Termination Analysis of Java Bytecode

Elvira Albert[1], Puri Arenas[1], Michael Codish[2],
Samir Genaim[3], Germán Puebla[3], and Damiano Zanardini[3]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CS, Ben-Gurion University of the Negev, Israel
[3] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming, and recently also for imperative languages. In existing approaches, termination is performed on source programs. However, there are many situations, including mobile code, where only the compiled code is available. In this work we present an automatic termination analysis for sequential Java Bytecode programs. Such analysis presents all of the challenges of analyzing a low-level language as well as those introduced by object-oriented languages. Interestingly, given a bytecode program, we produce a *constraint logic* program, CLP, whose termination entails termination of the bytecode program. This allows applying the large body of work in termination of CLP programs to termination of Java bytecode. A prototype analyzer is described and initial experimentation is reported.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination.

In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [15], and logic and functional languages [18, 10, 17]. Termination-proving techniques are also emerging in the imperative paradigm [6, 11, 15], even for dealing with large industrial code [11].

Static analysis of *Java ByteCode* (JBC for short) has received considerable attention lately [25, 23, 24, 22, 1]. The present paper presents a static analysis for sequential JBC which is, to the best of our knowledge, the first approach to proving termination. Bytecode is a low-level representation of a program, designed to be executed by a virtual machine rather than by dedicated hardware. As such, it is usually higher level than actual machine code, and independent of

the specific hardware. This, together with its security features, makes JBC [19] the chosen language for many *mobile code* applications. In this context, analysis of JBC programs may enable performing a certain degree of static (i.e., before execution) verification on program components obtained from untrusted providers. *Proof-Carrying Code* [20] is a promising approach in this area: mobile code is equipped with certain *verifiable evidence* which allows deployers to independently verify properties of interest about the code. Termination analysis is also important since the verification of functional program properties is often split into separately proving partial correctness and termination.

Object-oriented languages in general, and their low-level (bytecode) counterparts in particular, present new challenges to termination analyzers: (1) loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions; (2) *size measures* must consider primitive types, user defined objects, and arrays; and (3) *tracking* data is more difficult, as data can be stored in variables, operand stack elements or heap locations.

Analyzing JBC is a necessity in many situations, including mobile code, where the user only has access to compiled code. Furthermore, it can be argued that analyzing low-level programs can have several advantages over analyzing their high-level (Java) counterparts. One advantage is that low-level languages typically remain stable, as their high-level counterparts continue to evolve — analyzers for bytecode programs need not be enhanced each time a new language construct is introduced. Another advantage is that analyzing low-level code narrows the gap between what is verified and what is actually executed. This is relevant, for example, in safety critical applications.

In this paper we take a semantic-based approach to termination analysis, based on two steps. The first step transforms the bytecode into a *rule-based* program where all loops and all variables are represented uniformly, and which is semantically equivalent to the bytecode. This rule-based representation is based on previous work [1] in cost analysis, and is presented in Sec. 2. In the second step (Sec. 3), we adapt directly to the rule-based program standard techniques which usually prove termination of high-level languages. Sec. 4 reports on our prototype implementation and validates it by proving termination of a series of object-oriented benchmarks, containing recursion, nested loops and data structures such as trees and arrays. Conclusions and related work are presented in Sec. 5.

2 Java Bytecode and its Rule-Based Representation

We consider a subset of the Java Virtual Machine (JVM) language which handles integers and object creation and manipulation (by accessing fields and calling methods). For simplicity, exceptions, arrays, interfaces, and primitive types besides integers are omitted. Yet, these features can be easily handled within our setting: all of them are implemented in our prototype and included in benchmarks in Table 1. A full description of the JVM [19] is out of the scope of this paper.

A sequential JBC program consists of a set of *class files*, one for each class, partially ordered with respect to the subclass relation $\preceq$. A class file contains information about its name, the class it extends, and the fields and methods it defines. Each method has a unique signature m from which we can obtain the class, denoted $class(m)$, where the method is defined, the name of the method, and its signature. When it is clear from the context, we ignore the class and the types parts of the signature. The bytecode associated with m is a sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n \rangle$, where each b_i is a *bytecode instruction*, and pc_i is its address. The local variables of a method are denoted by $\langle l_0, \ldots, l_{n-1} \rangle$, of which the first $k{\leq}n$ are the formal parameters, and l_0 corresponds to the *this* reference (unlike Java, in JBC, the *this* reference is explicit). Similarly, each field f has a unique signature, from which we can obtain its name and the name of the class it belongs to. The bytecode instructions we consider include:

$$
\begin{array}{rl}
bcInst ::= & \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ i \mid \mathsf{aconst_null} \\
\mid & \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{iinc}\ v\ n \mid \mathsf{imul} \mid \mathsf{idiv} \\
\mid & \mathsf{if_}\phi\ pc \mid \mathsf{goto}\ pc \mid \mathsf{ireturn} \mid \mathsf{areturn} \\
\mid & \mathsf{new}\ c \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokespecial}\ m \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f
\end{array}
$$

where c is a class, ϕ is a comparison condition on numbers (ne, le, icmpgt) or references (null, nonnull), v is a local variable, i is an integer, and pc is an instruction address. Briefly, instructions are: (row 1) stack operations referring to constants and local variables; (row 2) arithmetic operations; (row 3) jumps and method return; and (row 4) object-oriented instructions. All instructions in row 3, together with invokevirtual, are *branching* (the others are *sequential*). For simplicity, we will assume all methods to return a value. Fig. 1 depicts the bytecode for the iterative method *fact*, where indexes $0, \ldots, 3$ stands for local variables *this*, n, *ft* and i respectively. $next(pc)$ is the address immediately after the program counter pc. As instructions have different sizes, addresses do not always increase by one (e.g., $next(6){=}9$).

We assume an operational semantics which is a subset of the JVM specification [19]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects (and arrays) allocated in the memory. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same object in the heap.

