Handling non-linear operations in the value analysis of COSTA

Diego Alonso [a] Puri Arenas[a] Samir Genaim[a]

[a] *DSIC, Complutense University of Madrid (UCM), Spain*

Abstract

Inferring precise relations between (the values of) program variables at different program points is essential for termination and resource usage analysis. In both cases, this information is used to synthesize ranking functions that imply the program's termination and bound the number of iterations of its loops. For efficiency, it is common to base *value analysis* on non-disjunctive abstract domains such as Polyhedra, Octagon, etc. While these domains are efficient and able to infer complex relations for a wide class of programs, they are often not sufficient for modeling the effect of non-linear and bit arithmetic operations. Modeling such operations precisely can be done by using more sophisticated abstract domains, at the price of performance overhead. In this paper we report on the value analysis of COSTA that is based on the idea of encoding the disjunctive nature of non-linear operations into the (abstract) program itself, instead of using more sophisticated abstract domains. Our experiments demonstrate that COSTA is able to prove termination and infer bounds on resource consumption for programs that could not be handled before.

1 Introduction

Termination and resource usage analysis of imperative languages have received a considerable attention [3,22,20,8,19,13,14]. Most of these analyses rely on a value (or size) analysis component, which infers relations between the values of the program variables (or the sizes of the corresponding data structures) at different program points. This information is then used to bound the number of iterations of the program's loops. Thus, the precision of value analysis directly affects the class of (terminating) programs for which the corresponding tool is able prove termination or infer lower and upper bounds on their resource consumption. Moreover, in the case of resource consumption, it also affects the quality of the inferred bounds (i.e., how tight there are).

Typically, for efficiency, the underlying abstract domains used in value analysis are based on conjunctions of linear constraints, e.g., Polyhedra [10],

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

Octagons [18], etc. While in practice these abstract domains are precise enough for bounding the loops of many programs, they are often not sufficient when the considered program involves non-linear arithmetic operations (multiplication, division, bit arithmetics, etc). This is because the semantics of such operations cannot be modeled precisely with only conjunctions of linear constraints. In order to overcome this limitation, one can use abstract domains that support non-linear constraints, however, these domain typically impose a significant performance overhead. Another alternative is to use disjunctive abstract domains, i.e., disjunctions of (conjunctions of) linear constraints. This allows splitting the behavior of the corresponding non-linear operation into several mutually exclusive cases, such that each one can be precisely described using only conjunctions of linear constraints. This alternative also imposes performance overhead, since the operations of such disjunctive abstract domains are usually more expensive.

In this paper, we develop a value analysis that handles non-linear arithmetic operations using disjunctions of (conjunctions of) linear constraints. However, similarly to [21], instead of directly using disjunctive abstract domains, we encode the disjunctive nature of the non-linear operations directly in the (abstract) program. This allows using non-disjunctive domains like Polyhedra, Octagons, etc., and still benefit from the disjunctive information in order to infer more precise relations for programs with non-linear arithmetic operations. We have implemented a prototype of our analysis in COSTA, a COSt and Termination Analyser for Java bytecode. Experiments on typical examples from the literature demonstrate that COSTA is able to handle programs with non-linear arithmetics that could not be handled before.

The rest of this paper is organized as follows: Section 2 briefly describes the intermediate language on which we develop our analysis (Java bytecode programs are automatically translated to this language); Section 3 motivates the techniques we use for handling non-linear arithmetic operations; Section 4 describes the different components of our value analysis; Section 5 presents a preliminary experimental evaluation using COSTA; and, finally, we conclude in Section 6.

2 A Simple Imperative Intermediate Language

We present our analysis on a simple *rule-based* imperative language [1] which is similar in nature to other representations of bytecode [23,16]. For simplicity, we consider a subset of the language presented in [1], which deals only with methods and arithmetic operations over integers. In the implementation we handle full sequential Java bytecode. A *rule-based program* P consists of a set of *procedures*. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*.

Rules adhere to this grammar:

$$\begin{aligned}
\mathit{rule} &::= p(\bar{x},\bar{y}) \ \leftarrow g, b_1, \ldots, b_n \\
g &::= \mathit{true} \mid e_1 \ \mathit{op} \ e_2 \mid g_1 \wedge g_2 \\
b &::= x{:=}e \mid x{:=}e - e \mid x{:=}e + e \mid q(\bar{x},\bar{y}) \\
&\quad\ x{:=}e * e \mid x{:=}e \ / \ e \mid x{:=}e \ \texttt{rem} \ e \\
&\quad\ x{:=}e \ \otimes \ e \mid x{:=}e \oplus e \mid x{:=}e \triangleright e \mid x{:=}e \triangleleft e \\
e &::= x \mid n \\
\mathit{op} &::= >|<|\leq|\geq|=
\end{aligned}$$

where $p(\bar{x},\bar{y})$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables and $q(\bar{x},\bar{y})$ a procedure call by value. The arithmetic operations / and `rem` refer respectively to integer division and remainder. They have the semantics of the bytecode instructions `idiv` and `irem` [17]. Operations $\otimes$, $\oplus$, $\triangleleft$ and $\triangleright$ refer respectively to bitwise AND, bitwise OR, left shift and right shift. They have the semantics of the bytecode instructions `iand`, `ior`, `ishl`, and `ishr` [17]. We ignore the overflow behavior of these instruction, supporting them is left for future work.

