Foundations of Software Technology and Theoretical Computer Science (Bangalore) 2008.
Editors: R. Hariharan, M. Mukund, V. Vinay; pp 316-327

Analyzing the Implicit Computational Complexity of object-oriented programs

.

Jean-Yves Marion[1], **Romain Péchoux**[2]

[1] LORIA, Carte team, and ENSMN, INPL, Nancy-Université,
Nancy, France.
jean-yves.marion@loria.fr

[2] Department of Computer Science, Trinity College,
Dublin, Ireland
romain.pechoux@cs.tcd.ie

ABSTRACT. A sup-interpretation is a tool which provides upper bounds on the size of the values computed by the function symbols of a program. Sup-interpretations have shown their interest to deal with the complexity of first order functional programs. This paper is an attempt to adapt the framework of sup-interpretations to a fragment of object-oriented programs, including loop and while constructs and methods with side effects. We give a criterion, called brotherly criterion, which uses the notion of sup-interpretation to ensure that each brotherly program computes objects whose size is polynomially bounded by the inputs sizes. Moreover we give some heuristics in order to compute the sup-interpretation of a given method.

1 Introduction

Computer security is defined as ensuring confidentiality, integrity and availability requirements in whatever context [6]. For example, a secured system should resist to a buffer-overflow. In this paper, we focus on analyzing the complexity of object-oriented programs, that is the number of objects created by a program during its execution, by static analysis. For that purpose, we use semantics interpretation tools called sup-interpretations. Sup-interpretations were introduced in [14, 15] in order to study the complexity of first order functional programs. A sup-interpretation consists in a function which provides an upper bound on the size of the values computed by some symbol of a given program. The notion of sup-interpretation strictly generalizes the notion of quasi-interpretation [8] (i.e. analyzes the complexity of strictly more algorithms) which has already been used to perform Bytecode verification [4] and which has been extended to reactive programs [5, 10]. Sup-interpretations allow to characterize complexity classes and, in particular, the class of NC^k functions [7, 16].

A major challenge consists in the adaptation of such an analysis to object-oriented programs with respect to the following points. Firstly, we have to carefully translate the notion of sup-interpretation from the functional paradigm to the object-oriented paradigm, taking into account the new object features such as method calls or side effects. Secondly, we also want to ensure the viability of our study by obtaining heuristics to compute sup-interpretations.

FSTTCS 2008
IARCS Annual Conference on
Foundations of Software Technology and Theoretical Computer Science
http://drops.dagstuhl.de/opus/volltexte/2008/1763

The considered language is inspired by the Featherweight Java of [12] and is a fragment of the Java language of [11] which includes side effects and to which we add loop and while constructs. This language is a purely object-oriented language like SmallTalk. For simplicity, inheritance, typing and subtyping are not considered in this paper. However, the analysis presented in this paper can be extended without restriction to a Java-like language including primitives types such as characters, integers or booleans.

Our work is a continuation of recent studies on the Implicit Computational Complexity of imperative programs [18, 13]. Contrarily to these seminal works, we work on polynomial algebra instead of matrix algebra. There are at least two reasons for such an approach. Firstly, the use of polynomials gives a clearest intuition and pushes aside a lot of technicalities. Secondly, polynomials give more flexibility in order to deal with method calls, which is essential in order to study the object oriented paradigm. Some studies on the cost analysis of Java Bytecode have already been developed in [1, 2]. In this paper, we make a distinct choice by considering a more formal and restricted language. We perform the analysis at the language level and not at the Bytecode level. The pros are that our study has more formal basis and more portability (i.e. it can easily be adapted to distinct object-oriented languages). The cons are the restrictions on the considered language. However, these restrictions are put in order to make the study more comprehensible and we claim that they could be withdraw without any difficulty.

The paper is organized as follows. After introducing our language and the notion of sup-interpretation of an object-oriented program, we give a criterion, called brotherly criterion, which ensures that each brotherly program computes objects whose size is polynomially bounded by the input size. Then, we extend this criterion to methods, thus obtaining heuristics for synthesizing sup-interpretations of non-recursive methods.

2 Object-oriented Programs

2.1 Syntax of programs

A program is composed by a sequence of classes, including a main class, which are named by class identifiers in $\mathtt{Class}$. A class $\mathtt{C} \in \mathtt{Class}$ is composed by a sequence of attribute declarations, a constructor and a sequence of methods. The main class $\mathtt{main}$ is only composed by attribute declarations and commands, i.e. there is no method and no constructor in $\mathtt{main}$. $\mathtt{var}\ X$; corresponds to the declaration of the attribute X, where X represents a field of a given class and belongs to a fixed set $\mathcal{X}$. A method is composed by a method identifier $\mathtt{f}$ belonging to a set $\mathcal{F}$, a sequence of arguments $x_1, \ldots, x_n \in \mathcal{P}$, also called parameters, and a command $\mathtt{Cm}$ and is of the shape $\mathtt{f}(x_1, \ldots, x_n)\,\{\mathtt{Cm}\,;\ \mathtt{return}\ X;\}$, where the attribute X corresponds to the field returned as output. A constructor $\mathtt{C}(x_1, ..., x_n)\,\{X_1 := x_1; \ldots ; X_n := x_n\}$ assigns a parameter to each attribute of the corresponding class. Throughout the paper, we use capital letters X, Y, Z and lower-case letters x, y, z in order to make the distinction between attributes and, respectively, parameters. A command is either the skip command, a variable assignment, a sequence of commands $\mathtt{Cm}_1;\mathtt{Cm}_2$, a loop command, a while command or a conditional command. An expression is either a parameter x, an attribute X, the null reference or the creation of a new object using a constructor. A method call is of the shape

$X.\mathtt{f}(e_1,\ldots,e_n)$, with $\mathtt{f} \in \mathcal{F}$, $X \in \mathcal{X}$ and with $e_1,\ldots,e_n$ expressions. The precise syntax of the language is summed up by the following grammar:

`Attributes`	$\ni A$	$::= \mathtt{var}\ X; \mid \mathtt{var}\ X; A$
`Expressions`	$\ni e$	$::= x \mid X \mid \mathtt{null} \mid \mathtt{new}\ \mathtt{C}(e_1,\ldots,e_n)$
`Method call`	$\ni a$	$::= X.\mathtt{f}(e_1,\ldots,e_n)$
`Commands`	$\ni \mathtt{Cm}$	$::= \mathtt{skip} \mid X := a \mid X := e \mid \mathtt{Cm}_1;\mathtt{Cm}_2 \mid \mathtt{loop}\ X\{\mathtt{Cm}\}$
		$\mid \mathtt{if}(e)\mathtt{then}\{\mathtt{Cm}_1\}\mathtt{else}\{\mathtt{Cm}_2\} \mid \mathtt{while}\ e\{\mathtt{Cm}\}$
`Methods`	$\ni \mathtt{M}$	$::= \mathtt{f}(x_1,\ldots,x_n)\{\mathtt{Cm}\ ;\ \mathtt{return}\ X;\}$
`Constructors`	$\ni \mathtt{Cons}$	$::= \mathtt{C}(x_1,...,x_n)\{X_1 := x_1;\ldots;X_n := x_n\}$
`Class`	$\ni \mathtt{C}$	$::= \mathtt{Class}\ \mathtt{C}\{A\ \mathtt{Cons}\ M_1\ldots M_n\}$
	`main`	$::= \mathtt{Class}\ \mathtt{main}\{A\ \mathtt{Cm}\}$

Notation 1 *We will use the notation $\bar{e}$ to represent the sequence $e_1,\ldots,e_n$ when n is clear from the context.*

The sets $\mathcal{X}$, $\mathcal{P}$, $\mathcal{F}$ and `Class` are pairwise disjoint. All attributes occurring in the methods of a given class `C` must belong to the attributes of this class. All parameters occurring in the command `Cm` of a given method must belong to the parameters $x_1,\ldots,x_n$. Let $\mathcal{F}_{\mathtt{C}}$ and $\mathcal{X}_{\mathtt{C}}$ be respectively the sets of methods and attributes declared in the class `C`.

We add the following syntactic restrictions to our language: We suppose that $\mathtt{C} \neq \mathtt{C}'$ implies $\mathcal{F}_{\mathtt{C}} \cap \mathcal{F}_{\mathtt{C}'} = \mathcal{X}_{\mathtt{C}} \cap \mathcal{X}_{\mathtt{C}'} = \emptyset$. There is no method overloading. A program is not allowed to write the attribute X during the execution of a $\mathtt{loop}\ X\{\mathtt{Cm}\}$. There are neither local variables, nor static variables. All these restrictions are put in order to simplify the discussion. However we claim that they also could be analyzed by our framework.

Example 1 (Linked list) *Consider the linked list class described in figure 1. X and Y represent the head and tail attributes whereas W and Z store intermediate computations. Notice that W and Z are required since the considered language has no local variables.*

```
  Class List{ var X; var Y; var W; var Z;
List(x,y,w,z){X := x; Y := y; W := w; Z := z;}
        getHead(){skip ; return X;}
        getTail(){skip ; return Y;}
       setTail(y){Y := y ; return X;}
    reverse(){   Z := new List(X,null);
                  W := Y;
                  loop Y{
          Z := new List(W.getHead(),Z,null);
                W := W.getTail();
                      };
                  return Z; }
                      }
```

Figure 1: Linked list

2.2 Semantics

In this section, we define a semantics without references. This semantic weakening is not a hard restriction since we are more concerned with providing a semantics which takes into account the number of object creations than by giving a precise semantics of object-oriented programs, as it will be illustrated by remark 2.2. The domain of computation is the set of objects (values) described in [12] and is defined inductively by:

$$\texttt{Objects} \ni o ::= \texttt{null} \mid \texttt{new C}(o_1,\ldots,o_n)$$

where $\texttt{C} \in \texttt{Class}$ is a class having n attributes and $o_1,\ldots,o_n$ are objects. Notice that objects are particular expressions, only using class constructors.

Definition 1.*[Size] The size $|o|$ of an object o is defined inductively by $|o| = 0$, if $o = \texttt{new C}()$, and $|o| = \sum_{i=1}^{n} |o_i| + 1$, if $o = \texttt{new C}(o_1,\ldots,o_n)$.*

Objects are created through explicit requests, using a constructor and the `new` construct. Consequently, an attribute X may be successively attached to distinct objects during the program execution. The operational semantics of our language is inspired by the operational semantics of the Java fragment given in [11]. It is closer to [11] than to [12] since we use variable assignments (i.e. there are side effects). Contrarily to [11], we do not make explicit use of references since the object description suggested above is sufficient to control program complexity (i.e. the number of object creations). In general, an object of the shape $\texttt{new C}(o_1,\ldots,o_n)$ can be viewed as an object of the class `C` with n implicit references to the objects $o_1,\ldots,o_n$.

A store σ is a partial mapping from attributes $\mathcal{X}$ and parameters $\mathcal{P}$ to objects in `Objects`. A store can be extended to expressions and method calls by $\texttt{null}\sigma = \texttt{null}$, $\texttt{new C}(e_1,\ldots,e_n)$ $\sigma = \texttt{new C}(e_1\sigma,\ldots,e_n\sigma)$ and $X.\texttt{f}(e_1,\ldots,e_n)\sigma = X\sigma.\texttt{f}(e_1\sigma,\ldots,e_n\sigma)$. Given a store σ, the notation $\sigma\{\diamond_1 \leftarrow o_1,...,\diamond_n \leftarrow o_n\}$ means that the object stored in $\diamond_i \in \mathcal{X}\cup\mathcal{P}$ is updated to the object o_i in σ, for each $i \in \{1,n\}$. Given an expression (or a method call) d and a store σ, the notation $\langle d,\sigma\rangle \downarrow \langle o,\sigma'\rangle$ means that d is evaluated to o and that the store σ is updated to the store σ' during this evaluation. Given a command `Cm`, we use the notation $\langle \texttt{Cm},\sigma\rangle \downarrow \langle\sigma'\rangle$, if σ is updated to σ' during the execution of `Cm`. Given a program `p` of main class $\texttt{Class main}\ \{A;\texttt{Cm}\}$ and a store σ, `p` computes a store σ' defined by $\langle \texttt{Cm},\sigma\rangle \downarrow \langle\sigma'\rangle$.