2.1 From Bytecode to Control Flow Graphs

The JVM language is unstructured. It allows conditional and unconditional jumps as well as other implicit sources of branching, such as virtual method invocation and exception throwing. The notion of a *Control Flow Graph* (CFG for short) is a well-known instrument which facilitates reasoning about programs

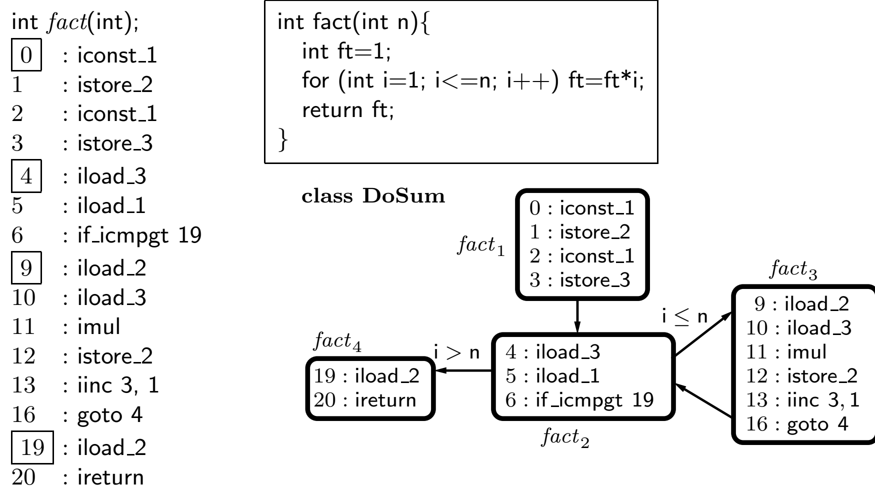

Fig. 1. A JBC method (left) with its corresponding source (center) and its CFG (right)

in unstructured languages. A CFG is similar to the older notion of a flow chart, but CFGs include a concept of "call to" and "return from". Methods in the bytecode program are represented as CFGs, and calls from one method to another correspond to calls between these graphs. In order to build CFGs, the first step is to partition a sequence of bytecode instructions into a set of maximal subsequences, or basic blocks, of instructions which execute sequentially, i.e., without branching nor jumping. Given a bytecode instruction $pc{:}b$, we say that $pc'{:}b'$ is a *predecessor* of $pc{:}bc$ if one of the following conditions holds: (1) b'=goto pc, (2) b'=if_ϕ pc, (3) $next(pc')=pc$.

Definition 1 (partition to basic blocks). *Given a method m and its sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n\rangle$, a partition into basic blocks $m_1, \ldots, m_k$ takes the form*

$$\underbrace{pc_{i_1}{:}b_{i_1}, \ldots, pc_{f_1}{:}b_{f_1}}_{m_1}, \underbrace{pc_{i_2}{:}b_{i_2} \ldots, pc_{f_2}{:}b_{f_2}}_{m_2}, \ldots \underbrace{pc_{i_k}{:}b_{i_k} \ldots, pc_{f_k}{:}b_{f_k}}_{m_k}$$

where i_1=1, f_k=n and

1. *the number of basic blocks (i.e. k) is minimal;*
2. *in each basic block m_j, only the instruction b_{f_j} can be branching; and*
3. *in each basic block m_j, only the instruction b_{i_j} can have more than one predecessor.*

A partition to basic blocks can be obtained as follows: the first sequence m_1 starts at pc_1 and ends at $pc_{f_1}{=}min(pc_{e_1}, pc_{s_1})$, where pc_{e_1} is the address of the first branching instruction after pc_1, and pc_{s_1} is the first address after pc_1

s.t. the instruction at address $next(pc_{s_1})$ has more than one predecessor. The sequence m_2 is computed similarly starting at pc_{i_2}=$next(pc_{f_1})$, etc. Note that this partition can be computed in two passes: the first computes the predecessors, and the second defines the beginning and end of each sub-sequence.

Example 1. The JBC *fact* method on the left of Fig. 1 is partitioned into four basic blocks. The initial addresses (pc_{i_x}) of these blocks are shown within boxes. Each block is labeled by $fact_{id}$ where *id* is a unique block identifier. A directed edge indicates a control flow from the last instruction in the source node to the first instruction in the destination node. Edges may be labeled by a guard which states conditions under which the edge may be traversed during execution. □

In *invokevirtual*, due to dynamic dispatching, the actual method to be called may not be known at compile time. To facilitate termination analysis, we capture this information and introduce it explicitly in the CFG. This is done by adding new blocks, the *implicit basic blocks*, containing calls to *actual methods* which might be called at runtime. Moreover, access to these blocks is guarded by mutually exclusive conditions on the runtime class of the calling object.

Definition 2 (implicit basic block). *Let m be a method which contains an instruction of the form pc:b, where b=*invokevirtual *m'. Let M be a superset of the methods (signatures) that might actually be called at runtime when executing pc:b. The implicit basic block for $m''{\in}M$ is $m_{pc:c}$, where c=$class(m'')$ if $class(m'') \preceq class(m')$, otherwise c=$class(m')$. The block includes the single special instruction* invoke(m')*. The guard of $m_{pc:c}$ is $m^g_{pc:c}$=*instanceof(n, c, D)*, where D=$\{class(m'') \mid m''{\in}M,\ class(m'') \prec c\}$, and n is the arity of m'.*

It can be seen that m is used to denote both methods and blocks in order to make them globally unique. The above condition instanceof(n, c, D) states that the $(n{+}1)$th stack element (from the top) is an instance of class c and not an instance of any class in D. Computing the set M in the above definition can be statically done by considering the class hierarchy and the method signature, which is clearly a safe approximation of the set of the actual methods that might be called at runtime when executing b. However, in some cases this might result in a much larger set than the actual one, which in turn affects the precision and performance of the corresponding static analysis. In such cases, class analysis [25] is usually applied to reduce this set as it gives information about the possible runtime classes of the object whose method is being called. Note that the instruction invoke(m') does not appear in the original bytecode, but it is instrumental to define our rule-based representation in Sec. 2.2. For example, consider the CFG in Fig. 2, which corresponds to the recursive method *doSum* and calls *fact*. This CFG contains two implicit blocks labeled $doSum_{11:DoSum}$ and $doSum_{19:DoSum}$.