The key features of this language which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, and (4) rules may have *multiple output* parameters which is useful for our transformation. The translation from Java bytecode programs to rule-based programs is performed in two steps. First, a control flow graph (CFG) is built. Second, a *procedure* is defined for each basic block in the CFG and the operand stack is *flattened* by considering its elements as additional local variables. The execution of rule-based programs mimics standard bytecode [17]. Multiple output arguments in procedures come from the extraction of loops into separated procedure (see Example 2.1). For simplicity, we assume that each rule in the program is given in static single assignment (SSA) form [5].

Example 2.1 Figure 1 depicts the Java code (left) and the corresponding intermediate representation (right) of our running example. Note that our analysis starts from the bytecode, the Java code is shown here just for clarity. Procedure m is defined by one rule, it receives x and b as input, and returns r as output, i.e., r corresponds to the return value of the Java method. Rule m corresponds to the first two instructions of the Java method, it initializes local variables y and z, and then passes the control to m_1. Procedure m_1 corresponds to the **if** statement, and is defined by two mutually exclusive rules. The first one is applied when $b \leq 1$, and simply returns the value of z in the output variable r. The second one is applied when $b > 1$, it calls procedure m_2 (the loop), and upon exit from m_2 it returns the value of z_1 in

```
int m(int x,int b){
  int y=1;
  int z=0;
  if (b>1) {
    while (y<x){
      z=z+1;
      y=y*b;
    }
  }
  return z;
}
```

$m(\langle x, b\rangle, \langle r\rangle) \leftarrow$
  $y{:=}1,$
  $z{:=}0,$
  $m_1(\langle x, b, y, z\rangle, \langle r\rangle).$

$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b \leq 1,$
  $r{:=}z.$

$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
  $b > 1,$
  $m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle),$
  $r{:=}z_1.$

$m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow$
  $y \geq x.$

$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow$
  $y < x,$
  $z_1{:=}z + 1,$
① $y_1{:=}y * b,$
  $m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$

Fig. 1. A Java program and its intermediate representation. Method m computes $\lceil \log_b(x) \rceil$.

the output variable r. Note that z_1 refers to the value of z upon exit from procedure m_2 (the loop), it is generated by the SSA transformation. Procedure m_2 corresponds to the **while** loop, and is defined by two mutually exclusive rules. The first one is applied when the loop condition is evaluated to *false*, and the second one when it is evaluated to *true*. Note that m_2 has two output variables, they correspond to the values of y and z upon exit from the loop.

3 Motivating Example

Proving that the program of Figure 1 terminates, or inferring lower and upper bounds on its resource consumption (e.g., number of execution steps), requires bounding the number of iterations that its loop can make. Bounding the number of iterations of a loop is usually done by finding a function f from the program states to a well-founded domain, such that if s and s' are two states that correspond to two consecutive iterations, then $f(s) > f(s')$. Traditionally, this function is called *ranking function* [11]. Note that for termination, it is enough to prove that such function exists, while inferring bounds on the resource consumption requires synthesizing such ranking function. For the program of Figure 1, if the program state is represented by the tuple $\langle x, b, y, z\rangle$, then $f(\langle x, b, y, z\rangle) = \texttt{nat}(x - y)$, where $\texttt{nat}(v) = \max(v, 0)$, is a ranking function for the **while** loop. Moreover, this function can be further refined to $f(\langle x, b, y, z\rangle) = \log_2(\texttt{nat}(x - y) + 1)$, which is more accurate for the sake of inferring bounds on the loop's resource consumption.

In this paper we follow the analysis approach used in [1], which divides the value analysis into several steps: (1) an *abstract compilation* [15] step that generates an abstract version of the program, replacing each instruction by an abstract description (e.g., conjunction of linear constraints) that over-

$m(\langle x, b\rangle, \langle r\rangle) \leftarrow$
$\quad \{y = 1\},$
$\quad \{z = 0\},$
$\quad m_1(\langle x, b, y, z\rangle, \langle r\rangle).$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
$\quad \{b \leq 1\},$
$\quad \{r = z\}.$
$m_1(\langle x, b, y, z\rangle, \langle r\rangle) \leftarrow$
$\quad \{b > 1\},$
$\quad m_2(\langle x, b, y, z\rangle, \langle y_1, z_1\rangle),$
$\quad \{r = z_1\}.$

$m_2(\langle x, b, y, z\rangle, \langle y, z\rangle) \leftarrow$
$\quad \{y \geq x\}.$
$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow$
$\quad \{y < x\},$
$\quad \{z_1 = z + 1\},$
① $\{y_1 = \top\},$
$\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$

Fig. 2. Abstract compilation of the program of Figure 1

approximates its behavior; (2) a fixpoint computation step that computes an abstract semantics of the program; and (3) in the last, we prove termination or infer bounds on resource consumption using the abstract program of point 1 and the abstract semantics of point 2.