The expression `null` is evaluated to `null`. Given a store σ, a variable or a parameter $\diamond$ is evaluated to $\diamond\sigma$. The expression $\texttt{new C}(e_1,\ldots,e_n)$ is evaluated to $\texttt{new C}(o_1,\ldots,o_n)$, if the expressions $e_1,\ldots,e_n$ are evaluated to the objects $o_1,\ldots,o_n$. The operational semantics of expressions is described in figure 2.

$$\frac{}{\langle \diamond,\sigma\rangle \downarrow \langle \diamond\sigma,\sigma\rangle}\ \diamond \in \mathcal{X}\cup\mathcal{P} \qquad\qquad \frac{}{\langle \texttt{null},\sigma\rangle \downarrow \langle \texttt{null},\sigma\rangle}$$

$$\frac{\forall i \in \{1,n\}\ \ \langle e_i,\sigma\rangle \downarrow \langle o_i,\sigma\rangle}{\langle \texttt{new C}(e_1,\ldots,e_n),\sigma\rangle \downarrow \langle \texttt{new C}(o_1,\ldots,o_n),\sigma\rangle}\ \texttt{C} \in \texttt{Class}$$

Figure 2: Operational semantics of an expression

The command `skip` does nothing. The command $X := d$ assigns the object computed by d to the attribute X in the store. The command $\texttt{Cm}_1;\texttt{Cm}_2$ corresponds to the sequential execution of $\texttt{Cm}_1$ and $\texttt{Cm}_2$. $\texttt{if}(e)\texttt{then}\{\texttt{Cm}_1\}\texttt{else}\{\texttt{Cm}_2\}$ executes either the command $\texttt{Cm}_1$ or the command $\texttt{Cm}_2$ depending on whether the expression e is evaluated to the object `null` or to any other object. The command $\texttt{loop}\ X\{\texttt{Cm}\}$ executes $|o|$ times the command `Cm`, if o is the object stored in X. Finally, the command $\texttt{while}\ e\,\{\texttt{Cm}\}$ is evaluated to $\texttt{skip}$, if e is evaluated to the object $\texttt{null}$, and to $\texttt{Cm};\texttt{while}\ e\,\{\texttt{Cm}\}$ otherwise. The operational semantics of commands is described in figure 3.

$$\frac{}{\langle \texttt{skip},\sigma\rangle \downarrow \langle \sigma\rangle} \qquad \frac{\langle d,\sigma\rangle \downarrow \langle o,\sigma'\rangle}{\langle X := d\rangle \downarrow \langle \sigma'\,\{X \leftarrow o\}\rangle}$$

$$\frac{\langle e,\sigma\rangle \downarrow \langle \texttt{null},\sigma\rangle}{\langle \texttt{if}(e)\texttt{then}\{\texttt{Cm}_1\}\texttt{else}\{\texttt{Cm}_2\},\sigma\rangle \downarrow \langle \texttt{Cm}_1,\sigma\rangle} \qquad \frac{\langle e,\sigma\rangle \downarrow \langle o,\sigma\rangle \quad o \neq \texttt{null}}{\langle \texttt{if}(e)\texttt{then}\{\texttt{Cm}_1\}\texttt{else}\{\texttt{Cm}_2\},\sigma\rangle \downarrow \langle \texttt{Cm}_2,\sigma\rangle}$$

$$\frac{\langle \texttt{Cm}_1,\sigma\rangle \downarrow \langle \sigma'\rangle \quad \langle \texttt{Cm}_2,\sigma'\rangle \downarrow \langle \sigma''\rangle}{\langle \texttt{Cm}_1;\texttt{Cm}_2,\sigma\rangle \downarrow \langle \sigma''\rangle} \qquad \frac{\langle X,\sigma\rangle \downarrow \langle o,\sigma\rangle}{\langle \texttt{loop}\ X\,\{\texttt{Cm}\},\sigma\rangle \downarrow \langle\, \underbrace{\texttt{Cm};\ldots;\texttt{Cm}}_{|o|\ \text{times}},\sigma\,\rangle}$$

$$\frac{\langle e,\sigma\rangle \downarrow \langle o,\sigma\rangle \quad o \neq \texttt{null}}{\langle \texttt{while}\ e\,\{\texttt{Cm}\},\sigma\rangle \downarrow \langle \texttt{Cm};\texttt{while}\ e\,\{\texttt{Cm}\},\sigma\rangle} \qquad \frac{\langle e,\sigma\rangle \downarrow \langle \texttt{null},\sigma\rangle}{\langle \texttt{while}\ e\,\{\texttt{Cm}\},\sigma\rangle \downarrow \langle \texttt{skip},\sigma\rangle}$$

Figure 3: Operational semantics of a command

If $\texttt{f}$ is a method defined by $\texttt{f}(x_1,\ldots,x_m)\,\{\texttt{Cm}\,;\ \texttt{return}\ X_k;\}$ in a class $\texttt{C}$ having n attributes $X_1,\ldots,X_n$, then, given a store σ s.t. $X\sigma = \texttt{new}\ \texttt{C}(o_1,\ldots,o_n)$, the evaluation of $X.\texttt{f}(e_1,\ldots,e_m)$ is performed first by evaluating the expressions e_j to the objects p_j, then, by evaluating the command $\texttt{Cm}$ with a store $\sigma\{x_1 \leftarrow p_1,\ldots,x_m \leftarrow p_m, X_1 \leftarrow o_1,\ldots,X_n \leftarrow o_n\}$ and, finally, by returning the object stored in X_k. The operational semantics of method call is described in figure 4.

$$\frac{\begin{array}{ll} \forall i\,\langle e_i,\sigma\rangle \downarrow \langle p_i,\sigma\rangle & \text{Class}\ \texttt{C}\,\{\ldots\texttt{var}\ \ X_j;\ldots\texttt{f}(x_1,\ldots,x_m)\,\{\texttt{Cm}\,;\ \texttt{return}\ X_k;\}\} \\ X\sigma = \texttt{new}\ \texttt{C}(o_1,\ldots,o_n) & \langle \texttt{Cm},\sigma\,\{X_1 \leftarrow o_1,\ldots,X_n \leftarrow o_n, x_1 \leftarrow p_1,\ldots,x_m \leftarrow p_m\}\rangle \downarrow \langle \sigma'\rangle \end{array}}{\langle X.\texttt{f}(e_1,\ldots,e_m),\sigma\rangle \downarrow \langle X_k\sigma',\sigma'\{X \leftarrow \texttt{new}\ \texttt{C}(X_1\sigma',\ldots,X_n\sigma')\}\rangle}$$

Figure 4: Operational semantics of a method call

Example 2 *Consider the following program together with the class of example 1:*

$\texttt{Class}$ main$\{\ \texttt{var}\ U;\ \texttt{var}\ V;\ \texttt{var}\ T;\ V := \texttt{new}\ \texttt{List}(U,\overline{\texttt{null}});\ \texttt{loop}\ \ T\,\{U := V.\texttt{setTail}(V);\}\ \}$