The following definition formalizes the notion of a CFG for a method. Although the invokespecial bytecode instruction always corresponds to only one possible method call which can be identified from the symbolic method reference, in order to simplify the presentation, we treat it as invokevirtual, and associate it to a single implicit basic block with the *true* guard. Note that every bytecode

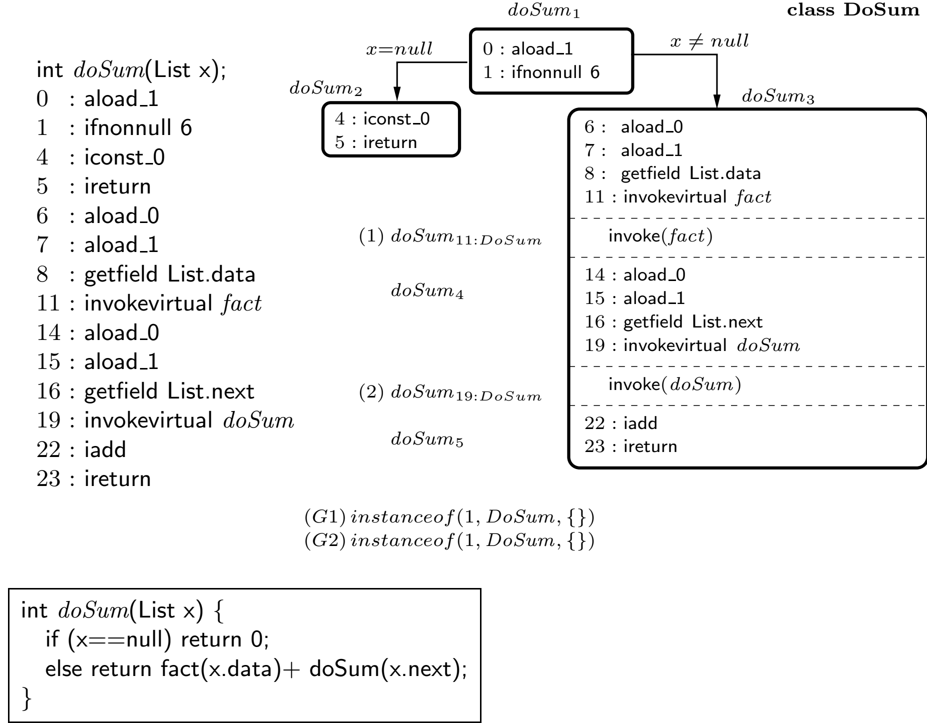

Fig. 2. The Control Flow Graph of the *doSum* example

instruction belongs to exactly one basic block. By $BlockId(pc, m)=i$ we denote the fact that the instruction pc in m belongs to block m_i. In addition, for a given *invokevirtual* instruction $pc{:}b$ in a method m, we use M^m_{pc} and G^m_{pc} to denote the set of its *implicit basic blocks* and their corresponding guards respectively.

Definition 3 (CFG). *The* control flow graph *for a method m is a graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$. Nodes $\mathcal{N}$ consist of:*

(a) basic blocks $m_1, \ldots, m_k$ of m; and
(b) implicit basic blocks corresponding to calls to methods.

Edges in $\mathcal{E}$ take the form $\langle m_i \rightarrow m_j, condition_{ij} \rangle$ where m_i and m_j are, resp., the source *and* destination *node, and $condition_{ij}$ is the Boolean condition labeling this transition. The set of edges is constructed, by considering each node $m_i \in \mathcal{N}$ which corresponds to a (non-implicit) basic block, whose last instruction is denoted as $pc{:}b$, as follows:*

1. *if b=goto pc' and $j=BlockId(pc', m)$ then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$;*
2. *if b=if_ϕ pc', $j=BlockId(pc', m)$ and $i'=BlockId(next(pc), m)$ then we add both $\langle m_i \rightarrow m_j, \phi \rangle$ and $\langle m_i \rightarrow m_{i'}, \neg\phi \rangle$ to $\mathcal{E}$;*

3. *if* $b \in \{\text{invokevirtual } m', \text{invokespecial } m'\}$, *and* $i'=BlockId(next(pc), m)$ *then, for all* $d \in M^m_{pc}$ *and its corresponding* $g^m_{pc:d} \in G^m_{pc}$, *we add* $\langle m_i \rightarrow m_{pc:d}, g^m_{pc:d} \rangle$ *and* $\langle m_{pc:d} \rightarrow m_{i'}, true \rangle$ *to* $\mathcal{E}$*;*
4. *otherwise, if* $j=BlockId(next(pc), m)$ *then we add* $\langle m_i \rightarrow m_j, true \rangle$ *to* $\mathcal{E}$*.*

For conciseness, when a branching instruction b involving implicit blocks leads to a single successor block, we include the corresponding invoke instruction within the basic block b belongs to. For instance, consider that the classes *DoSum* and *List* are not extended by any other class. In this case, the branching instructions 11 and 19 have a single continuation. Their associated implicit blocks marked with (1) and (2) in Fig. 2 are, thus, just included within the basic block $doSum_3$. $G1$ and $G2$ at the bottom indicate the guards which should label the edge.

2.2 Rule-Based Representation

The CFG, while having advantages, is not optimal for our purposes. Therefore, we introduce a *Rule-Based Representation* (RBR) on which we demonstrate our approach to termination analysis. This RBR is based on a recursive representation presented in previous work [1], where it has been used for cost analysis.

The main advantages of the RBR are that: (1) all iterative constructs (loops) fit in the same setting, independently of whether they originate from recursive calls or iterative loops (conditional and unconditional jumps); and (2) all variables in the local scope of the method a block corresponds to (formal parameters, local variables, and stack values) are represented uniformly as explicit arguments. This is possible as in JBC the height of the *operand stack* at each program point is statically known. We prefer to use this rule-based representation, rather than other existing ones (e.g., BoogiePL [13] or those in Soot [26]), as in a simple post-processing phase we can eliminate almost all stack variables, which results, as we will see in Sec. 3.1, in a more efficient analysis.

A *Rule-Based Program* (RBP for short) defines a set of *procedures*, each of them defined by one or more rules. As we will see later, each block in the CFG generates one or two procedures. Each rule has the form $head(\bar{x}, \bar{y}){:=}guard, instr, cont$ where *head* is the name of the procedure the rule belongs to, $\bar{x}$ and $\bar{y}$ indicate sequences $\langle x_1, \ldots, x_n \rangle$, $n{>}0$ (resp. $\langle y_1, \ldots, y_k \rangle$, $k{>}0$) of input (resp. output) arguments, *guard* is of the form $\mathsf{guard}(\phi)$, where ϕ is a Boolean condition on the variables in $\bar{x}$, *instr* is a sequence of (decorated) bytecode instructions, and *cont* indicates a possible call to another procedure representing the continuation of this procedure. In principle, $\bar{x}$ includes the method's local variables and the stack elements at the beginning of the block. In most cases, $\bar{y}$ only needs to store the return value of the method, which we denote by r. For simplicity, guards of the form $\mathsf{guard}(true)$ are omitted. When a procedure p is defined by means of several rules, the corresponding guards must cover all cases and be pairwise exclusive.

Decorating Bytecode Instructions. In order to make all arguments explicit, each bytecode instruction in *instr* is *decorated* explicitly with the (local and stack) variables it operates on. We denote by $t{=}stack_height(pc, m)$ the height of the

stack immediately before the program point pc in a method m. Function dec in the following table shows how to *decorate* some selected instructions, where n is the number of arguments of m.