Applying the first step on the program of Figure 1 results in the abstract program of Figure 2. It can be observed that linear arithmetic instructions are precisely described by their corresponding abstract versions. For example, $z_1{:=}z+1$ updates z_1 to hold the value of $z+1$, and its corresponding abstract version $\{z_1 = z + 1\}$ is a denotation which states that the value of z_1 is equal to the value of z plus 1. However, in the case of non-linear arithmetic instructions, the abstract description often loses valuable information. This is the case of the instruction $y_1{:=}y * b$ which is annotated with ① in both Figures 1 and 2. While the instruction updates y_1 to hold the value of $y * b$, its abstract description $\{y_1 = \top\}$ states that y_1 can take any value. Here $\top$ is interpreted as any integer value. This makes it impossible to bound the number of iterations of the loop, since in the abstract program the function $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x-y)$ does not decrease in each two consecutive iterations.

Without any knowledge on the values of y and b, the constraint $\{y_1 = \top\}$ is indeed the best description for $y_1{:=}y * b$ when only conjunctions of linear constraints are allowed. However, in the program of Figure 1 it is guaranteed that the value of y is positive and that of b is greater than 1. Using this context information the abstraction of $y_1{:=}y{*}b$ can be improved to $\{y_1 \geq 2{*}y\}$, which in turn allows synthesizing the ranking function $f(\langle x, b, y, z\rangle) = \mathtt{nat}(x - y)$ and its refinement $f(\langle x, b, y, z\rangle) = \log_2(\mathtt{nat}(x - y) + 1)$. This suggests that the abstract compilation can benefit from context information when only conjunctions of linear constraints are allowed. However, the essence of abstract compilation is to use only syntactic information, and clearly context information cannot be obtained always by syntactic analysis of the program.

One way to solve the loss of precision when abstracting non-linear arithmetic instructions is to allow the use of disjunctions of linear constraints. For example, the instruction $y_1{:=}y * b$ could be abstracted to $\varphi_1 \vee \cdots \vee \varphi_n$ where each φ_i is a conjunction of linear constraints that describes a possible scenario. E.g., we could have $\varphi_j = \{y \geq 1, b \geq 2, y_1 \geq 2 * b\}$ in order to handle the case in which $y \geq 1$ and $b \geq 2$. Then, during the fixpoint computation, when the context becomes available, the appropriate φ_i will be automatically selected. However, for efficiency reasons, we restrict our value analysis to use only conjunctions of linear constraints. In order to avoid the use of disjunctive constraints, similarly to [21], we follow an approach that encodes the disjunctive information into the (abstract) program itself. For example, the second rule of m_2 would be abstracted to:

$$
\begin{array}{ll}
m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 0, c = 0\}. \\
\quad \{y < x\}, & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a = 1, c = b\}. \\
\quad \{z_1 = z + 1\}, & \qquad\qquad \vdots \\
\textcircled{1}\ \ op_*(\langle y, b\rangle, \langle y_1\rangle), & op_*(\langle a, b\rangle, \langle c\rangle) \leftarrow \{a \geq 2, b \geq 2, c \geq 2 * a\}. \\
\quad m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle). &
\end{array}
$$

Here, the instruction $y_1{:=}y * b$ was abstracted to $op_*(\langle y, b\rangle, \langle y_1\rangle)$ which is a call to an auxiliary abstract rule that defines possible abstract scenarios for different inputs. During the fixpoint computation, since op_* is called in a context in which $y \geq 1$ and $b \geq 2$, only the second and last rules of op_* will be selected. Then, these two rules propagate the constraint $y_1 \geq 2 * y$ back, which is required for synthesizing the expected ranking functions, without using disjunctive abstract domains.

4 Value Analysis

In this section we describe the value analysis of COSTA, which is based on the ideas presented in Section 3. The analysis receives as input a program in the intermediate language and a set of initial entries, and, for each (abstract) procedure $p(\bar{x}, \bar{y})$ it infers: (1) A pre-condition (over $\bar{x}$) that holds whenever p is called; and (2) a post-condition (over $\bar{x}$ and $\bar{y}$) that holds upon exit from p. The pre- and post-conditions are conjunction of linear constraints over the domain of Polyhedra [10]. Later, they can be composed in order to obtain invariants for some program points of interest.

In Section 4.1 we describe the abstract compilation step which translates the program P into an abstract version P^α. In Section 4.2 we describe a standard fixpoint algorithm that is used to infer the pre- and post-conditions. Finally, in Section 4.3 we explain how this information is used for bounding the number of iterations of the program's loops.

4.1 Abstract Compilation

This section describes how to transform a given program P into an abstract program P^α. In the implementation, we support also the abstraction of data-structures using the path-length measure [22] (the depth of a data-structure) and the abstraction of arrays to their length. However, in this paper we omit these features since they do not benefit from the techniques we use for abstracting non-linear arithmetic operations. Given a rule $\mathrm{r} \equiv \mathrm{p}(\bar{\mathrm{x}}, \bar{\mathrm{y}}) \leftarrow \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, the abstract compilation of r is $r^\alpha \equiv p(\bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha$, where:

(i) the abstract guard g^α is equal to the (linear) guard g;

(ii) if $\mathrm{b_i} \equiv \mathrm{q}(\bar{\mathrm{z}}, \bar{\mathrm{w}})$, then $b_i^\alpha \equiv q(\bar{z}, \bar{w})$;

(iii) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \in \{+, -\}$, then $b_i^\alpha \equiv \{x = e_1 \diamond e_2\}$; and

