Termination Analysis of Java Bytecode*

E. Albert[1], P. Arenas[1], M. Codish[2],
S. Genaim[3], G. Puebla[3], and D. Zanardini[3]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
[3] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

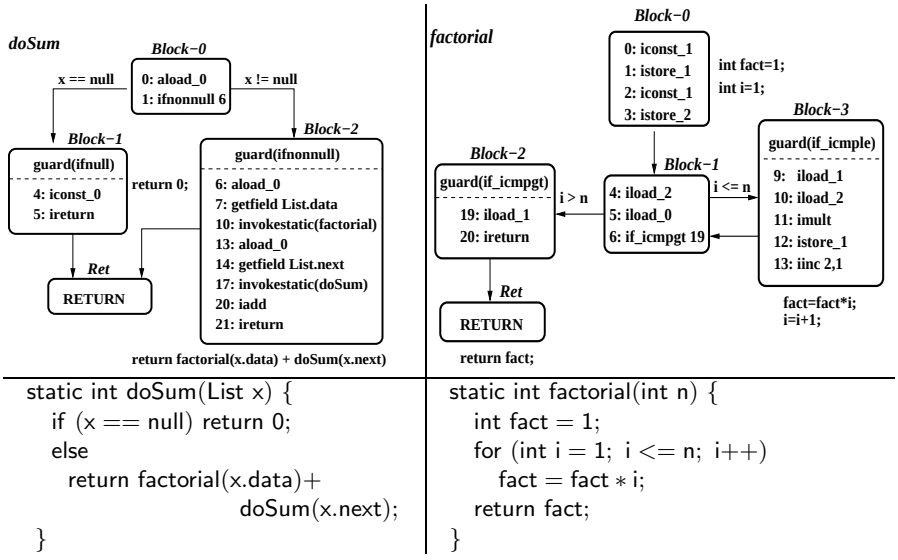
Introduction: The state of the art in termination analysis includes advanced techniques developed for logic and functional programming [12, 4, 9, 11, 10] and imperative languages [2, 5, 8, 6, 10], as well as for term rewriting systems [10]. In [6, 5] tools for proving termination of large industrial code are presented. However, termination of low-level languages, such as Java bytecode, has received little attention. In some situations, such as with *mobile code*, the user only has access to compiled code which may be obtained from an untrusted party. Termination analysis may help avoid, for example, *denial of service* attacks. Such analysis must be applied directly on the low-level code. Java bytecode [13] is widely used in this context due to its security features and platform-independence.

Termination analyses are typically based on *ranking functions*, which map program states to the elements of a well-founded domain. Termination is guaranteed if a ranking function which decreases during computation is found, in particular as the program goes through its loops. Termination analysis of Java bytecode presents some peculiar features which stem from its low-level and object-oriented nature: (1) loops originate from different sources (conditional and unconditional jumps, method calls, or even exceptions); (2) *size measures* must consider supported data types (primitive types, objects, and arrays); and (3) data can be stored in local variables, operand stack elements or heap locations.

In this work, we describe a termination analysis for Java bytecode, based on a translation into a structured *intermediate representation* similar to that of [1] where it is applied in the context of cost analysis. This representation captures in a uniform setting all loops (regardless of whether they originate from recursive calls or iterative loops) and all variables (either local variables or operand stack elements). Given this representation we consider a general form of the *size-change graphs* principle [11, 4] to prove termination of the intermediate program, which in turn implies the termination of the original program.

Termination of Java Bytecode by Example: We illustrate our approach by means of a running example which consists of two Java methods, doSum and

* 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. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.




```
static int doSum(List x) {
  if (x == null) return 0;
  else
    return factorial(x.data)+
                    doSum(x.next);
}
```

```
static int factorial(int n) {
  int fact = 1;
  for (int i = 1; i <= n; i++)
    fact = fact * i;
  return fact;
}
```

Fig. 1. Control flow graphs for the running example, with possible Java source code

factorial. Fig. 1 depicts their *control flow graphs* (CFGs for short), explained in more detail below, and provides a possible Java source for each method. The process is divided into five steps. The first two are required to obtain a structured representation of the program. The other three are required for proving termination.

Step I: Control Flow Graph. To facilitate standard termination proving techniques, we first transform the unstructured control flow of bytecode into a structured intermediate representation. A control flow graph consists of *guarded* basic blocks and edges which describe how control flows between blocks. Basic blocks are sequences of non-branching bytecode instructions, and edges are obtained from instructions which might branch such as virtual method invocation, conditional jumps, exceptions, etc. Observe in the CFG for doSum in Fig. 1 (left), the branching at `Block-0` which distinguishes the base case (`Block-1`) and the recursive step (`Block-2`) for the recursive definition (in the Java code). The successive blocks at a branch have mutually exclusive *guards* since only one of them can be executed. Guards take the form `guard(cond)`, where `cond` is a Boolean condition on the local variables and stack elements. For example, `guard(ifnull)` succeeds if the top stack element s_0 is a null reference. The branching at `Block-1` in the CFG of factorial corresponds to the condition of the for loop in which the guard `guard(if_icmpgt)` on block `Block-2` succeeds if the top two stack elements s_0 and s_1 satisfy $s_0 > s_1$. The special block `Ret` indicates a normal exit of the method. Note also that (static) method invocations –both the recursive call to doSum and the non-recursive call to factorial– appear within `invokestatic` bytecode instructions.