$pc{:}b$	$\mathsf{dec}(b)$
iconst i	$iconst(i, s_{t+1})$
istore v	$istore(s_t, \ell_v)$
iload v	$iload(l_v, s_{t+1})$
new c	$new(c, s_{t+1})$
ireturn	$ireturn(s_t, r)$

$pc{:}b$	$\mathsf{dec}(b)$
iadd	$iadd(s_{t-1}, s_t, s_{t-1})$
invoke(m)	$m(\langle s_{t-n}, \ldots, s_t\rangle, \langle s_{t-n}\rangle)$
getfield f	$getfield(f, s_t, s_t)$
putfield f	$putfield(f, s_{t-1}, s_t, s_{t-1})$
guard$(icmpgt)$	$\mathsf{guard}(icmpgt(s_{t-1}, s_t))$

Guards are translated according to the bytecode instruction they come from. Note that branching instructions do not need to appear in the RBR, since their effect is already captured by the branching at the RBR level and since invoke instructions are replaced by calls to the entry rule of the corresponding method.

Definition 4 (RBR). *Let m be a method with $l_0, \ldots, l_{n-1}$ local variables, of which $l_0, \ldots, l_{k-1}$ are the formal parameters together with the* this *reference l_0 ($k \leq n$), and let $\langle \mathcal{N}, \mathcal{E}\rangle$ be its CFG. The* rule-based representation *of $\langle \mathcal{N}, \mathcal{E}\rangle$ is* $\mathsf{rules}(\langle \mathcal{N}, \mathcal{E}\rangle) = \mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) \bigcup_{m_p \in \mathcal{N}} \mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$*, with:*

$$\mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) = \{m(\langle \ell_0, \ldots, \ell_{k-1}\rangle, \langle r\rangle) {:=} \mathsf{init_local_vars}(\langle l_k, \ldots, l_{n-1}\rangle), m_1(\langle \ell_0, \ldots, \ell_{n-1}\rangle, \langle r\rangle)\}$$

where the call init_local_vars *initializes the local variables of the method, and*

$$\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle) = \begin{cases} \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) {:=} TBC^p_m.\} & \nexists \langle m_p \mapsto _, _\rangle \in \mathcal{E} \\ \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) {:=} TBC^p_m, m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle).\} \bigcup & \\ \quad \{m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle) {:=} g, m_q(\langle \bar{l}, s_0, \ldots, s_{q_i-1}\rangle, \langle r\rangle). & \textit{otherwise} \\ \quad\quad \mid \langle m_p \rightarrow m_q, \phi_q\rangle \in \mathcal{E} \wedge g {=} \mathsf{dec}(\phi_q)\} & \end{cases}$$

In the above formula, p_i (resp., p_o) denotes the height of the operand stack of m at the entry (resp., exit) of m_p. Also, q_i is the height of the stack at the entry of m_q, and TBC^p_m is the decorated bytecode for m_p. We use "_" to indicate that the value at the corresponding position is not relevant.

The function $\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$ is defined by cases. The first case is applied when m_p is a *sink* node with no out-edges. Otherwise, the second rule introduces an additional procedure m^c_p (c is for *continuation*), which is defined by as many rules as there are out-edges for m_p. These rules capture the different alternatives which execution can follow from m_p. We will unfold calls to m^c_p whenever it is deterministic (m_p has a single out-edge). This results in m_p calling m_q directly.

Example 2. The RBR of the CFG in Fig. 1 consists of the following rules where local variables have the same name as in the source code and o is the *this* object:

$$
\begin{array}{ll}
\mathit{fact}(\langle o,n\rangle,\langle r\rangle) & := \mathsf{init_local_vars}(\langle \mathit{ft},i\rangle),\ \mathit{fact}_1(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle). \\
\mathit{fact}_1(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle) & := \mathsf{iconst}(1,s_0),\ \mathsf{istore}(s_0,\mathit{ft}),\ \mathsf{iconst}(1,s_0), \\
& \quad \mathsf{istore}(s_0,i),\ \mathit{fact}_2(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle). \\
\mathit{fact}_2(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle) & := \mathsf{iload}(i,s_0),\ \mathsf{iload}(n,s_1), \\
& \quad \mathit{fact}_2^c(\langle o,n,\mathit{ft},i,s_0,s_1\rangle,\langle r\rangle). \\
\mathit{fact}_2^c(\langle o,n,\mathit{ft},i,s_0,s_1\rangle,\langle r\rangle) & := \mathsf{guard}(\mathsf{icmpgt}(s_0,s_1)),\ \mathit{fact}_4(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle). \\
\mathit{fact}_2^c(\langle o,n,\mathit{ft},i,s_0,s_1\rangle,\langle r\rangle) & := \mathsf{guard}(\mathsf{icmple}(s_0,s_1)),\ \mathit{fact}_3(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle). \\
\mathit{fact}_3(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle) & := \mathsf{iload}(\mathit{ft},s_0),\ \mathsf{iload}(i,s_1),\ \mathsf{imul}(s_0,s_1,s_0), \\
& \quad \mathsf{istore}(s_0,\mathit{ft}),\ \mathsf{iinc}(i,1),\ \mathit{fact}_2(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle). \\
\mathit{fact}_4(\langle o,n,\mathit{ft},i\rangle,\langle r\rangle) & := \mathsf{iload}(\mathit{ft},s_0),\ \mathsf{ireturn}(s_0,r).
\end{array}
$$

The first rule corresponds to the entry. Block $fact_4$ is a sink block. Blocks $fact_1$ and $fact_3$ have a single out-edge and we have unfolded the continuation. Finally, block $fact_2$ has two out-edges and needs the procedure $fact_2^c$. The RBR from the CFG of *doSum* in Fig. 2 is ($doSum_3$ merges several blocks with one out-edge):