(iv) if $\mathrm{b_i} \equiv \mathrm{x{:=}e_1 \diamond e_2}$ and $\diamond \notin \{+, -\}$, then $b_i^\alpha \equiv op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$

Then, $P^\alpha = \{r^\alpha \mid \mathrm{r} \in P\}$. Note that we use the same names for constraint variables as those of the program variables (but in italic font for clarity). This is possible since we have assumed that the rules of P are given in SSA form. In the above abstraction, linear guards (point i) and linear arithmetic instructions (point iii) are simply replaced by a corresponding constraint that accurately model their behavior. Note that $\mathrm{x{:=}e_1 \diamond e_2}$ is an assignment while $\{x = e_1 \diamond e_2\}$ is an equality constraint. In point ii, calls to procedures are simply replaced by calls to abstract procedures. In what follows we explain the handling of non-linear arithmetic (point iv).

If the elements of the underlying abstract domain consist only in conjunctions of linear constraints, then non-linear operations are typically abstracted to $\top$. As we have seen in Section 3, this results in a significant loss of precision that prevents bounding the loop's iterations. A well-know solution is to use disjunctions of linear constraints which allow splitting the input domain into special cases that can be abstracted in a more accurate way. This can be done by directly using disjunctive abstract domains, however, this comes on the price of performance overhead. The solution we use in our implementation, inspired by [21], is to encode the disjunctions in the (abstract) program itself, without the need for using disjunctive abstract domains. In practice, this amounts to abstracting the non-linear arithmetic instruction $\mathrm{x{:=}e_1 \diamond e_2}$ into a call $op_\diamond(\langle e_1, e_2\rangle, \langle x\rangle)$ to an auxiliary abstract procedure $op_\diamond$, which is defined by several rules that cover all possible inputs and simulate the corresponding disjunction. The rules of $op_\diamond$ are designed by partitioning its input domain and, for each input class, define the strongest possible post-condition. Clearly, the more partitions there are, the more precise are the post-conditions, but the more expensive is the analysis too. Therefore, when designing the rules of $op_\diamond$ this performance and precision trade-off should be taken into account. For

the purposes of termination and resource usage analyzes, the partitioning of the input domain aims at propagating accurate information about constancy, equality and progression (e.g, multiplication by a constant), with the least possible number of rules. In what follows, we explain the auxiliary abstract procedures associated to the non-linear arithmetic operations of our language.

Integer division. The auxiliary abstract rule $op_{\texttt{rem}}$ and $op_{/}$ are defined in terms of $op_{\texttt{dr}}$ which stands for $x = y * q + r$:

$$
\begin{array}{|l|}
\hline
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = 0, q = 0, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = 1, q = x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{y = -1, q = -x, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = y, q = 1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x = -y, q = -1, r = 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > y > 1, 0 < q \leq \frac{x}{2}, 0 \leq r < y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > y > 1, \frac{x}{2} \leq q < 0, -y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{x > -y > 1, -\frac{x}{2} \leq q < 0, 0 \leq r < -y\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{-x > -y > 1, 0 < q \leq -\frac{x}{2}, y < r \leq 0\}. \\
op_{\texttt{dr}}(\langle x, y\rangle, \langle q, r\rangle) \leftarrow \{|y| > |x|, q = 0, r = x\}. \\
\hline
op_{/}(\langle x, y\rangle, \langle q\rangle) \quad \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle q, \_\rangle). \\
op_{\texttt{rem}}(\langle x, y\rangle, \langle r\rangle) \leftarrow op_{\texttt{dr}}(\langle x, y\rangle, \langle \_, r\rangle). \\
\hline
\end{array}
$$

Note that, in practice, abstract rules that involve $|\cdot|$ are folded into several cases. The sixth rule, for example, states that if $x > y > 1$ then x/y is a positive number smaller than or equal to $\frac{x}{2}$, and x rem y is a non-negative number smaller than y. This rule is also essential for synthesizing logarithmic ranking functions, when the input value is reduced at least by half in every iteration. Note that we ignore the special cases when $x =$ `MIN_VALUE` and $y = -1$, since it is a kind of overflow behavior.

Multiplication. The auxiliary abstract procedure op_{*} is defined as follows:

$$
\begin{array}{|l|}
\hline
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = 1, z = y\}. \\
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -y\}. \\
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \geq 2, y \geq 2, z \geq 2 * x, z \geq 2 * y\}. \\
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \geq 2, z \leq 2 * x, z \leq -2 * y\}. \\
op_{*}(\langle x, y\rangle, \langle z\rangle) \leftarrow \{x \leq -2, y \leq -2, z \geq -2 * x, z \geq -2 * y\}. \\
\hline
\end{array}
$$

We have omitted those rules that can be obtained by swapping the arguments x and y. In this abstraction, we distinguish the cases in which $x = 0$ (constancy), $x = \pm 1$ (equality) and those in which $|x| > 1$ and $|y| > 1$ (progress). Note that, for example, the post-condition $z \geq 2 * x$ is essential for finding a logarithmic

ranking function for loops like that of Figure 2. For example, it is not be possible to synthesize such ranking function if we use a weaker, yet sound, post-condition $z > x$.

