which makes the construction of the proof faster. However, the precision of the proof decreases with the number of timeouts. Hence, below each program, we report the number of methods proved to terminate. This number increases with the number of simplifications applied to the CLP program, since less timeouts are triggered.

6 Conclusion

We have presented techniques for simplifying the CLP programs that are automatically generated during termination analysis of Java bytecode programs. Those techniques are proved to keep the termination-equivalence of the CLP programs. Their application to some real case of analysis shows that they decrease the time for building a proof by some order of magnitude. Moreover, simplified CLP programs induce less timeouts during the construction of the proof of termination, so that our simplification techniques actually induce more precise termination analyses.

In [2], *useless variables* are eliminated from *CLP* programs expressing cost relationships for Java bytecode programs. That technique removes most stack variables. We have verified that almost no stack variable survives after our unfolding of clauses (Subsection 4.2). Our unfolding can be seen as a *CLP* view of the simplification done in [2] from a Java bytecode perspective. On the one hand, as in [2] the elimination of variables is done earlier, all related static analysis benefit from this simplification. On the other hand, note that we have a correctness proof for that simplification and that subsequent simplifications are not related to that in [2].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools.* Addison Wesley Publishing Company, 1986.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In R. L. Wainwright and H. Haddad, editors, *Proc. of the 2008 ACM Symposium on Applied Computing (SAC'08)*, pages 368–375, Fortaleza, Ceara, Brazil, March 2008. ACM.

[3] A. M. Ben-Amram and M. Codish. A SAT-Based Approach to Size Change Termination with Global Ranking Functions. In C. R. Ramakrishnan and J. Rehof, editors, *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 218–232, Budapest, Hungary, 2008. Springer.

[4] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance Analyses from Invariance Analyses. In M. Hofmann and M. Felleisen, editors, *Proc. of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 211–224, Nice, France, January 2007.

[5] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 386–400, Seattle, WA, USA, August 2006. Springer.

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists Are Counter Automata. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531, Seattle, WA, USA, August 2006. Springer.

[7] M. Codish. Proving Termination with (Boolean) Satisfaction. In A. King, editor, *Proc. of the 17th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'07)*, volume 4915 of *Lecture Notes in Computer Science*, pages 1–7, Kongens Lyngby, Denmark, 2007. Springer.

[8] M. Codish and C. Taboch. A Semantics Basis for Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418, Seattle, WA, USA, August 2006. Springer.

[10] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming (TPLP)*, 5(1-2):75–91, 2005.

[11] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In U. Furbach and N. Shankar, editors, *3th International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286, Seattle, WA, USA, August 2006. Springer.

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

[13] A. Loginov, T. W. Reps, and M. Sagiv. Refinement-Based Verification for Possibly-Cyclic Lists. In T. W. Reps, M. Sagiv, and J. Bauer, editors, *Proc. of Theory and Practice of Program Analysis and Compilation, Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 247–272. Springer, 2006.

[14] P. Manolios and D. Vroon. Termination Analysis with Calling Context Graphs. In T. Ball and R. B. Jones, editors, *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414, Seattle, WA, USA, August 2006. Springer.

[15] A. Podelski and A. Rybalchenko. Transition Predicate Abstraction and Fair Termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3), May 2007.

[16] 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`.

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

[18] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *London Mathematical Society*, 42(2):230–265, 1936.