$$
\begin{array}{ll}
\mathit{doSum}(\langle o,x\rangle,\langle r\rangle) & := \mathsf{init_local_vars}(\langle\rangle),\ \mathit{doSum}_1(\langle o,x\rangle,\langle r\rangle). \\
\mathit{doSum}_1(\langle o,x\rangle,\langle r\rangle) & := \mathsf{aload}(x,s_0), \mathit{doSum}_1^c(\langle o,x,s_0\rangle,\langle r\rangle). \\
\mathit{doSum}_1^c(\langle o,x,s_0\rangle,\langle r\rangle) & := \mathsf{guard}(\mathsf{nonnull}(s_0)), \mathit{doSum}_3(\langle o,x\rangle,\langle r\rangle). \\
\mathit{doSum}_1^c(\langle o,x,s_0\rangle,\langle r\rangle) & := \mathsf{guard}(\mathsf{null}(s_0)), \mathit{doSum}_2(\langle o,x\rangle,\langle r\rangle). \\
\mathit{doSum}_2(\langle o,x\rangle,\langle r\rangle) & := \mathsf{iconst}(0,s_0), \mathsf{ireturn}(s_0,r). \\
\mathit{doSum}_3(\langle o,x\rangle,\langle r\rangle) & := \mathsf{aload}(o,s_0), \mathsf{aload}(x,s_1), \mathsf{getfield}(List.data,s_1,s_1), \\
& \quad \mathit{fact}(\langle s_0,s_1\rangle,\langle s_0\rangle), \mathsf{aload}(o,s_1), \mathsf{aload}(x,s_2), \\
& \quad \mathsf{getfield}(List.next,s_2,s_2), \mathit{doSum}(\langle s_1,s_2\rangle,\langle s_1\rangle), \\
& \quad \mathsf{iadd}(s_0,s_1,s_0), \mathsf{ireturn}(s_0,r).
\end{array}
$$

We can see that a call to a different method, *fact*, occurs in $doSum_3$. This shows that our RBR allows simultaneously handling the two CFGs in our example. □

Rule-based Programs vs JBC Programs. Given a JBC program P, P_r denotes the RBP obtained from P. Note that, it is trivial to define an *interpreter* (or *abstract machine*) which can execute any P_r and obtain the same return value and termination behaviour as a JVM does for P. RBPs, in spite of their declarative appearance, are in fact imperative programs. As in the JVM, an interpreter for RBPs needs, in addition to a stack for activation records, a global heap. These activation records differ from those in the JVM in that the operand stack is no longer needed (as stack elements are explicit) and in that the scope of variables is no longer associated to methods but rather to rules. In RBPs all rules are treated uniformly, regardless of the method they originate from, so that method borders are somewhat blurred. As in the JVM, call-by-value is used for passing arguments in calls.

3 Proving Termination

This section describes how to prove termination of a JBC program given its RBR. The approach consists of two steps. In the first, we *abstract* the RBR rules by replacing all program data by their corresponding *size*, and replacing calls corresponding to bytecode instructions by *size constraints* on the values their variables can take. This step results in a Constraint Logic Program (CLP) [16] over integers, where, for any bytecode trace t, there exists a CLP trace t' whose states are abstractions of t states. In particular, every infinite (non terminating) bytecode trace has a corresponding infinite CLP trace, so that termination of the CLP program implies termination of the bytecode program. Note that, unlike in bytecode traces which are always deterministic, the execution of a CLP program can be non-deterministic, due to the precision loss inherent to the abstraction.

In the second step, we apply techniques for proving termination of CLP programs [9], which consist of: (1) analyzing the rules for each method to infer input-output *size relations* between the method input and output variables; (2) using the input-output size relations for the methods in the program, we infer a set of abstract *direct calls-to pairs* which describe, in terms of size-change, all possible calls from one procedure to another; and (3) given this set of abstract direct calls-to pairs, we compute a set of all possible calls-to pairs (direct and indirect), describing all transitions from one procedure to another. Then we focus on the pairs which describe loops, and try to identify *ranking functions* which guarantee the termination of each loop and thus of the original program.

3.1 Abstracting the Rules

As mentioned above, rule abstraction replaces data by the *size* of data, and focuses on relations between data size. For integers, their size is just their integer value [12]. For references, we take their size to be their *path-length* [24], i.e., the length of the maximal path reachable from the reference. Then, bytecode instructions are replaced by constraints on sizes taking into account a Static Single Assignment (SSA) transformation. SSA is needed because variables in CLP programs cannot be assigned more than one value. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be abstracted to $s'_0{=}s_1{+}s_0$ where s'_0 refers to the value of s_0 after executing the instruction. Also, the bytecode $\mathsf{getfield}(f, s_0, s_0)$ is abstracted to $s_0{>}s'_0$ if it can be determined that s_0 (before executing the instruction) does not reference a cyclic data-structure, since the length of the longest-path reachable from s_0 is larger than the length of the longest path reachable from s'_0.

bytecode b	*abstract bytecode* b^α	ρ_{i+1}
$\mathsf{iload}(l_v, s_j)$	$s'_j{=}\rho_i(l_v)$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$s'_j{=}\rho_i(s_j){+}\rho_i(s_{j+1})$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{guard}(\mathit{icmpgt}(s_j, s_{j+1}))$	$\rho_i(s_j){>}\rho_i(s_{j+1})$	ρ_i
$\mathsf{getfield}(f, s_j, s_j)$	if f is of ref. type: $\rho_i(s_j){>}s'_j$ if s_j is not cyclic otherwise $\rho_i(s_j) \geq s'_j$. If f is not of ref. type: *true*	$\rho_i[s_j \mapsto s'_j]$
$\mathsf{putfield}(f, s_j, s_{j+1})$ s.t f is of ref. type	if s_j and s_{j+1} do not share, $s'_k{\leq}\rho_i(s_k){+}\rho_i(s_{j+1})$ for any s_k that shares with s_j, otherwise *true*.	$\rho_i[s_k \mapsto s'_k]$

To implement the SSA transformation we maintain a mapping ρ of variable names (as they appear in the rule) to new variable names (constraint variables). Such a mapping is referred to as a *renaming*. We let $\rho[x \mapsto y]$ denote the modification of the renaming ρ such that it maps x to the new variable y. We denote by $\rho[\bar{x} \mapsto \bar{y}]$ the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

Definition 5 (abstract compilation). *Let* $\mathrm{R} \equiv p(\overline{x}, \overline{y}) := b_1, \ldots, b_n$ *be a rule. Let* ρ_i *be a renaming associated with the point before each* b_i *and let* ρ_1 *be the identity renaming (on the variables in the rule). The abstraction of* R *is denoted* R^α *and takes the form* $p(\overline{x}, \overline{y}') := b_1^\alpha, \ldots, b_n^\alpha$ *where* b_i^α *are computed iteratively from left to right as follows:*

1. *if* b_i *is a bytecode instruction or a guard, then* b_i^α *and* ρ_{i+1} *are obtained from a predefined lookup table similar to the one above.*
2. *if* b_i *is a call to a procedure* $q(\overline{w}, \overline{z})$*, then the abstraction* b_i^α *is* $q(\overline{w}', \overline{z}')$ *where each* $w'_k \in \overline{w}'$ *is* $\rho_i(w_k)$*, variables* $\overline{z}'$ *are fresh, and* $\rho_{i+1} = \rho_i[\overline{z} \mapsto \overline{z}', \overline{u} \mapsto \overline{u}']$ *where* $\overline{u}'$ *are also fresh variables and* $\overline{u}$ *is the set of all variables in* $\overline{w}$ *which reference data-structures that can be modified when executing* q *and those that share (i.e., might have common regions in the heap) with them.*
3. *at the end we define each* $y'_i \in \overline{y}'$ *to be the constrained variable* $\rho_{n+1}(y_i)$*.*