Given a store σ such that $U\sigma = o^U$ and $T\sigma = o^T$ we have:

$$\langle V := \texttt{new List}(U,\overline{\texttt{null}}),\sigma\rangle \downarrow \left\langle \sigma\left\{V \leftarrow \texttt{new List}(o^U,\overline{\texttt{null}})\right\}\right\rangle$$
$$\langle U := V.\texttt{setTail}(V),\sigma\rangle \downarrow \left\langle \sigma\left\{V \leftarrow \texttt{new List}(o^U,\texttt{new List}(o^U,\overline{\texttt{null}}),\overline{\texttt{null}})\right\}\right\rangle$$
$$\langle \texttt{loop}\ \ T\,\{U := V.\texttt{setTail}(V)\}\,,\sigma\rangle \downarrow \left\langle \sigma\left\{V \leftarrow \texttt{f}^{|o^T|}(\texttt{null})\right\}\right\rangle$$

where $\texttt{f}(x) = \texttt{new List}(o^U,x,\overline{\texttt{null}})$ *and* $\forall n \in \mathbb{N} - \{0\}$, $\texttt{f}^{n+1} = \texttt{f} \circ \texttt{f}^n$.

Remarks: The considered domain of computation is a set of terms without references and, consequently, it roughly approximates complex data structures such as cyclic data structure.

For example, given a store σ, a main program of the shape:

$$\begin{array}{l|l} X_1 := \texttt{new List}(X,\overline{\texttt{null}}); & X_0 := X_1.\texttt{setTail}(X_2); \\ X_2 := \texttt{new List}(Y,\overline{\texttt{null}}); & X_0 := X_2.\texttt{setTail}(X_1); \end{array}$$

computes a store σ' such that:

$$X_1\sigma' = \texttt{new List}(X\sigma,\texttt{new List}(Y\sigma,\overline{\texttt{null}}),\overline{\texttt{null}})$$
$$X_2\sigma' = \texttt{new List}(Y\sigma,\texttt{new List}(X\sigma,\texttt{new List}(Y\sigma,\overline{\texttt{null}}),\overline{\texttt{null}}),\overline{\texttt{null}})$$

However, this is not a serious drawback since the concern of this paper is to provide upper bounds to the number of object creations and such data are preserved by the representation of objects by terms.

3 Sup-interpretations and weights

3.1 Assignments

Let $\mathbb{R}^+$ be the set of positive real numbers.

DEFINITION 2.*[Class assignment] Given a class* $\texttt{C}$ *with* n *attributes, the assignment* $I_{\texttt{C}}$ *of the class* $\texttt{C}$ *is a mapping of domain dom*$(I_{\texttt{C}}) \subseteq \mathcal{F}_{\texttt{C}} \cup \{\texttt{C}\}$*, where* $\mathcal{F}_{\texttt{C}}$ *is the set of the methods of the class* $\texttt{C}$*. It assigns a function* $I_{\texttt{C}}(\texttt{f}) : (\mathbb{R}^+)^{m+1} \longmapsto \mathbb{R}^+$ *to each method symbol* $\texttt{f} \in$ *dom*$(I_{\texttt{C}})$ *of arity* m *and a function* $I_{\texttt{C}}(\texttt{C}) : (\mathbb{R}^+)^n \longmapsto \mathbb{R}^+$ *to the constructor* $\texttt{C}$.

DEFINITION 3.*[Program assignment] Given a program* $\texttt{p}$*, the assignment* I *of* $\texttt{p}$ *consists in the union of the assignments of each class* $\texttt{C}$ *of* $\texttt{Class}$*, i.e.* $I(b) =^{def} I_{\texttt{C}}(b)$ *whenever* $b \in$ *dom*$(I_{\texttt{C}})$.

DEFINITION 4.*[Canonical extension] A program assignment* I *is defined over an expression or method call* d *if each symbol of* $\mathcal{F} \cup \texttt{Class}$ *in* d *belongs to dom*(I)*. Suppose that the assignment* I *is defined over* d*, the partial assignment of* d *w.r.t.* I*, that we note* $I^*(d)$ *is the canonical extension of the assignment* I *defined as follows:*

1. *If* $\diamond$ *is an attribute or a parameter (in* $\mathcal{X} \cup \mathcal{P}$*), then* $I^*(\diamond) = \Box$*, with* $\Box$ *a new variable ranging over* $\mathbb{R}^+$*, s.t. the restriction of* I^* *to* $\mathcal{X} \cup \mathcal{P}$ *is an injective function.*
2. *If* $\texttt{C}$ *is a constructor of a class* $\texttt{C} \in \texttt{Class}$ *having* n *attributes and* $e_1, \ldots, e_n$ *are expressions then we have* $I^*(\texttt{new C}(e_1,\ldots,e_n)) = I(\texttt{C})(I^*(e_1),\ldots,I^*(e_n))$.

3. *If* $\mathtt{f} \in \mathcal{F}$ *is a method of arity* m *and* $e, e_1, \ldots, e_m$ *are expressions, then:*
   $$I^*(e.\mathtt{f}(e_1, \ldots, e_m)) = I(\mathtt{f})(I^*(e_1), \ldots, I^*(e_m), I^*(e))$$

Notice that the assignment $I^*(d)$ of an expression or method call d with m parameters $x_1, \ldots, x_m$ occurring in a class $\mathtt{C}$ having n attributes $X_1, \ldots, X_n$ denotes a function ϕ from $(\mathbb{R}^+)^{n+m} \to \mathbb{R}^+$ satisfying $\phi(I^*(X_1), \ldots, I^*(X_n), I^*(x_1), \ldots, I^*(x_m)) = I^*(d)$. Throughout the paper, we use the notation $I^*(e)(a_1, \ldots, a_n, b_1, \ldots, b_m)$ to denote $\phi(a_1, \ldots, a_n, b_1, \ldots, b_m)$.