The bitwise $\otimes$ and $\oplus$. The auxiliary abstract rules $op_{\otimes}$ and $op_{\oplus}$ are defined in terms of op_{ao} as follows:

$$
\begin{array}{|l|}
\hline
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = 0, a = 0, o = y\}. \\
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = -1, a = y, o = -1\}. \\
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x = y, a = x, o = x\}. \\
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > y > 0, 0 \leq a \leq y, o \geq x\}. \\
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x > 0, y < -1, 0 \leq a \leq x, y \leq o \leq -1\}. \\
op_{\text{ao}}(\langle x, y\rangle, \langle a, o\rangle) \leftarrow \{x < y < -1, a \leq x, y \leq o \leq -1\}. \\
\hline
op_{\otimes}(\langle x, y\rangle, \langle a\rangle) \quad \leftarrow op_{\text{ao}}(\langle x, y\rangle, \langle a, \_\rangle). \\
op_{\oplus}(\langle x, y\rangle, \langle o\rangle) \quad \leftarrow op_{\text{ao}}(\langle x, y\rangle, \langle \_, o\rangle). \\
\hline
\end{array}
$$

Since these operations are commutative we omit rules derivable by swapping the input arguments. The first two rules describe the cases $x = 0$ and $x = -1$, i.e., vectors in which all bits are respectively 0 or 1. The third rule handles the case $x = y$. The rest of rules are based on that the result of $x \otimes y$ has less 1-bits than either x or y, whereas the result of $x \oplus y$ has more 1-bits than either x or y.

Shift left and right. Although shift operations in Java bytecode accept any integer value as the shift operand, the number of shifted positions is determined only by the five least significant bits, i.e., it is a value between 0 and $2^5 - 1$ (for type `long` it is determined by the six least significant bits). For the shift left operation $\triangleleft$, the auxiliary abstract procedure $op_{\triangleleft}$ is defined as follows:

$$
\begin{array}{|l|}
\hline
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, z \geq 2x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, 0 < |s| < 2^5, z \leq 2x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, z \geq x\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, z \leq x\}. \\
\hline
\end{array}
$$

The above rules provide an accurate post-condition when the shift operand s satisfies $0 \leq |s| < 2^5$. In the last two abstract rules, the post-conditions are respectively $z \geq x$ and $z \leq x$ since we cannot observe the value of the first five bits of s when $|s| \geq 2^5$. Similarly, for the shift right operation $\triangleright$, the auxiliary abstract rule $op_{\triangleright}$ is defined as follows:

$$
\begin{array}{l}
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = 0, z = 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x = -1, z = -1\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{s = 0, z = x\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, 0 < |s| < 2^5, x > z, x \geq 2z, z \geq 0\}. \\
op_{\triangleright}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < -1, 0 < |s| < 2^5, x - 1 \leq 2z, z < 0\}. \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x > 0, |s| \geq 2^5, 0 \leq z \leq x\} \\
op_{\triangleleft}(\langle x, s\rangle, \langle z\rangle) \leftarrow \{x < 0, |s| \geq 2^5, x \leq z \leq -1\}\}.
\end{array}
$$

Note that when the program includes several non-linear instructions for the same operations, then it might be useful to generate different auxiliary abstract procedures for them, e.g, op^1_*, op^2_*, etc. This is required mainly when the calling contexts of these instructions are disjoint, and therefore separating their auxiliary abstract procedures avoids merging the calling contexts, which usually results in a loss of precision. In addition, non-linear arithmetic instructions that do not affect the termination of the program can be abstracted as before, i.e., to $\{x = \top\}$, and thus avoid the performance overhead caused by unnecessary auxiliary abstract procedures. These instructions can be identified using dependency analysis, similar to what have been done in [4] for identifying program variables that affect termination.

4.2 Fixpoint algorithm

Algorithm 1 implements the value analysis using a top-down strategy in the style of [7]. It receives as input an abstract program P^α and a set of initial pre-conditions E, and computes pre- and post-conditions for each procedure in P (stored in tables PRE and POST respectively). The meaning of a pre-condition $\mathsf{PRE}[q(\bar{x})] \equiv \varphi$, is that φ holds when calling q, and of a post-condition $\mathsf{POST}[q(\bar{x}, \bar{y})] \equiv \varphi$ is that φ holds upon exit from q.

Procedure FIXPOINT initializes the event queue $\mathcal{Q}$ to $\emptyset$ (L2), initializes the elements of tables PRE and POST to *false* (L4 and L5), processes the initial pre-conditions E by calling ADD_PRE for each one (L6) which in turn adds the corresponding event to $\mathcal{Q}$, and then in the while loop it processes the events of $\mathcal{Q}$ until no more events are available. In each iteration, an event q (a procedure name) is removed from $\mathcal{Q}$ (L8) and processed as follows: the current pre-condition ψ of q is retrieved (L9), each of the rules of q is evaluated in order to generate a post-condition for that specific rule w.r.t. ψ (L11), all post-conditions are joint into a single element δ (using the least upper-bound $\sqcup$ of the underlying abstract domain), and finally δ is added as a post-condition for q by calling ADD_POST. Note that the call to ADD_POST might add more events to $\mathcal{Q}$. The evaluation of a rule (procedure EVALUATE) w.r.t. a pre-condition ψ processes each b^α_i in the rule's body B as follows: if b^α_i is a call $q'(\bar{w}, \bar{z})$, then it registers the corresponding pre-condition by calling ADD_PRE (L16) and adds