In addition, all reference *variables are (implicitly) assumed to be non-negative.*

Note that in point 2 above, the set of variables such that the data-structures they point to may be modified during the execution of q can be approximated by applying constancy analysis [14], which aims at detecting the method arguments that remain constant during execution, and sharing analysis [23] which aims at detecting reference variables that might have common regions on the heap. Also, the non-cyclicity condition required for the abstraction of getfield can be verified by non-cyclicity analysis [22]. In what follows, for simplicity, we assume that abstract rules are normalized to the form $p(\overline{x}, \overline{y}) := \varphi, p_1(\overline{x}_1, \overline{y}_1), \ldots, p_j(\overline{x}_j, \overline{y}_j)$ where φ is the conjunction of the (linear) size constraints introduced in the abstraction and each $p_i(\overline{x}_i, \overline{y}_i)$ is a call to a procedure (i.e., block or method).

Example 3. Recall the following rule from Ex. 2 (on the left) and its abstraction (on the right) where the renamings are indicated as comments.

Rule	Abstraction	Renaming
$fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) :=$	$fact_3(\langle o, n, ft, i\rangle, \langle r'\rangle) :=$	% $\rho_1 = id$
iload(ft, s_0),	$s'_0 = ft$,	% $\rho_2 = \rho_1[s_0 \mapsto s'_0]$
iload(i, s_1),	$s'_1 = i$,	% $\rho_3 = \rho_2[s_1 \mapsto s'_1]$
imul(s_0, s_1, s_0),	*true*,	% $\rho_4 = \rho_3[s_0 \mapsto s''_0]$
istore(s_0, ft),	$ft' = s''_0$,	% $\rho_5 = \rho_4[ft \mapsto ft']$
iinc$(i, 1)$,	$i' = i + 1$,	% $\rho_6 = \rho_5[i \mapsto i']$
$fact_2(\langle o, n, ft, i\rangle, \langle r\rangle)$.	$fact_2(\langle o, n, ft', i'\rangle, \langle r'\rangle)$.	% $\rho_7 = \rho_6[r \mapsto r']$

Note that imul is abstracted to *true*, since it imposes a non-linear constraint. □

3.2 Input Output Size-relations

We consider the abstract rules obtained in the previous step to infer an abstraction (w.r.t. size) of the input-output relation of the program blocks. Concretely, we infer *input-output size relations* of the form $p(\overline{x}, \overline{y}) \leftarrow \varphi$, where φ is a constraint describing the relation between the sizes of the input $\overline{x}$ and the output $\overline{y}$ upon exit from p. This information is needed since output of one call may be input to another call. E.g., consider the following contrived abstract rule $p(\langle x \rangle, \langle r \rangle)$:=$\{x{>}0, x{>}z\}, q(\langle z \rangle, \langle y \rangle), p(\langle y \rangle, \langle r \rangle)$. To prove termination, it is crucial to know the relation between x in the head and y in the recursive call to p. This requires knowledge about the input-output size relations for $q(\langle z \rangle, \langle y \rangle)$. Assuming this to be $q(\langle z \rangle, \langle y \rangle) \leftarrow z{>}y$, we can infer $x{>}y$. Since abstract programs are CLP programs, inferring relations can rely on standard techniques [4].

Computing an approximation of input-output size relation requires a global fixpoint. In practice, we can often take a trivial over-approximation where for all rules there is no information, namely, $p(\overline{x}, \overline{y}) \leftarrow \mathit{true}$. This can prove termination of many programs, and results in a more efficient implementation. It is not enough in cases as the above abstract rule, which however, in our experience, often does not occur in imperative programs.

3.3 Call-to Pairs

Consider again the abstract rule from Ex. 3 which (ignoring the output variable) is of the form $\mathit{fact}_3(\bar{x}) := \varphi, \mathit{fact}_2(\bar{z})$. It means that whenever execution reaches a call to $\mathit{fact}_3(\bar{x})$ there will be a subsequent call to $\mathit{fact}_2(\bar{z})$ and the constraint φ holds. In general, subsequent calls may arise also from rules which are not binary. Given an abstract rule of the form $p_0 := \varphi, p_1, \ldots, p_n$, a call to p_0 may lead to a call to p_i, $1{\le}i{\le}n$. Given the input-output size relations for the individual calls $p_1, \ldots, p_{i-1}$, we can characterize the constraint for a transition between the subsequent calls p_0 and p_i by adding these relations to φ. We denote a pair of such subsequent calls by $\langle p_0(\overline{x}) \rightsquigarrow p_i(\overline{y}), \varphi_i \rangle$ and call it a *calls-to pair*.

Definition 6 (direct calls-to pairs). *Given a set of abstract rules $\mathcal{A}$ and its input-output size relations $I_{\mathcal{A}}$, the* direct calls-to pairs *induced by $\mathcal{A}$ and $I_{\mathcal{A}}$ are:*

$$C_{\mathcal{A}} = \left\{ \langle p(\overline{x}) \rightsquigarrow p_i(\overline{x}_i), \psi \rangle \;\middle|\; \begin{array}{l} p(\overline{x}, \overline{y}) := \varphi, p_1(\overline{x}_1, \overline{y}_1), \ldots, p_j(\overline{x}_j, \overline{y}_j) {\in} \mathcal{A}, \\ i{\in}\{1, \ldots, j\},\ \forall 0{<}k{<}i.\ p_k(\overline{x}_k, \overline{y}_k) \leftarrow \varphi_k {\in} I_{\mathcal{A}} \\ \psi = \bar{\exists} \overline{x} \cup \overline{x}_i . \varphi \wedge \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \end{array} \right\}$$

where $\bar{\exists} v$ means eliminating all variables but v from the corresponding constraint.

