

# Memory and Time Consumption of Java Bytecode Programs

Laurent Hubert

February 2006



## Introduction

Mobile code is spreading in our daily life and an important part of this code is in Java bytecode. This is the main language for applications for mobile phones, PDAs (*Personal Digital Assistant*) and *smartphones*. For the consumer, i.e the mobile device user, to trust the software he wants to download, the main technology used is cryptographic signatures. It allows the consumer to have a proof of the provider of the software, but not to have a proof that the software actually respects any security policy.

Trust in mobile code is a major issue and so the european Mobius project [13] has been created. It involves both industrial and academic partners and it has the purpose of developing the technologies to establish trust and security in mobile code. The target architecture chosen by Mobius is Java.

The first part of my internship consists in the conception of a static analysis to detect bounds on memory and time consumption for the Java bytecode. To compute those bounds, we need to bound the values of, for example, loop counters. A well know method to detect such invariant properties on program is *Abstract Interpretation*, presented in section 1.

Such analyses can be expansive. If developers uses those analyses to prove their software, users have generally to trust the developers. Another solution is to let users prove themselves the software but for several reasons, given in section 2, it can be difficult. *Proof-Carrying Code* (PCC for short) is a solution proposed by G. Necula in [14], retained by the Mobius project, and presented in section 2.

Finally, section 3 presents some results that have already been obtained by other related analyses.

## 1 Abstract interpretation

Abstract Interpretation has been proposed by P. Cousot and R. Cousot in [6]. The main idea is to “execute” the program on abstract values instead of concrete ones.

### 1.1 Theoretical basis

#### 1.1.1 Semantic of programs

The purpose of abstract interpretation is to approximate the semantics of a program, we therefore need to defined semantics. In abstract interpretation, the semantics considered are fixpoint-based semantics, modeled in a partially ordered sets. The partial order permit to compare the precisions of different semantics. In this setting, a monotonic semantic operator  $F_P$  is associated to each program  $P$ .  $F_P$  is defined on a semantic domain describing the

properties we are interested in. The program semantics is then defined as the least fixpoint of the  $F_P$  function:

$$\llbracket P \rrbracket = \text{lf}p(F_P)$$

There are some constraints on the semantic domains and they are given in section 1.1.2.

An example of a fixpoint-based semantic is the set of accessible states of a program.

**Example 1 (Set of Accessible States)** *The semantic of a program  $P$  can be expressed as the set of accessible states  $\llbracket P \rrbracket \in 2^{\text{State}}$  from an initial set of states  $S_0$  and a transition relation  $\rightarrow_P \subseteq \text{State} \times \text{State}$ .*

$$\llbracket P \rrbracket = \{s \mid \exists s_0 \in S_0, s_0 \rightarrow_P^* s\} = \bigcup_{n \in \mathbf{N}} \text{post}^n(S_0)$$

with  $\text{post}(S) = \{s \mid \exists s' \in S, s' \rightarrow_P s\}$ . Thus, we can defined  $\llbracket P \rrbracket$  as :

$$\llbracket P \rrbracket = \text{lf}p(F_P) \text{ with } F_P(S) = S \cup \text{post}(S)$$

This semantics is the one that will be used for the memory and time consumption computation.

### 1.1.2 Fixpoint

Tarski theorem tells us that there exists a least fixpoint for  $F_P$  under the hypothesis that the computation is done in a complete lattice.

**Theorem 1 (Tarski theorem)** *Let  $L = (D, \subseteq, \cap, \cup, \perp, \top)$  be a complete lattice and  $F_P : D \rightarrow D$  a monotonic function, then  $F_P$  has a least fixpoint:*

$$\text{lf}p(F_P) = \bigcap \{x \in D \mid F_P(x) \subseteq x\}$$

Tarski theorem proves the existence of a least fixpoint but Kleen theorem gives a more constructive definition of it.

**Theorem 2 (Kleen theorem)** *Let  $L = (D, \subseteq, \cap, \cup, \perp, \top)$  be a complete lattice and  $F_P : D \rightarrow D$  a continuous function, then*

$$\begin{aligned} \text{lf}p(F_P) &= \bigcup_{n \geq 0} F_P^n(\perp) \text{ and} \\ \text{gfp}(F_P) &= \bigcap_{n \geq 0} F_P^n(\top) \end{aligned}$$

with  $F_P^0(\vec{x}) = \vec{x}$  and  $F_P^{n+1} = F_P(F_P^n(\vec{x}))$ .