DEFINITION 5. *Let **Max-Poly** $\{\mathbb{R}^+\}$ be the set of functions defined to be constant functions in $\mathbb{R}^+$, projections,* max, $+$, $\times$ *and closed by composition. Given a class with n attributes, an assignment I is said to be polynomial if for every symbol b of dom(I), $I(b)$ is a function of **Max-Poly** $\{\mathbb{R}^+\}$.*

DEFINITION 6. *The assignment of a constructor* $\mathtt{C}$ *of arity* n *is* additive *if:*

$$\begin{aligned} I(\mathtt{C})(\diamond_1, \ldots, \diamond_n) &= \sum_{i=1}^{n} \diamond_i + \alpha_{\mathtt{C}}, \text{ where } \alpha_{\mathtt{C}} \geq 1, && \text{if } n > 0 \\ I(\mathtt{C}) &= 0 && \text{if } n = 0 \end{aligned}$$

If the assignment of each constructor $\mathtt{C} \in \mathtt{Class}$ *is additive then the program assignment is additive.*

LEMMA 7. *Given a program* $\mathtt{p}$ *having an additive assignment* I*, there is a constant* α *such that for each attribute* X *and each store* σ*, the following inequalities are satisfied:*

$$|X\sigma| \leq I^*(X\sigma) \leq \alpha \times |X\sigma|$$

PROOF. Define $\alpha = \max_{\mathtt{C} \in \mathtt{Class}}(\alpha_{\mathtt{C}})$, where $\alpha_{\mathtt{C}}$ is taken to be the constant of definition 6, if $\mathtt{C}$ is of strictly positive arity, and $\alpha_{\mathtt{C}}$ is equal to the constant 0 otherwise. The inequalities follow directly by induction on the size of an object.

3.2 Sup-interpretations

DEFINITION 8. *Given a program* $\mathtt{p}$*, a sup-interpretation of* $\mathtt{p}$ *is an assignment* θ *of* $\mathtt{p}$ *which satisfies:*

1. *The assignment* θ *is weakly monotonic. i.e. for each symbol* $b \in \mathrm{dom}(\theta)$*, the function* $\theta(b)$ *satisfies* $\forall \diamond_1, \ldots, \diamond_n, \diamond'_1, \ldots, \diamond'_n \in \mathbb{R}^+, \diamond_i \geq \diamond'_i \Rightarrow \theta(b)(\ldots, \diamond_i, \ldots) \geq \theta(b)(\ldots, \diamond'_i, \ldots)$.
2. *For each object* $o \in \mathtt{Object}$*,* $\theta^*(o) \geq |o|$
3. *For each method* $\mathtt{f} \in \mathrm{dom}(\theta)$ *of arity* m*, for each* $o_1, \ldots, o_m \in \mathtt{Objects}$ *and for each store* σ*, if* $\langle X.\mathtt{f}(o_1, \ldots, o_m), \sigma \rangle \downarrow \langle o, \sigma' \rangle$ *then:*
   - $\theta(\mathtt{f})(\theta^*(o_1), \ldots, \theta^*(o_m), \theta^*(X\sigma)) \geq \theta^*(o)$
   - $\theta(\mathtt{f})(\theta^*(o_1), \ldots, \theta^*(o_m), \theta^*(X\sigma)) \geq \theta^*(X\sigma')$

A sup-interpretation is polynomial if it is a polynomial assignment.

Notice that the last condition on methods allows to bound both the sup-interpretation of the output $\theta^*(o)$ and the sup-interpretation of the side effect $\theta^*(X\sigma')$.

Example 3 *Consider the program of example 1. We claim that the partial assignment θ defined by $\theta(\texttt{null}) = 0$, $\theta(\texttt{setTail})(\Box_y, \Box) = \Box_y + \Box$ and $\theta(\texttt{List})(\Box_X, \Box_Y, \Box_W, \Box_Z) = \Box_X + \Box_Y + \Box_W + \Box_Z + 1$ is a sup-interpretation of this program. Indeed, the considered functions are monotonic. Since this assignment is additive, by lemma 7, we obtain that for each list l, $\theta(l) \geq |l|$. Finally, given a store σ such that $\langle X.\texttt{setTail}(o), \sigma\rangle \downarrow \langle v, \sigma'\rangle$ and $X\sigma = \texttt{new List}(h, t, o_W, o_Z)$, for some objects h, t o_W and o_Z, we have that $v = h$ and $\sigma' = \sigma\{X \leftarrow \texttt{new List}(h, o, o_W, o_Z)\}$. Consequently, we check that θ is a sup-interpretation:*

$$\begin{aligned}
\theta(\texttt{setTail})(\theta^*(o), \theta^*(\texttt{new List}(h, t, o_W, o_Z))) &\geq \theta^*(o) + \theta^*(\texttt{new List}(h, t, o_W, o_Z)) \\
&\geq \theta^*(o) + \theta^*(h) + \theta^*(t) + \theta^*(o_W) + \theta^*(o_Z) + 1 \\
&\geq \max(\theta^*(v), \theta^*(X\sigma'))
\end{aligned}$$

LEMMA 9. *Given a program* $\texttt{p}$ *having a sup-interpretation* θ *defined over* $X.\texttt{f}(e_1, \dots, e_n)$*, for each store* σ*, if* $\langle X.\texttt{f}(e_1, \dots, e_n), \sigma\rangle \downarrow \langle o, \sigma'\rangle$*, then* $\theta^*(X.\texttt{f}(e_1, \dots, e_n)\sigma) \geq \max(\theta^*(o), \theta^*(X\sigma'))$.

PROOF. We show this lemma in two steps. First, we can show easily by structural induction on an expression e that, for each store σ, if $\langle e, \sigma\rangle \downarrow \langle o, \sigma\rangle$ then $\theta^*(e\sigma) = \theta^*(o)$. Second, suppose that $a = X.\texttt{f}(e_1, \dots, e_n)$, $\langle a, \sigma\rangle \downarrow \langle o, \sigma'\rangle$ and that, for each $i \in \{1, n\}$ $\langle e_i, \sigma\rangle \downarrow \langle o_i, \sigma\rangle$.

$$\begin{aligned}
\theta^*(a\sigma) &\geq \theta^*(X\sigma.\texttt{f}(e_1\sigma, \dots, e_n\sigma)) && \text{By definition of } \sigma \\
&\geq \theta(\texttt{f})(\theta^*(e_1\sigma), \dots, \theta^*(e_n\sigma), \theta^*(X\sigma)) && \text{By definition of } \theta \\
&\geq \theta(\texttt{f})(\theta^*(o_1), \dots, \theta^*(o_n), \theta^*(X\sigma)) && \text{By step 1} \\
&\geq \max(\theta^*(o), \theta^*(o')) && \text{By definition 8}
\end{aligned}$$