Example 4. Consider the rule for *doSum* in Ex. 2: note that input-output relations for *fact* and *doSum* are *true*. Direct calls-to pairs for those rules are:

$$\begin{array}{l}
\langle doSum(o,x) \rightsquigarrow doSum_1(o',x'), \{x'{=}x, o'{=}o\}\rangle \\
\langle doSum_1(o,x) \rightsquigarrow doSum_1^c(o',x',s_0), \{x'{=}x, o'{=}o, s_0{=}x\}\rangle \\
\langle doSum_1^c(o,x,s_0) \rightsquigarrow doSum_3(o',x'), \{x'{=}x, o'{=}o, s_0{>}0\}\rangle \\
\langle doSum_1^c(o,x.s_0) \rightsquigarrow doSum_2(o',x'), \{x'{=}x, o'{=}o, s_0{=}0\}\rangle \\
\langle doSum_3(o,x) \rightsquigarrow fact(s_0',s_1''), \{s_0'{=}o\}\rangle \\
\langle doSum_3(o,x) \rightsquigarrow doSum(s_1''',s_2''), \{s_1'''{=}o, x{>}s_2''\}\rangle
\end{array}$$

In the last rule, s_2'' corresponds to $x.next$, so that we have the constraint $x{>}s_2''$. It can be seen that since the list is not cyclic and does not share with other variables, size analysis finds the above decreasing of its size $x{>}s_2''$. Note also that all variables corresponding to references are assumed to be non-negative. Similarly, we can obtain direct calls-to pairs for the rule of *fact*. □

It should be clear that the set of direct calls-to pairs relations $C_{\mathcal{A}}$ is also a binary CLP program that we can execute from a given goal. A key feature of this binary program is that if an infinite trace can be generated using the abstract program described in Sec. 3.1, then an infinite trace can be generated using this binary CLP program [10]. Therefore, absence of such infinite traces (i.e., termination) in the binary program $C_{\mathcal{A}}$ implies absence of infinite traces in the abstract bytecode program, as well as in the original bytecode program.

Theorem 1 (Soundness). *Let P be a JBC program and $C_{\mathcal{A}}$ the set of direct calls-to pairs computed from P. If there exists a non-terminating trace in P then there exists a non-terminating derivation in $C_{\mathcal{A}}$.*

Intuitively, the result follows from the following points. By construction, the RBP captures all possibly non-terminating traces in the original program. By the correctness of size analysis, we have that, given a trace in the RBP, there exists an equivalent one in $C_{\mathcal{A}}$, among possibly others. Therefore, termination in $C_{\mathcal{A}}$ entails termination in the JBC program.

3.4 Proving Termination of the Binary program $C_{\mathcal{A}}$

Several automatic termination tools and methods for proving termination of such binary constraint programs exists [9, 10, 17]. They are based on the idea of first computing all possible calls-to pair from the direct ones, and then finding a ranking function for each recursive calls-to pairs, which is sufficient for proving termination. Computing all possible calls-to pairs, usually called the *transitive closure* $C_{\mathcal{A}}^*$, can be done by starting from the set of direct calls-to pairs $C_{\mathcal{A}}$, and iteratively adding to it all possible compositions of its elements until a fixed-point is reached. Composing two calls-to pairs $\langle p(\overline{x}) \rightsquigarrow q(\overline{y}), \varphi_1\rangle$ and $\langle q(\overline{w}) \rightsquigarrow r(\overline{z}), \varphi_2\rangle$ returns the new calls-to pair $\langle p(\overline{x}) \rightsquigarrow r(\overline{z}), \bar{\exists}\bar{x} \cup \bar{z}.\varphi_1 \wedge \varphi_2 \wedge (\bar{y} = \bar{w})\rangle$.

Example 5. Applying the transitive closure on the direct calls-to pairs of Ex. 4, we obtain, among many others, the following calls-to pairs. Note that x (resp. i) strictly decreases (resp. increases) at each iteration of its corresponding loop:

$$\langle doSum(o,x) \rightsquigarrow doSum(o',x'), \{o'{=}o, x{>}x', x{\geq}0\}\rangle$$
$$\langle fact_2(o,n,ft,i) \rightsquigarrow fact_2(o',n',ft',i'), \{o'{=}o, n'{=}n, i'{>}i, i{\geq}1, n{\geq}i'{-}1\}\rangle$$

□

As already mentioned, in order to prove termination, we focus on *loops* in $C^*_{\mathcal{A}}$. Loops are the *recursive* entities of the form $\langle p(\overline{x}) \rightsquigarrow p(\overline{y}), \varphi\rangle$ which indicate that a call to a program point p with values $\bar{x}$ eventually leads to a call to the same program point with values $\bar{y}$ and that φ holds between $\overline{x}$ and $\overline{y}$. For each loop, we seek a *ranking function* F over a well-founded domain such that $\varphi \models \text{F}(\overline{x}){>}\text{F}(\overline{y})$. As shown in [9, 10], finding a ranking function for every recursive calls-to pair implies termination. Computing such functions can be done, for instance, as described in [21]. As an example, for the loops in Ex. 5 we get the following ranking functions: $\text{F}_1(o,x){=}x$ and $\text{F}_2(o,n,ft,i){=}n{-}i{+}1$.

3.5 Improving Termination Analysis by Extracting Nested Loops

In proving termination of JBC programs, one important question is whether we can prove termination at the JBC level for a class of programs which is comparable to the class of Java *source* programs for which termination can be proved using similar technology. As can be seen in Sec. 4, directly obtaining the RBR of a bytecode program is non-optimal, in the sense that proving termination on it may be more complicated than on the source program. This happens because, while in source code it is easy to reason about a nested loop independently of the outer loop, loops are not directly visible when control flow is unstructured. Loop extraction is useful for our purposes since nested loops can be dealt with one at a time. As a result, finding a ranking function is easier, and computing the closure can be done locally in the strongly connected components. This can be crucial in proving the termination of programs with nested loops.

To improve the accuracy of our analysis, we include a component which can detect and extract loops from CFGs. Due to space limitations, we do not describe how to perform this step here (more details in work about decompilation [2], where loop extraction has received considerable attention). Very briefly, when a loop is extracted, a new CFG is created. As a result, a method can be converted into several CFGs. These ideas fit very nicely within our RBR, since calls to loops are handled much in the same way as calls to other methods.