Algorithm 1 The fixpoint algorithm

```
1:  procedure FIXPOINT(P^α, E)
2:      Q = ∅;
3:      for all q(x̄, ȳ) ∈ P do
4:          PRE[q(x̄)] = false;
5:          POST[q(x̄, ȳ)] = false;
6:      for all ⟨p(x̄), φ⟩ ∈ E do ADD_PRE(p(x̄), φ);
7:      while Q.notempty() do
8:          q = Q.poll();
9:          ψ = PRE[q(x̄)];
10:         δ = false;
11:         for all q(x̄, ȳ) ← B^α ∈ P^α do δ = δ ⊔ EVALUATE(q(x̄, ȳ) ← B^α, ψ);
12:         ADD_POST(q(x̄, ȳ), δ);
13: function EVALUATE(q(x̄, ȳ) ← B^α, ψ)
14:     for all b_i^α ∈ B^α do
15:         if b_i^α ≡ q'(w̄, z̄) then
16:             ADD_PRE(q'(w̄), ∃̄w̄.ψ);
17:             ψ = ψ ⊓ POST[q'(w̄, z̄)];
18:         else ψ = ψ ⊓ b_i^α;
19:     return ∃̄x̄ ∪ ȳ.ψ;
20: procedure ADD_PRE(q(x̄), φ)
21:     ψ = PRE[q(x̄)];
22:     if φ ⊭ ψ then
23:         PRE[q(x̄)] = ψ ⊔ φ;
24:         Q.add(q);
25: procedure ADD_POST(q(x̄, ȳ), φ)
26:     δ = POST[q(x̄, ȳ)];
27:     if δ ⊭ φ then
28:         POST[q(x̄, ȳ)] = δ ⊔ φ;
29:         for all p ∈ P do
30:             if p calls q then Q.add(p);
```

the current post-condition of q to ψ (L17); otherwise, b_i^α is a constraint and it simply adds it to ψ (L18).

Procedure ADD_PRE adds a new pre-condition for q if it does not imply the current one, and adds the corresponding event to $\mathcal{Q}$. Procedure ADD_POST adds a new post-condition for q if it does not imply the current one, and adds events for all procedures that call q since they might have to be re-analyzed. Note that both procedures use the least upper bound $\sqcup$ of the underlying abstract domain in order to join the new pre- or post-conditions with the current one. Note also that since we use abstract domains with infinite ascending chains, in

practice, these procedures incorporate a widening operator in order to ensure termination.

Example 4.1 Consider again the abstract program of Figure 2, where the second abstract rule of m_2 is replaced by

$$m_2(\langle x, b, y, z\rangle, \langle y_2, z_2\rangle) \leftarrow \\ \{y < x\},\ \{z_1 = z + 1\},\ op_*(\langle b, y\rangle, \langle y_1\rangle),\ m_2(\langle x, b, y_1, z_1\rangle, \langle y_2, z_2\rangle).$$

and the initial set of entries $E = \{\langle m(\langle x, b\rangle), true\rangle\}$. Then, the fixpoint algorithm infers $\mathsf{PRE}[m_2(\langle x, b, y, z\rangle)] = \{z \geq 0, y \geq 1, b \geq 2\}$, $\mathsf{PRE}[op_*(\langle b, y\rangle)] = \{b > 1, y \geq 1\}$, and $\mathsf{POST}[op_*(\langle b, y\rangle, \langle y_1\rangle)] = \{y_1 \geq 2 * y\}$.

4.3 Bounding the loops

In this section we describe how the abstract program and the pre- and post-conditions are used in order to bound the program's loops, as done in [1]. Briefly, for each abstract rule $p(\bar{x}, \bar{y}) \leftarrow g^\alpha, b_1^\alpha, \ldots, b_n^\alpha \in P^\alpha$, we generate a set of transitions

$$\left\{ \langle p(\bar{x}) \rightarrow q(\bar{w}), \bar{\exists}\bar{x} \cup \bar{w}.\varphi\rangle \;\middle|\; \begin{array}{l} i \in [1, \ldots, n],\ b_i^\alpha = q(\bar{w}, \bar{z}), \\ \varphi = \ \mathsf{PRE}[q(\bar{x})] \wedge g^\alpha \wedge \phi(b_1^\alpha) \cdots \wedge \phi(b_{i-1}^\alpha) \end{array} \right\}$$

where $\bar{\exists}\bar{x} \cup \bar{w}.\varphi$ is the projection of φ on the variables $\bar{x} \cup \bar{w}$; $\phi(b_i^\alpha) = b_i^\alpha$ if b_i^α is a constraint; and $\phi(b_i^\alpha) = \mathsf{POST}[b_i^\alpha]$ if b_i^α is a call. Then, the set of all transitions is passed to, for example, the tool of [2], which in turn infers ranking functions for the corresponding loops.