Example 4 *Consider the linked list class of example 1, a method call $V.\texttt{setTail}(U)$ and a store σ such that $U\sigma = \texttt{new List}(\texttt{null}, \texttt{new List}(\overline{\texttt{null}}), \overline{\texttt{null}})$ and $V\sigma = \texttt{new List}(h, t, \overline{\texttt{null}})$, for some objects h and t. The method call $V.\texttt{setTail}(U)$ updates the tail of the object contained in V to the object contained in U and then returns the head of the object contained in V. Consequently, we obtain that:*

$$\langle V.\texttt{setTail}(U), \sigma\rangle \downarrow \langle h, \sigma\{V \leftarrow \texttt{new List}(h, \texttt{new List}(\texttt{null}, \texttt{new List}(\overline{\texttt{null}}), \overline{\texttt{null}}), \overline{\texttt{null}})\}\rangle$$

Taking the sup-interpretation θ defined in example 3, we check that:

$$\begin{aligned}
\theta^*(V.\texttt{setTail}(U)\sigma) &\geq \theta(\texttt{setTail})(\theta^*(U\sigma), \theta^*(V\sigma)) \\
&\geq \theta^*(\texttt{new List}(h, t, \overline{\texttt{null}})) + \theta^*(\texttt{new List}(\texttt{null}, \texttt{new List}(\overline{\texttt{null}}), \overline{\texttt{null}})) \\
&\geq \theta^*(h) + \theta^*(t) + 3 \\
&\geq \max(\theta^*(h), \theta^*(h) + 3) \\
&\geq \max(\theta^*(h), \theta^*(\texttt{new List}(h, \texttt{new List}(\texttt{null}, \texttt{new List}(\overline{\texttt{null}}), \overline{\texttt{null}}), \overline{\texttt{null}})))
\end{aligned}$$

3.3 Weights

The notion of weight allows to control the size of the objects (and a fortiori the number of instantiated objects) during loop iterations. A weight is a partial mapping over commands.

DEFINITION 10.*[Context] A context* $\mathtt{C}[\bullet_1,\dots,\bullet_n]$ *is a special command defined by the following grammar:*

$$\begin{aligned}\mathtt{C}[\overline{\bullet}] ::= &\mathtt{skip} \mid \bullet_1 \mid \dots \mid \bullet_n \mid X := a \mid X := e \mid \mathtt{C}_1[\overline{\bullet}];\mathtt{C}_2[\overline{\bullet}] \mid \mathtt{loop}\ \ X\,\{\mathtt{C}_1[\overline{\bullet}]\}\\ &\mid \mathtt{if}(e)\mathtt{then}\{\mathtt{C}_1[\overline{\bullet}]\}\mathtt{else}\{\mathtt{C}_2[\overline{\bullet}]\} \mid \mathtt{while}\ \ e\,\{\mathtt{C}_1[\overline{\bullet}]\}\end{aligned}$$

Let $\mathtt{C}[\mathtt{Cm}_1,\dots,\mathtt{Cm}_n]$ *denote the substitution of each* $\bullet_i$ *by the command* $\mathtt{Cm}_i$ *in* $\mathtt{C}[\bullet_1,\dots,\bullet_n]$*. A one-hole context is a context having exactly one occurrence of each* $\bullet_i$*. One-hole contexts induce a partial ordering* $\sqsubseteq$ *(resp. strict partial ordering* $\sqsubset$*) over commands defined by* $\mathtt{Cm}_1 \sqsubseteq \mathtt{Cm}_2$ *(resp.* $\mathtt{Cm}_1 \sqsubset \mathtt{Cm}_2$*) if and only if there is a one-hole context* $\mathtt{C}[\bullet]$ *(resp. distinct from* $\bullet$*) such that* $\mathtt{Cm}_2 = \mathtt{C}[\mathtt{Cm}_1]$*.*

DEFINITION 11.*[Minimal, while and loop commands] A command* $\mathtt{Cm}$ *is:*
- a minimal command if there is no context of the shape $\mathtt{C}[\bullet_1,\bullet_2] = \mathtt{if}(e)\mathtt{then}\{\bullet_1\}\mathtt{else}\{\bullet_2\}$ *or* $\mathtt{C}[\bullet_1,\bullet_2] = \bullet_1;\bullet_2$ *such that* $\mathtt{Cm} = \mathtt{C}[\mathtt{Cm}_1,\mathtt{Cm}_2]$*, for some commands* $\mathtt{Cm}_1$ *and* $\mathtt{Cm}_2$*.*

- a while command if there are a one-hole context $\mathtt{C}[\bullet]$ *and a command* $\mathtt{Cm}_1 = \mathtt{while}\ \ e\,\{\mathtt{Cm}_2\}$ *such that* $\mathtt{Cm} = \mathtt{C}[\mathtt{Cm}_1]$*.*

- a loop command if $\mathtt{Cm}$ *is not a while command and there are a one-hole context* $\mathtt{C}[\bullet]$ *and a command* $\mathtt{Cm}_1 = \mathtt{loop}\ \ X\,\{\mathtt{Cm}_2\}$ *such that* $\mathtt{Cm} = \mathtt{C}[\mathtt{Cm}_1]$*.*

Example 5 *We illustrate the distinct notions introduced above by the following example:*

$$\begin{aligned}&\mathtt{Class\ main}\,\{\,\mathtt{var}X;\ \mathtt{var}Y;\ \mathtt{var}Z;\\ &\qquad\qquad \mathtt{Cm}_1 : X := Y;\mathtt{loop}\ X\ \{\mathtt{while}\ Y\,\{\ Y = Y.\mathtt{getTail}();\}\};\\ &\qquad\qquad \mathtt{Cm}_2 : \mathtt{loop}\ \ X\,\{\mathtt{Cm}_3 : Z := Z.\mathtt{getTail}();\};\}\end{aligned}$$

The command $\mathtt{Cm}_1$ *is a while command but neither a minimal command nor a loop command. The command* $\mathtt{Cm}_2$ *is a minimal and loop command. The command* $\mathtt{Cm}_3$ *is only a minimal command.*

DEFINITION 12.*[Weight] Given a program* $\mathtt{p}$ *having a main class with* n *attributes, the weight* ω *is a partial mapping which assigns to every minimal and loop command* $\mathtt{Cm}$*, a total function* $\omega_{\mathtt{Cm}}$ *from* $(\mathbb{R}^+)^{n+1}$ *to* $\mathbb{R}^+$ *which satisfies:*