4 Experimental results

Our prototype implementation is based on the size analysis component of [1] and extends it with the additional components needed to prove termination. The

Benchmark	CFG	RBR	Size	TC	RF	Total_1	Termin	Total_2	Ratio
Polynomial	138	12	260	1453	26	1890	yes	2111	1.12
DivByTwo	52	4	168	234	4	462	yes	538	1.17
EvenDigits	59	7	383	1565	17	2030	yes	2210	1.09
Factorial	43	3	46	268	3	363	yes	353	0.97
ArrayReverse	58	5	208	339	24	635	yes	834	1.32
Concat	65	8	660	943	38	1715	yes	3815	2.23
Incr	35	12	854	4723	28	5652	yes	6590	1.17
ListReverse	21	5	141	310	5	481	yes	515	1.07
MergeList	107	23	130	5184	21	5464	yes	5505	1.01
Power	14	3	72	357	9	454	yes	459	1.01
Cons	25	7	65	1318	10	1424	yes	1494	1.05
ListInter	136	22	585	9769	49	10560	yes	27968	2.65
SelectOrd	154	16	1298	4076	48	5592	no	25721	4.60
DoSum	57	10	64	923	6	1060	yes	1069	1.01
Delete	121	14	54	2418	1	2608	yes	33662	12.91
MatMult	240	11	2411	4646	294	7602	no	32212	4.24
MatMultVector	254	15	2563	8744	242	11817	no	34688	2.94
Hanoi	39	5	172	979	3	1198	no	1198	1.00
Fibonacci	23	3	90	290	5	411	yes	401	0.98
BST	68	12	97	4643	18	4838	yes	4901	1.01
BubbleSort	152	12	1125	4366	83	5738	no	14526	2.53
Search	65	11	307	756	11	1150	yes	1430	1.24
Sum	64	7	480	1758	35	2343	no	5610	2.39
FactSumList	65	12	80	961	5	1123	yes	1306	1.16
Scoreboard	268	23	1597	4393	81	6362	no	32999	5.19

Table 1. Measured time (in ms) of the different phases of proving termination

analyzer can also output the set of direct call-pairs, which allows using existing termination analyzers based on similar ideas [10, 17]. The system is implemented in Ciao Prolog, and uses the Parma Polyhedra Library (PPL) [3].

Table 1 shows the execution times of the different steps involved in proving the termination of JBC programs, computed as the arithmetic mean of five runs. Experiments have been performed on an Intel 1.86 GHz Pentium M with 1 GB of RAM, running Linux on a 2.6.17 kernel. The table shows a range of benchmarks for which our system can prove termination, and which are meant to illustrate different features. We show classical recursive programs such as *Hanoi*, *Fibonacci*, *MergeList* and *Power*. Iterative programs *DivByTwo* and *Concat* contain a single loop, while *Sum*, *MatMult* and *BubbleSort* are implemented with nested loops. We also include programs written in object-oriented style, like *Polynomial*, *Incr*, *Scoreboard*, and *Delete*. The remaining benchmarks use data structures: arrays (*ArrayReverse*, *MatMultVector*, and *Search*); linked lists (*Delete* and *ListReverse*); and binary trees (*BST*).

Columns **CFG**, **RBR**, **Size**, **TC**, **RF**, $\textbf{Total}_1$ contain the running times (in ms) required for the CFG (including loop extraction), the RBR, the size analysis

(including input-output relations), the transitive closure, the ranking functions and the total time, respectively. Times are high, as the implementation has been developed to check if our approach is feasible, but is still preliminary. The most expensive steps are the size analysis and the transitive closure, since they require global analysis. Last three columns show the benefits of loop extraction. **Termin** tells if termination can be proven (using polyhedra) without extraction. In seven cases, termination is only proven if loop extraction is performed. $\textbf{Total}_2$ shows the total time required to check termination without loop extraction. **Ratio** compares $\textbf{Total}_2$ with $\textbf{Total}_1$ ($\textbf{Total}_2/\textbf{Total}_1$), showing that, in addition to improving precision, loop extraction is beneficial for efficiency, since $\textbf{Ratio} \geq 1$ in most cases, and can be as high as 12.91 in *Delete*. Note that termination of these programs may be proved without loop extraction by using other domains such as *monotonicity constraints* [7]. However, we argue that loop extraction is beneficial as it facilitates reasoning on the loops separately. Also, if it fails to prove termination, it reports the possibly non-terminating loops.

5 Conclusions and Related Work

We have presented a termination analysis for (sequential) JBC which is, to the best of our knowledge, the first approach in this direction. This analysis successfully deals with the challenges related to the low-level nature of JBC, and adapts standard techniques used, in other settings, in decompilation and termination analysis. Also, we believe that many of the ideas presented in this paper are also applicable to termination analysis of low-level languages in general, and not only JBC. We have used the notion of path-length to measure the size of data structures on the heap. However, our approach is parametric on the abstract domain used to measure the size. As future work, we plan to implement non-cyclicity analysis [22], constancy analysis [14], and sharing analysis [23], and to enrich the transitive closure components with monotonicity constraints [7]. Unlike polyhedra, monotonicity constraints can handle disjunctive information which is often crucial for proving termination. In [5], a termination analysis for C programs, based on binary relations similar to ours, is proposed. It uses separation logic to approximate the heap structure, which in turn allows handling termination of programs manipulating cyclic data structures. We believe that, for programs whose termination does not depend on cyclic data-structures, both approaches deal with the same class of programs. However, ours might be more efficient, as it is based on a simpler abstract domains (a detailed comparison is planned for future work). Recently, a novel termination approach has been suggested [8]. It is based on cyclic proofs and separation logic, and can even handle complicated examples as the reversal of panhandle data-structures. It is not clear to us how practical this approach is.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of

Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. Samir Genaim was supported by a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. Part of this work was performed during a research stay of Michael Codish at UPM supported by a grant from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java Bytecode. In *Proc. of ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
2. F. Allen. Control flow analysis. In *Symp. on Compiler optimization*, 1970.
3. R. Bagnara, E. Ricci, E. Zaffanella, and P. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *SAS*, LNCS, 2002.
4. F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *LOPSTR'06*, volume 1207 of *LNCS*, pages 204–223. Springer-Verlag, 1997.
5. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV*, 2006.
6. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
7. A. Brodsky and Y. Sagiv. Inference of Inequality Constraints in Logic Programs. In *Proceedings of PODS'91*, pages 95–112. ACM, 1991.
8. J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, Jan 2008.
9. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM TOPLAS*, 29(2), 2007.
10. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.
11. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*. ACM, 1978.
13. R. DeLine and R. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft, 2005.
14. S. Genaim and F. Spoto. Technical report, 2007. Personal Communication.
15. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
16. J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
17. C. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proc. POPL*. ACM, 2001.
18. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. A-W, 1996.
20. G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

21. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
22. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI*, volume 3855 of *LNCS*. S-V, 2006.
23. S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In *SAS*, pages 320–335, 2005.
24. F. Spoto, P. M. Hill, and E. Payet. Path-length analysis for object-oriented programs. In *Proc. EAAI*, 2006.
25. F. Spoto and T. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.
26. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of CASCON 1999*, pages 125–135, 1999.