Thus, in order to define fixpoint semantics, we need to define a continuous function as semantic operator and a complete lattice as semantic domain.

## 1.2 Approximations

The computation of the semantics corresponds to solving a fixpoint equation in the semantic domain  $D$ , which is uncomputable in the general case (because of possibly infinite Kleen sequences). Even the computation terminates, it can be too expensive because the concrete semantic domain is too complicated to manipulate.

Two levels of approximations have been introduced by P. Cousot and R. Cousot in [6, 5] respectively presented in sections 1.2.1 and 1.2.2.

### 1.2.1 Static approximation

Concrete values can be too complicated or expensive to manipulate and some information given by the concrete values may be useless with respect to the studied properties. As a solution to this problem, P. Cousot and R. Cousot proposed in [6] to replace the concrete semantic domain by an abstract semantic domain linked with the concrete one by a Galois connection.

**Definition 1 (Galois connection)** *Let  $D(\subseteq)$  and  $D^\sharp(\sqsubseteq)$  be two partially ordered sets and  $\alpha$  and  $\gamma$  two functions such as  $\alpha : D \rightarrow D^\sharp$  and  $\gamma : D^\sharp \rightarrow D$ . Then,  $(D, \alpha, \gamma, D^\sharp)$  is a Galois connection if :*

$$\forall x \in D, \forall y \in D^\sharp, \alpha(x) \sqsubseteq y \Leftrightarrow x \subseteq \gamma(y)$$

$\alpha$  is called the abstraction function and  $\gamma$  the concretization function. An abstract value corresponds to a possibly infinite set of concrete values.

**Definition 2** *Let  $(D, \alpha, \gamma, D^\sharp)$  be a Galois connection and  $f : D^n \rightarrow D$  a monotonic function.*

$$\alpha(f) : \begin{array}{l} D^{\sharp n} \rightarrow D^\sharp \\ (x_1, \dots, x_n) \mapsto \alpha(f(\gamma(x_1), \dots, \gamma(x_n))) \end{array}$$

To the semantic operator  $F_P$ , we now associate the abstract operator  $F_P^\sharp : D^\sharp \rightarrow D^\sharp$  with  $F_P^\sharp = \alpha(F_P)$ . Let  $\llbracket P \rrbracket^\sharp$  be the least fixpoint of  $F_P^\sharp$ ,  $\llbracket P \rrbracket^\sharp = \text{lfp}(F_P^\sharp)$ .

The Galois connection between the abstract and the concrete domains ensures that computation of the least fixpoint in the abstract domain is a correct over-approximation of the abstraction of the least fixpoint computed with concrete values, that is to say :

$$\alpha(\llbracket P \rrbracket) \sqsubseteq \llbracket P \rrbracket^\sharp$$

### 1.2.2 Dynamic approximation

Static approximation makes the computation easier but the height of abstract domains can still be infinite. As a consequence, the computation might not be convergent. To solve this problem, P. and R. Cousot introduced in [5] a *widening* operator  $\nabla$  which allows to compute the fixpoint in a finite number of steps — and faster than without it if the number of step was already finite.

**Definition 3 (Widening operator)** *Let  $(D, \alpha, \gamma, D^\sharp)$  be a Galois connection.  $\nabla$  is a widening operator if:*

- $\forall (x_1, x_2) \in D^{\sharp^2}, (x_1 \cup x_2) \sqsubseteq (x_1 \nabla x_2)$  and
- *every infinite sequence defined by  $s_0 = \perp$  and  $s_n = s_{n-1} \nabla x_n$  is not strictly increasing (for  $x_n$  arbitrary abstract values).*

### 1.3 Domains for integer representation

In order to compute the memory and time consumption we need to infer invariant properties on numeric variables (as loop counters for example).

Abstract domains for integer representation have been studied for a while and they offer different levels of expressivity and complexity. This section exhibits some of them, presented in [12].