1. $\omega_{\mathtt{Cm}}$ *is weakly monotonic* $\forall i,\ \square_i \geq \square'_i \Rightarrow \omega_{\mathtt{Cm}}(\dots,\square_i,\dots) \geq \omega_{\mathtt{Cm}}(\dots,\square'_i,\dots)$
2. $\omega_{\mathtt{Cm}}$ *has the subterm property* $\forall i,\ \forall \square_i \in \mathbb{R}^+\ \omega_{\mathtt{Cm}}(\dots,\square_i,\dots) \geq \square_i$

A weight ω *is polynomial if each* $\omega_{\mathtt{Cm}}$ *is a function of* ***Max-Poly*** $\{\mathbb{R}^+\}$*.*

Example 6 *The program of example 5 has three attributes and exactly one minimal and loop command* $\mathtt{Cm}_2$*. Consequently, the mapping* ω *defined by* $\omega_{\mathtt{Cm}_2}(\square,\square_X,\square_Y,\square_Z) = \square + \max(\square_X,\square_Y,\square_Z)$ *is a polynomial weight.*

4 Criteria to control resources

4.1 Brotherly criterion

The brotherly criterion gives constraints on weights and sup-interpretations in order to bound the size of the objects computed by the program by some polynomial in the size of the inputs.

DEFINITION 13. *A program having a main class with n attributes $X_1, \dots, X_n$ is **brotherly** if there are a total, polynomial and additive sup-interpretation θ and a polynomial weight ω such that:*

- *For every minimal and loop command* `Cm` *of the main class:*
  - *For every method call a of the shape X_j.*`f`$(e_1, \dots, e_m)$ *occurring in* `Cm`*:*

    $$\omega_{\texttt{Cm}}(\Box+1, \theta(X_1), \dots, \theta(X_n)) \geq \omega_{\texttt{Cm}}(\Box, \theta(X_1), \dots, \theta(X_{j-1}), \theta^*(a), \theta(X_{j+1}), \dots, \theta(X_n))$$

    where $\Box$ is a fresh variable.
  - *For every variable assignment $X_i := d \sqsubseteq$* `Cm`*:*

    $$\omega_{\texttt{Cm}}(\Box+1, \theta(X_1), \dots, \theta(X_n)) \geq \omega_{\texttt{Cm}}(\Box, \theta(X_1), \dots, \theta(X_{i-1}), \theta^*(d), \theta(X_{i+1}), \dots, \theta(X_n))$$

    where $\Box$ is a fresh variable.
- *For every minimal and while command* `Cm` *of the main class:*
  - *For every variable assignment $X_i := d \sqsubseteq$* `Cm`*, $\max(\theta(X_1), \dots, \theta(X_n)) \geq \theta^*(d)$*

Intuitively, the first condition on loop commands ensures that the size of the objects held by the attributes remains polynomially bounded. The fresh variable $\Box$ can be seen as a temporal factor which takes into account the number of iterations allowed in a loop. Such a number is polynomially bounded by the size of the objects held by the attributes in the store. The second condition on while commands ensures that a computation is non-size-increasing since we have no piece of information about the termination of while commands.

THEOREM 14. *Given a brotherly program* `p` *of main class* `Class main`$\{A$ `Cm`$\}$*, having n attributes $X_1, \dots, X_n$, there exists a polynomial P such that for any store σ and any command* `Cm`$_1 \sqsubseteq$ `Cm` *if $\langle \texttt{Cm}_1, \sigma\rangle \downarrow \langle \sigma' \rangle$ then $P(|X_1\sigma|, \dots, |X_n\sigma|) \geq \max_{i=1..n}(|X_i\sigma'|)$.*

PROOF. We can build the polynomial P by structural induction on commands.

Example 7 *Consider the following program*

$$\texttt{Class main}\,\{\texttt{Var}\ U;\ \texttt{Var}\ V;\ \texttt{Var}\ T;\ \texttt{loop}\ T\,\{U := V.\texttt{reverse()}\};U.\texttt{setTail}(T)\}$$

`Cm` $=$ `loop` $T\,\{U := V.$`reverse()`$\}$ *is the only minimal and loop command. Consequently, we have to find a polynomial weight ω and a polynomial and additive sup-interpretation θ such that:*

$$\omega_{\texttt{Cm}}(\Box+1, \theta(U), \theta(V), \theta(T)) \geq \omega_{\texttt{Cm}}(\Box, \theta(U), \theta(V.\texttt{reverse()}), \theta(T))$$
$$\omega_{\texttt{Cm}}(\Box+1, \theta(U), \theta(V), \theta(T)) \geq \omega_{\texttt{Cm}}(\Box, V.\texttt{reverse()}, \theta(V), \theta(T))$$

in order to check the brotherly criterion. We let the reader check that the assignment θ defined by $\theta(\texttt{reverse})(\Box) = \Box$ together with the assignment of example 3 defines a total (i.e. defined for every method symbol), polynomial and additive sup-interpretation.

Moreover, taking $\omega_{\texttt{Cm}}(\Box, \Box_U, \Box_V, \Box_T) = \Box + \Box_U + \Box_V + \Box_T$, we obtain that this program is brotherly by checking that the above inequalities are satisfied.

4.2 Heuristics for method sup-interpretation synthesis

The previous criterion is very powerful. However, before being applied, it requires to know the sup-interpretation of the methods. Consequently, an interesting issue is to give some criterion on a method of some class in order to build its sup-interpretation.