Example 4.2 Using the abstract rule and the pre- and post-conditions of Example 4.1, we generate the transition relation

$$\langle m_2(\langle x, b, y, z\rangle) \rightarrow m_2(\langle x, b, y_1, z_1\rangle), \varphi\rangle$$

where $\varphi = \{z \geq 0, y \geq 1, b \geq 2, x < y, z_1 = z + 1, y_1 \geq 2 * y\}$. Then, the solver of [2] infers the expected ranking functions as explained in Section 3.

5 Experimental Evaluation

We have implemented, in the context of COSTA [3], a prototype of the value analysis described in Section 4. We have performed some experiments on typical examples from the literature that use non-linear and bit arithmetic operations. The benchmarks are available at `http://costa.ls.fi.upm.es/papers/bytecode2011`. Unfortunately, the implementation cannot be tried out via COSTA's web-interface since it has not been integrated in the main branch yet.

costa, with the new value analysis, was able to prove termination of all benchmarks. Note that without this value analysis costa could not handle any of these benchmarks. We have also analyzed the benchmarks using other termination analyzers for Java bytecode. Julia [1] [22] was not able to prove termination of any of these benchmarks. AProVE [2] [12] could not prove termination of programs with bit arithmetic operations, but could handle programs with non-linear arithmetic operations such as multiplication and integer division, except for the program of Figure 1 for which it could not complete the proof in a time limit of 5 minutes. In what follows we explain the results of our analysis on some of the benchmarks.

EX1: We start with an example borrowed from [9]:

```
void and(int x){
  while( x > 0)
    x = x & x-1;
}
```

$$
\begin{array}{l}
and(\langle x\rangle,\langle\rangle) \leftarrow and_1(\langle x\rangle,\langle\rangle).\\
and_1(\langle x\rangle,\langle\rangle) \leftarrow \{x \leq 0\}.\\
and_1(\langle x\rangle,\langle\rangle) \leftarrow \{x > 0\},\\
\quad \{y = x-1\},\\
\quad op_{\otimes}(\langle x,y\rangle,\langle x_1\rangle),\\
\quad and_1(\langle x_1\rangle,\langle\rangle).
\end{array}
$$

The code on the right is the abstract compilation of the corresponding intermediate representation of the Java method. In order to bound the number of iterations of the **while** loop, it is essential to infer that the value of x decreases in each iteration. This cannot be guaranteed when considering the instruction x=x & x−1 separately, since, for example, it does not decrease when x=0. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\otimes}(x,y)] = \{y = x-1, x > 0\}$, i.e., the context $x > 0$ is available when calling $op_{\otimes}$, which in turn makes it possible to infer the post-condition $\mathsf{POST}[op_{\otimes}(\langle x,y\rangle,\langle x_1\rangle)] = \{y = x-1, x > 0, 0 \leq x_1 \leq x-1\}$. Using this information we generate the transition $\langle and_1(\langle x\rangle) \rightarrow and_1(\langle x_1\rangle), \{x > 0, 0 \leq x_1 \leq x-1\}\rangle$ for which we synthesize the ranking function $f(\langle x\rangle) = \mathtt{nat}(x)$.

EX2: The next example implements the Euclidean algorithm for computing the greatest common divisor of two natural numbers. It is taken from the Java bytecode termination competition database [3]:

[1] using the online version http://julia.scienze.univr.it/
[2] using the online version http://aprove.informatik.rwth-aachen.de/
[3] http://termcomp.uibk.ac.at

```
int gcd(int a, int b){
  int tmp;
  while(b>0 && a>0){
    tmp = b;
    b = a % b;
    a = tmp;
  }
  return a;
}
```

$gcd(\langle a,b\rangle,\langle r\rangle) \leftarrow gcd_1(\langle a,b\rangle,\langle r\rangle).$
$gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{a \leq 0\}.$
$gcd_1(\langle a,b\rangle,\langle a\rangle) \leftarrow \{b \leq 0\}.$
$gcd_1(\langle a,b\rangle,\langle r\rangle) \leftarrow$
$\{a > 0, b > 0\},$
$\{tmp = b\},$
$op_{\texttt{rem}}(\langle a,b\rangle,\langle b_1\rangle),$
$\{a_1 = tmp\},$
$gcd_1(\langle a_1,b_1\rangle,\langle r\rangle).$

costa was not able to prove termination of this program in the competition of July 2010, mainly because it ignores the calling context when abstracting b=a % b, and therefore it cannot infer that b decreases. Our analysis infers the pre-condition $\mathsf{PRE}[op_{\texttt{rem}}(\langle a,b\rangle)] = \{a > 0, b > 0\}$, which in turn makes it possible to infer the post-condition $\mathsf{POST}[op_{\texttt{rem}}(\langle a,b\rangle,\langle b_1\rangle)] = \{a > 0, b > 0, b > b_1\}$. Using this information we generate the transition $\langle gcd_1(\langle a,b\rangle) \rightarrow gcd_1(\langle a_1,b_2\rangle), \{a > 0, b > 0, b > b_1\}\rangle$ for which we synthesize the ranking function $f(\langle a,b\rangle) = \texttt{nat}(b)$.