#### 1.3.1 Intervals

With intervals, a concrete set of values is represented by the smallest interval containing all the concrete values.

$$I = \{\perp^\sharp\} \cup \{(x_1, x_2) \mid (x_1, x_2) \in (\mathbf{N} \cup \{-\infty\}) \times (\mathbf{N} \cup \{\infty\}) \wedge x_1 \leq x_2\}$$

**Definition 4 (abstraction function)**

$$\begin{aligned} \alpha : 2^N &\rightarrow N^2 \\ \emptyset &\mapsto \perp \\ C &\mapsto (\min(C), \max(C)) \end{aligned}$$

**Definition 5 (concretization function)**

$$\begin{aligned} \gamma : N^2 &\rightarrow 2^N \\ (x_1, x_2) &\mapsto \{n \mid x_1 \leq n \leq x_2\} \end{aligned}$$

Union and intersection of intervals are intervals and are defined as follows.

**Definition 6 (Union)**

$$A \cup B = \begin{cases} \perp^\# & \text{if } A = \perp^\# \text{ and } B = \perp^\# \\ (a, a') & \text{if } A = (a, a') \text{ or } B = (a, a') \\ (\min(\{a; b\}), \max(\{a'; b'\})) & \text{if } A = (a, a') \text{ and } B = (b, b') \end{cases}$$

**Definition 7 (Intersection)**

$$A \cap B = \begin{cases} (\max(\{a; b\}), \min(\{a'; b'\})) & \text{if } A = (a, a') \text{ and } B = (b, b') \\ \perp^\# & \text{otherwise} \end{cases}$$

The widening operator can be defined as follows.

**Definition 8 (widening operator)**

$$\begin{aligned} \perp^\# \nabla (a, a') &= (a, a') \\ (a, a') \nabla (b, b') &= (c, d) \quad \text{with } c = -\infty \text{ if } b < a, a \text{ otherwise, and} \\ &\quad d = \infty \text{ if } b' > b, a' \text{ otherwise} \end{aligned}$$

There are other ways to define it and, in particular, by extending a bound to an other bound (e.g. to a bound given in the declaration of the variable if there is one, or the maximum value of a counter in a loop, etc.).

**1.3.2 Zones and Octagons**

**Zone** It can represent bounds on variables ( $a_i \leq v_i \leq b_i$ ) and on differences of variables ( $a_{i,j} \leq v_i - v_j \leq b_{i,j}$ ). A zone is represented by a graph and used algorithms are usual graph algorithms. The time complexity of those operations is usually  $O(N^3)$ .

**Octagon** This lattice extends the previous one by giving the possibility to represent every constraint of the form  $av_i + bv_j \leq c$  with  $a$  and  $b$  in  $\{-1; 0; 1\}$ . Algorithms are also based on the ones of graphs and the time complexity is still  $O(N^3)$ . The space complexity is  $O(N^2)$ .

**1.3.3 Sparse octagons**

This lattice has been studied in [11]. It is based on the idea that relations between some variables are more important than others. The set of observed variables can be partitioned to reflect the importance of the different relations. Those partitions can share variables to be able to deduce constraints between variables that are not in the same partition.

Compared to octagons, the time complexity is reduced if some variables are shared between the different subsets. If all the variables are shared, it is better to take an octagon; if no variables are shared, it is better to take an octagon per subset.

### 1.3.4 Convex Polyhedra

Convex polyhedra can represent every conjunction of linear constraints on the variables of the program:  $a_1v_1 + \dots + a_Nv_N \leq c$  where  $a_1, \dots, a_n$  are rational constants,  $v_1, \dots, v_n$  are the variables of the program and  $c$  is a real constant.

There are two ways to represent a polyhedron: with linear inequations and with generators. The two representations are dual and the time complexity of the conversion is exponential (depending on the number of variables).

## 2 *Proof-Carrying Code*

Proving some safety properties on code can be difficult. It can be sometimes automated by some techniques, as abstract interpretation described in section 1, but the analysis can be expensive in space and time. It may also happen that some properties still need to be proven by hand or by other mechanisms.

With mobile code, the consumer is generally different from the producer. If the producer proves the property before distributing the code but nothing except the code is distributed, consumers have to trust the producer.

If the producer does not prove the property, each consumer must prove the properties and it can be more difficult for them for several reasons.

Firstly, consumers are generally not supposed to know how a functionality is performed by the code, which makes a hand proof impossible.

Secondly, the most precise analyses are also the most expensive and mobile code is often used by consumers with limited resources, like mobile phones. Consumers can therefore not use precise analyses.

Finally, they have generally only native machine code, which does not contain as much information as the high-level languages that have been used by the producer.