DEFINITION 15._[Method weight] The weight of a method_ `D` _having m parameters and belonging to a class_ `C` _having n attributes is a monotonic and subterm function $\omega_{\mathtt{D}}$ from $(\mathbb{R}^+)^{m+2}$ to $\mathbb{R}^+$.A weight $\omega_{\mathtt{D}}$ is polynomial if it belongs to **Max-Poly** $\{\mathbb{R}^+\}$._

DEFINITION 16. _Given a class_ `C` _with n attributes $X_1, \dots, X_n$, a method_ `D` _of_ `C` _of the shape $\mathtt{f}(x_1, \dots, x_m)\,\{\mathtt{Cm}\,;\ \mathtt{return}\ X;\}$ is **brotherly** if there is a polynomial and additive sup-interpretation θ s.t.:_

1. _If_ `Cm` _is a while command then for every variable assignment $X_i := d \sqsubseteq \mathtt{Cm}$, we have:_
$$\max(\theta(x_1), \dots, \theta(x_m), \theta(X_1), \dots, \theta(X_n)) \geq \theta^*(d)$$
2. _Else there is a polynomial method weight $\omega_{\mathtt{D}}$ such that:_
   - _For every method call $a = X_j.\mathtt{f}(e_1, \dots, e_m)$ occurring in_ `Cm`_:_
   $\omega_{\mathtt{D}}(\Box+1, \theta(x_1), \dots, \theta(x_m), \sum_{k=1}^{n} \theta(X_k)) \geq \omega_{\mathtt{D}}(\Box, \theta(x_1), \dots, \theta(x_m), \sum_{k \neq j, k=1}^{n} \theta(X_k) + \theta^*(a))$
   - _For every variable assignment $X_i := d \sqsubseteq \mathtt{Cm}$, we have:_
   $\omega_{\mathtt{D}}(\Box+1, \theta(x_1), \dots, \theta(x_m), \sum_{k=1}^{n} \theta(X_k)) \geq \omega_{\mathtt{D}}(\Box, \theta(x_1), \dots, \theta(x_m), \sum_{k \neq i, k=1}^{n} \theta(X_k) + \theta^*(d))$

   where $\Box$ is a fresh variable.

THEOREM 17. _Given a program_ `p`_, a class_ `C` _having n attributes $X_1, \dots, X_n$ and a sup-interpretation θ such that the method $\mathtt{D} = \mathtt{f}(x_1, \dots, x_m)\,\{\mathtt{Cm}\,;\ \mathtt{return}\ X_i;\}$ of_ `C` _is brotherly, we have:_

- _If_ `Cm` _is a while command then $\theta(\mathtt{f})(\Box_1, \dots, \Box_m, \Box) =_{def} \max(\Box_1, \dots, \Box_m, \Box)$ is a sup-interpretation of_ `f`_._
- _Else, if R is a polynomial upper bound on the number of variable assignments occurring during the execution of_ `Cm` _then $\theta(\mathtt{f})(\Box_1, \dots, \Box_m, \Box) =_{def} \omega_{\mathtt{D}}(R(\Box), \Box_1, \dots, \Box_m, \Box)$ is a sup-interpretation of_ `f`_._

PROOF. The proof is similar to the proof of theorem 14. The only distinction is that parameters can appear in the commands.

Remarks: Since a command `loop` $X\,\{\mathtt{Cm}\}$ cannot write in the attribute X, the polynomial R can be computed by static analysis. Consequently, if we manage to check the brotherly criterion for a given method then we obtain a sup-interpretation of the method.

Example 8 _Consider the method_ `setTail` _of example 1. The command $Y := y$ is not a while command. Consequently, we have to find a polynomial weight $\omega_{\mathtt{D}} : (\mathbb{R}^+)^3 \rightarrow \mathbb{R}^+$ satisfying:_

$$\omega_{\mathtt{D}}(\Box+1, \theta(y), \sum_{K \in \{X,Y,W,Z\}} \theta(K)) \geq \omega_{\mathtt{D}}(\Box, \theta(y), \sum_{K \in \{X,W,Z\}} \theta(K) + \theta(y))$$

This inequality is satisfied by taking $\omega_{\mathtt{D}}(\Box, \Box_y, \Box') = \Box \times \Box_y + \Box'$. We know that there is exactly one variable assignment in the execution of such a method (i.e. $R = 1$) and, by theorem 17, $\theta(\mathtt{setTail})(\Box_y, \Box) = 1 \times \Box_y + \Box = \Box_y + \Box$ is a sup-interpretation of `setTail`_._

5 Conclusion and perspectives

We have suggested a high level approach for analyzing the complexity of object oriented programs. This static analysis is performed using semantics interpretations and provides upper bounds on the number of object creations during the execution of a given program.

Consequently, this study is complementary to the works of [9, 17, 3] using abstract interpretations which guarantee that there is no buffer overflow in the memory locations of a given program. Our study allows to perform a resource analysis of a huge number of programs. Some improvements can obviously be performed in several directions: Currently, a while iteration cannot compute more than a maximum function. A more precise analysis of while iterations should be performed using the work of [19] on the termination of imperative while programs. The criterion for sup-interpretation synthesis has no sense when considering recursive (and a fortiori mutual recursive) methods (Since we have to previously know the sup-interpretation of the considered symbol). As a consequence, we have to develop a criterion in the general recursive case, even if side effects make such a study difficult.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing useless variables in cost analysis of Java bytecode. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 368–375. ACM New York, NY, USA, 2008.

[3] X. Allamigeon and C. Hymans. Static Analysis by Abstract Interpretation: Application to the Detection of Heap Overflows. *Journal in Computer Virology*, 4, 2007.

[4] R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, volume 3210 of *LNCS*, pages 265–279. Springer, 2004.

[5] R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. In *CONCUR*, volume 3170 of *LNCS*, pages 68–82. Springer, 2004.

[6] R.J. Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, New York, 2001.

[7] G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR 2006*, volume 4246 of *LNAI*, pages 90–104. Springer, 2006.

[8] G. Bonfante, J.Y. Marion, and J.Y. Moyen. Quasi-interpretations, a way to control resources. *Theoretical Computer Science*. Accepted.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77*, pages 238–252, 1977.

[10] S. Dal-Zilio and R. Gascon. Resource bound certification for a tail-recursive virtual machine. In *APLAS 2005*, volume 3780 of *LNCS*, pages 247–263. Springer, 2005.

[11] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. LNCS, pages 41–82. Springer, 1999.

[12] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[13] L. Kristiansen and N.D. Jones. The flow of data and the complexity of algorithms. *New Computational Paradigms*, 3526:263–274, 2006.

[14] J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS 2006*, volume 3945 of *LNCS*, pages 163–176, 2006.

[15] J.-Y. Marion and R. Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Transactions on Computational Logic (TOCL)*, 2008. Accepted.

[16] J.Y. Marion and R. Péchoux. A Characterization of NCk by First Order Functional Programs. In *TAMC*, volume 4978 of *LNCS*, pages 136–147. Springer, 2008.

[17] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63, Ottawa, Ontario, Canada, June 2006. ACM Press. `http://www.di.ens.fr/~mine/publi/article-mine-lctes06.pdf`.

[18] K.H. Niggl and H. Wunderlich. Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs. *SIAM Journal on Computing*, 35:1122, 2006.

[19] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144. ACM, 2005.