Step II: Intermediate Representation. In the next step, the CFG is represented in a procedural way by means of an *intermediate representation.* This representation consists of a set of *guarded rules* which are obtained from the blocks in the CFG. A principal advantage is that all possible forms of loops in the program are represented now in a uniform way. We can observe this, for instance, in the representation of the for loop by means of rule $factorial_1$ and the representation of the recursive procedure doSum in the rule *doSum*. Although method factorial is not recursive, its associated representation is, as $factorial_1$ contains a loop.

$$
\begin{array}{ll}
factorial(n, ret) & \leftarrow factorial_0(n, fact, i, ret). \\
factorial_0(n, fact, i, ret) & \leftarrow \texttt{iconst}(1, s_0), \texttt{istore}(s_0, fact), \texttt{iconst}(1, s_0), \\
 & \quad \texttt{istore}(s_0, i), factorial_1(n, fact, i, ret). \\
 & \\
factorial_1(n, fact, i, ret) & \leftarrow \texttt{iload}(i, s_0), \texttt{iload}(n, s_1), \\
 & \quad (factorial_2(n, fact, i, s_0, s_1, ret); \\
 & \quad factorial_3(n, fact, i, s_0, s_1, ret)). \\
 & \\
factorial_2(n, fact, i, s_0, s_1, ret) & \leftarrow \texttt{guard}(\texttt{if_icmpgt}(s_0, s_1)), \texttt{iload}(fact, s_0), \\
 & \quad \texttt{ireturn}(s_0, ret). \\
 & \\
factorial_3(n, fact, i, s_0, s_1, ret) & \leftarrow \texttt{guard}(\texttt{if_icmple}(s_0, s_1)), \\
 & \quad \texttt{iload}(fact, s_0), \texttt{iload}(i, s_1), \texttt{imul}(s_0, s_1, s_0), \\
 & \quad \texttt{istore}(s_0, fact), \texttt{iinc}(i, 1), factorial_1(n, fact, i, ret).
\end{array}
$$

$$
\begin{array}{ll}
doSum(x, ret) & \leftarrow doSum_0(x, ret). \\
doSum_0(x, ret) & \leftarrow \texttt{aload}(x, s_0), (doSum_1(x, s_0, ret); doSum_2(x, s_0, ret)). \\
doSum_1(x, s_0, ret) & \leftarrow \texttt{guard}(\texttt{ifnull}(s_0)), \texttt{iconst}(0, s_0), \texttt{ireturn}(s_0, ret). \\
doSum_2(x, s_0, ret) & \leftarrow \texttt{guard}(\texttt{ifnonnull}(s_0)), \texttt{aload}(x, s_0), \texttt{getfield}(List.data, s_0, s_0), \\
 & \quad factorial(s_0, s_0), \texttt{aload}(x, s_1), \texttt{getfield}(List.next, s_1, s_1), \\
 & \quad doSum(s_1, s_1), \texttt{iadd}(s_1, s_0, s_0), \texttt{ireturn}(s_0, ret).
\end{array}
$$

A relevant feature of our representation is that the arguments of each rule correspond to: (1) the method's local variables ($n, fact, i$ for factorial, and x for doSum in our example); (2) a single variable which corresponds to the method's return value (ret); and (3) the *active stack* elements s_i at the block's entry and exit, i.e., the stack elements are considered as local variables. The guards and the bytecodes which appear in the rules are written in a different font in the above example to distinguish them from calls to blocks. They are obtained from the block's guard and bytecode instructions, by adding the local variables and stack elements on which they operate as explicit arguments.

Step III: Inferring Size Relations and Binary Clauses. Next, a global size analysis is performed on the recursive representation to infer *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) occurring in the body. In our example, assuming that the input to doSum is an acyclic list, we obtain the following *calls-to* size-relations:

$$
\begin{array}{ll}
\langle factorial(n) & \mapsto factorial_0(n', fact, i), \{n'{=}n\}\rangle \\
\langle factorial_0(n, fact, i) & \mapsto factorial_1(n', fact', i'), \{n'{=}n, fact'{=}1, i'{=}1\}\rangle \\
\langle factorial_1(n, fact, i) & \mapsto factorial_2(n', fact', i', s_0, s_1), \{s_0{=}i, s_1{=}n, s_0{\leq}s_1\} \cup \varphi_{id}\rangle \\
\langle factorial_1(n, fact, i) & \mapsto factorial_3(n', fact', i', s_0, s_1), \{s_0{=}i, s_1{=}n, s_0{>}s_1\} \cup \varphi_{id}\rangle \\
\langle factorial_3(n, fact, i, s_0, s_1) & \mapsto factorial_1(n', fact', i'), \{n'{=}n, i'{=}i{+}1\}\rangle \\
\langle doSum(x) & \mapsto doSum_0(x'), \{x'{=}x\}\rangle \\
\langle doSum_0(x) & \mapsto doSum_1(x', s_0), \{x'{=}x, s_0{=}x, s_0{=}0\}\rangle \\
\langle doSum_0(x) & \mapsto doSum_2(x', s_0), \{x'{=}x, s_0{=}x, s_0{>}0\}\rangle \\
\langle doSum_2(x, s_0) & \mapsto factorial(_, _), \{\}\rangle \\
\langle doSum_2(x, s_0) & \mapsto doSum(x'), \{x'{=}x{-}1\}\rangle
\end{array}
$$

where $\varphi_{id} \equiv \{n'{=}n, fact'{=}fact, i'{=}i\}$. Size relations provide information about the changes in the size of the data structure (or in the value of integer variables) when control goes from one part of the program (e.g., a block or a method) to another one. For example, rule $factorial_3$ shows that the value of the loop counter i increases by 1. The analysis is done by: (1) abstracting bytecode instructions into the linear constraints they impose on their arguments size, where the size of an integer variable is its value [7], the size of a reference variable is the length of the maximal-path [14] reachable from that variable and the size of an array is its length; for instance, the abstraction of the bytecode instruction `iinc(i,1)` results in `i=i+1`; and (2) computing a fixpoint which tracks the *calls-to* relations. Note that the path-length analysis [14] abstracts cyclic data structures to top, therefore our analysis cannot infer termination when it depends on cyclic data structures.

Step IV: Transitive Closure of Binary clauses. The *calls-to* relations are exactly the *binary clauses* that represents the *direct calls* between the different blocks of the program. Starting from this set, we iteratively compute the transitive closure of their composition in order to obtain *binary clauses* for the indirect calls, in particular those that correspond to loops. Informally, composing two binary clauses $\langle p(\bar{x}) \mapsto q(\bar{y}), \varphi_1\rangle$ and $\langle q(\bar{y}) \mapsto h(\bar{z}), \varphi_2\rangle$ results in a new binary clause $\langle p(\bar{x}) \mapsto h(\bar{z}), \exists \bar{y}.\varphi_1 \wedge \varphi_2\rangle$. The next recursive binary clauses are, among unnecessary others, obtained by the transitive closure of the above *binary clauses*:

$$
\begin{array}{ll}
\langle factorial_1(n, fact, i) & \mapsto factorial_1(n', fact', i'), \{n'{=}n, i'{>}i, i{\leq}n\}\rangle \\
\langle doSum(x) & \mapsto doSum(x'), \{x'{<}x, x{>}0\}\rangle
\end{array}
$$

The notion of *binary clauses* of [4] is a general form of *size-change graphs* [11] in the sense that it allows the use of arbitrary constraints domains to describe the corresponding program states. This is useful in practice when different loops might diverge in different directions and when the decreasing measure is a non-trivial combination of some of the loop arguments. Similar notions to that of *binary clauses* have been introduced also in [12, 9, 16, 3].

Step V: Ranking Functions for Loops. Recursive *binary clauses* in the transitive closure represent all possible loops in the program, and in order to prove that the corresponding program terminates, it is sufficient to check that for each such recursive *binary clause* $\langle p(\bar{x}) \mapsto p(\bar{y}), \varphi\rangle$ there exists a function f over a well-founded domain such that $\varphi \models f(p(\bar{x})) > f(p(\bar{y}))$ [3]. In the last step we check the existence of a ranking function [15] for each recursive *binary clause* in the transitive closure. The functions $f(factorial_1(n, fact, i)){=}n{-}i$ and

$f(doSum(x))=x$ are ranking functions which guarantee that the corresponding loop will be traversed finitely, therefore the program terminates.

Summary: We have a preliminary implementation which is based on the size analysis of [1] and the *binary clauses* component of the TerminWeb analyzer [17]. Ongoing work is to formally justify the analysis following the approach of [4], namely by deriving a concrete *binary clauses* semantics for Java bytecode from which termination is observable, and then define the *size analysis* as an abstract interpretation of this semantics. From the practical side we plan to improve the analyzer by adopting some of the techniques described in [6], which have been proved to be very efficient in practice.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP*, 2007.
2. A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
3. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *TOPLAS*, 2006. to appear.
4. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
5. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, 2002.
6. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.
8. P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, 2005.
9. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):117–156, 2001.
10. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
11. C. S. Lee, N. D. Jones, and A. M. Ben-Ammar. The size-change principle for program termination. In *POPL*, 2001.
12. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. A-W, 1996.
14. P. M. Hill, E. Payet, and F. Spoto. Path-length analysis of object-oriented programs. In *EAAI*, 2006.
15. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.
16. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.
17. C. Taboch, S. Genaim, and M. Codish. Terminweb: Semantic based termination analyser for logic programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.