The solution proposed by G. Necula is *Proof-Carrying Code*, described in [14].

### 2.1 Concepts of *Proof-Carrying Code*

The idea is to let the producer prove his code respects some properties and provide a proof with the code. When consumers want to execute the code for the first time, they check the provided proof on the provided code. To achieve this, a common safety policy must be defined and accessible by producers and consumers. The PCC architecture can then be divided into three different stages.

**Certification** During the *certification* process, the producer compiles and proves that the (native) code that will be distributed respects the safety policy. To do so, the producer can use complex analyses, with possibly heuristics, user interactions, etc. A proof of successful verification, a *certificate*, is then produced and added to the code to form the PCC binary.

**Validation** In the *validation* stage, the consumer validates the proof contained in the PCC binary. It should be quick and driven by straightforward algorithms. In addition to the soundness of the safety policy, the implementation of the validation algorithm is the only code consumers have to trust.

**Execution** Finally, consumers can execute the code confidently, as many times as they want.

The PCC architecture is represented by the figure 1.

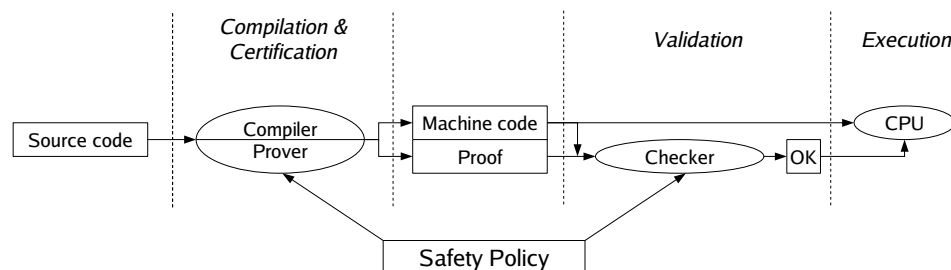


Figure 1: Overview of the *Proof Carrying Code* architecture

## 2.2 *Abstraction-Carrying Code*

PCC corresponds to principles but the choice of the enabling technologies was left. The technology used by G. Necula in [14] was based on first order logic but there also exist analyses and proofs based on other kinds of logics and on type checking. Each one comes with different sizes of certificate and different costs for the certification and validation stages. Depending on the requirement of the architecture, one or another solution was chosen.

A successful certificate infrastructure should therefore allow enabling technologies that correspond to all those requirements. *Abstract-Carrying Code* [1] (ACC for short) claims to be a such infrastructure.

### 2.2.1 Principles

ACC relies on abstract interpretation, presented in section 1. The overall architecture is globally the same as for PCC but more precisely described

(specially in [2]). It has been designed to be independent it has been applied to Ciao Prolog [3].

Depending on the safety properties that must be proved to be respected, an abstract domain  $D^\#(\sqsubseteq)$  is chosen. The safety policy  $\mathcal{I}^\#$  can then be expressed with substitutions over this domain. Abstract domains used can be represent time costs, space costs, types, etc.

An abstraction of the program  $\llbracket P \rrbracket^\#$  is computed by abstract interpretation by the fixpoint based analyzer. It is then easy to check that the calculated abstraction respects the safety policy by doing inclusion tests, i.e  $\llbracket P \rrbracket^\# \sqsubseteq \mathcal{I}^\#$ .

The checker algorithm is a simplified analyzer in that it does not iterate. The consumer has to run the checker on the received program with the received abstraction. If the abstraction is not modified by the checker, then it is a fixpoint and, thus, a correct abstraction of the program. The abstraction plays the role of the certificate.

The consumer can then prove that the abstraction respects the safety policy.

To express the safety policy in an easier way, a *verification condition generator* is used to extract, from the safety policy and the abstraction, the inclusion tests that must be verified.

The ACC architecture is represented by figure 2.