EX3: The next example is taken from the method `toString(int i, int radix)` of class `java.lang.Integer`. It is used for writing a number in any numeric base. For simplicity, we have removed code that does not affect the termination, and annotated the loop with a pre-condition that is inferred by our analysis:

```
// {i <= 0, 2 <= radix}
while (i <= -radix) {
    i = i / radix;
}
```

$p(\langle i, radix\rangle,\langle\rangle) \leftarrow \{i > -radix\}.$
$p(\langle i, radix\rangle,\langle\rangle) \leftarrow$
$\{i \leq -radix\},$
$op_{/}(\langle i, radix\rangle,\langle i_1\rangle),$
$p(\langle i_1, radix\rangle,\langle\rangle).$

Due to the pre-condition $\mathsf{PRE}[op_{/}(\langle i, radix\rangle,\langle i_1\rangle)] = \{2 \leq radix, i \leq -radix\}$, our analysis infers the post-condition $\mathsf{POST}[op_{/}(\langle i, radix\rangle,\langle i_1\rangle)] = \{2 \leq radix, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}$. Using this post-condition we generate the transition $\langle p(\langle i, radix\rangle) \rightarrow p(\langle i_1, radix\rangle), \{2 \leq radix, i \leq -radix, \frac{i}{2} \leq i_1 < 0\}\rangle$. For this transition we synthesize the ranking function $f(\langle i, radix\rangle) = \log_2(\texttt{nat}(-i)+1)$.

EX4: The next example is a variation of a loop from the class `Integer` in the method `toUnsignedString(int i, int shift)`, which is used for writing a number in binary, octal or hexadecimal form:

```
// { 1 <= shift <= 4 }
while ( i > 0 ) {
    i >>= shift;
}
```

$$
\begin{array}{l}
p(\langle i, \mathit{shift}\rangle, \langle\rangle) \leftarrow \{i \leq 0\}. \\
p(\langle i, \mathit{shift}\rangle, \langle\rangle) \leftarrow \\
\quad \{i > 0\}, \\
\quad op_{\triangleright}(\langle i, \mathit{shift}\rangle, \langle i_1\rangle), \\
\quad p(\langle i_1, \mathit{shift}\rangle, \langle\rangle).
\end{array}
$$

Due to the pre-condition $\mathsf{PRE}[op_{\triangleright}(\langle i, \mathit{shift}\rangle] = \{i > 0, 1 \leq \mathit{shift} \leq 4\}$, our analysis infers the post-condition $\mathsf{POST}[op_{\triangleright}(\langle i, \mathit{shift}\rangle, \langle i_1\rangle)] = \{i > 0, 1 \leq \mathit{shift} \leq 4, i \geq 2 * i_1, i_1 \geq 0\}$. Using this postcontidion we generate the transition $\langle p(\langle i, \mathit{shift}\rangle) \rightarrow p(\langle i_1, \mathit{shift}\rangle), \{i > 0, 1 \leq \mathit{shift} \leq 4, i \geq 2 * i_1, i_1 \geq 0\}\rangle$, for which we synthesize the ranking function $f(\langle i, \mathit{shift}\rangle) = \log_2(\mathtt{nat}(i) + 1)$.

6 Conclusions

In this paper we have described how we handle non-linear arithmetic instructions in the value analysis of COSTA. It is well-know that handling such operations is problematic when the underlying abstract domain allows only the use of conjunctions of linear constraints. It is also well-know that the use of disjunctive abstract domains is a possible solution to this problem, however, on the price of performance overhead. In this paper, instead of using disjunctive abstract domains, we encoded the disjunctive nature of non-linear arithmetic instructions into the abstract program itself. This encoding, when combined with a value analysis that is based on non-disjunctive abstract domains such as Polyhedra or Octagons, makes it possible to dynamically select the best abstraction depending on the context from which the code that correspond to the encoding was reached. Our experiments demonstrate that COSTA is now able to prove termination and infer bound on resource consumption for programs that it could not handle before. For future work, we plan to improve the scalability of the analyzer, support overflow in arithmetic operations, and support floating point arithmetic. Note that, given the latest developments in the Parma Polyhedra Library [6], supporting overflow and floating point arithmetic is relatively straightforward.

Acknowledgement

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project. Diego Alonso is partially supported by the UCM PhD scholarship program.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In Gilles Barthe and Frank de Boer, editors, *IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18, Oslo, Norway, June 2008. Springer-Verlag, Berlin.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *6th International Symposioum on Formal Methods for Components and Objects (FMCO'08)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV08)*, pages 368–375, Fortaleza, Brasil, March 2008. ACM Press, New York.

[5] A. W. Appel. Ssa is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[7] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program.*, 38(3):355–370, 1999.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426, Tucson, Arizona, USA, 2006.

[9] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 2010.

[10] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL'78)*. ACM Press, 1978.

[11] R. W. Floyd. Assigning Meanings to Programs. In J.T Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI, 1967.

[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE. In *Proc. of 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume LNCS 3091, pages 210–220. Springer-Verlag, 2004.

[13] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In *Symposium on Principles of Programming Languages (POPL'09)*, pages 127–139. ACM, 2009.

[14] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010.

[15] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[16] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode'07)*, Electronic Notes in Theoretical Computer Science, pages 35–50. Elsevier, 2007.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[18] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[19] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[20] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of java bytecode by term rewriting. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[21] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static Analysis in Disjunctive Numerical Domains. In *Static Analysis, 13th International Symposium, (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006.

[22] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *Proc. of CASCON'99*, pages 125–135. IBM, 1999.