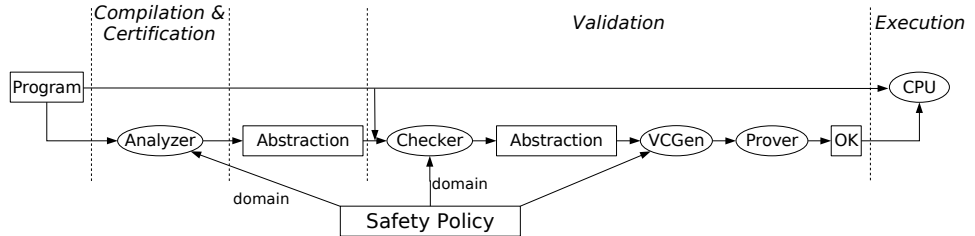


Figure 2: Overview of the *Abstraction-Carrying Code* architecture

### 3 Results related to Java bytecode Analyses

The target architecture proposed by Mobius is Java and the method is PCC. As the code transmitted is bytecode and PCC, as explained in section 2, refers to transmitted code (and not to source code) there is a need for analyses for Java bytecode.

#### 3.1 Other Abstract Interpretation Analyses

Abstract interpretation has been studied for almost 30 years and a great number of abstract interpretation-based analyses exist.

First, abstract interpretation has been applied by P. Cousot and R. Cousot to *structured* imperative languages in [6].

It has also been applied with a lot of success to logic programs in order to debug, verify and optimize logic programs. Examples of those analyses have been implemented in CiaoPP and some of them are presented in [9].

Another interesting example is the work on memory consumption of first-order functional programs by M. Hofmann and S. Jost in [10]. This analysis is based on a type system — but, as explained in [4], types are abstract interpretations — and can give the amount of heap memory needed by a function as well as an under-estimation of the size of the free-list after a successful execution of a function.

However, there are fewer analyses done on assembler or bytecode.

### 3.2 Other Java bytecode Analyses

Fausto Spoto has worked on bytecode analysis, mainly on class analysis [15] with Thomas Jensen and Frédéric Besson, and information flow analysis [8] with Samir Genaim. His work may be useful to extend an intra-procedural analysis to an inter-procedural analysis.

David Pichardie et al. have also developed a certified analysis for memory consumption [7] of Java bytecode. The analysis is proven to be correct but is very imprecise: each time a memory allocation is found in a loop (which can be a recursion), the result is *unbounded*.

## Conclusion

*Abstract interpretation* is a very powerful framework and is well adapted to analyses on bounds of variables, which is needed for memory consumption computation.

*Proof-Carrying Code* is a method which allows a consumer, or mobile code user, to have a proof that the software he wants to execute is safe, in the sense that it respects a safety policy defined by the consumer (or at least the consumer agrees with it).

Through *Abstraction-Carrying Code*, abstract interpretation and PCC are used together to develop a framework to prove properties and give a checkable proof of those properties along with the code.

As seen in section 3, there is still a lack for analyses suitable for ACC and Java bytecode.

I have already developed an analyzer for Java bytecode for memory consumption in Irisa. However, firstly, the analysis rely on polyhedra and is therefore expensive. Secondly, the increase in precision in comparison with a simple loop detection is unknown on real programs: there is no automatic diagnostic and each time but one I check by myself, a loop detection would have given the same result. On the other hand, it is easy to design

an example that shows better results with this analysis than with a simple loop detection. Thirdly, the result is meaningless without an inter-procedural analysis and until now there is not.

I therefore plan to develop an analysis for memory and time consumption for Java bytecode and to implement an ACC infrastructure for those programs.

## References

- [1] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004.
- [2] Elvira Albert, Germán Puebla, and Manuel Hermenegildo. Abstraction-carrying code: a model for mobile code safety. Technical report, Technical University of Madrid, 2005.
- [3] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). Technical report, School of Computer Science (UPM), 2004. Available at <http://clip.dia.fi.upm.es/Software/Ciao/>.
- [4] Patrick Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, january 1997. ACM Press, New York, NY.
- [5] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In Dunod, editor, *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, Paris, France, 1976.
- [6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.
- [7] David Cachera et al. Certified memory usage analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, number 3582 in Lecture Notes in Computer Science, pages 91–106. Springer-Verlag, 2005.
- [8] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, January 2005. Springer-Verlag.
- [9] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Science of Computer Programming*, pages 115–140, October 2005.

- [10] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, volume 38-1 of *ACM SIGPLAN Notices*, pages 185–197, New York, January 2003. ACM Press.
- [11] Hugo Métivier. Représentation d’ensembles de valeurs numériques en vérification de programme. Master’s thesis, Ifsic, Rennes, France, 2005.
- [12] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Normale Supérieure, december 2004.
- [13] Mobius project: <http://mobius.inria.fr>.
- [14] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
- [15] F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):578–630, September 2003.