Precise Analysis of Memory Consumption using Program Logics*

Gilles Barthe
INRIA Sophia–Antipolis, France
Gilles.Barthe@sophia.inria.fr

Mariela Pavlova
INRIA Sophia–Antipolis, France
Mariela.Pavlova@sophia.inria.fr

Gerardo Schneider
Department of Informatics, University of Oslo, Norway
Gerardo.Schneider@ifi.uio.no

Abstract

Memory consumption policies provide a means to control resource usage on constrained devices, and play an important role in ensuring the overall quality of software systems, and in particular resistance against resource exhaustion attacks. Such memory consumption policies have been previously enforced through static analysis, which yield automatic bounds at the cost of precision, or run-time analysis, which incur an overhead that is not acceptable for constrained devices.

In this paper, we study the use of logical methods to specify and statically verify precise memory consumption policies for Java bytecode programs. First, we demonstrate how the Bytecode Specification Language (a variant of the Java Modelling Language tailored to bytecode) can be used to specify precise memory consumption policies for (sequential) Java applets, and how verification tools can be used to enforce such memory consumption policies. Second, we consider the issue of inferring some of the annotations required to express the memory consumption policy, and report on an inference algorithm.

Our broad conclusion is that logical methods provide a suitable means to specify and verify expressive memory consumption policies, with an acceptable overhead.

1. Introduction

Trusted personal devices (TPDs for short) such as smart cards, mobile phones, and PDAs commonly rely on execution platforms such as the Java Virtual Machine and the Common Language Runtime. Such platforms feature security mechanisms such as bytecode verification and stack inspection that guarantee basic safety properties of downloaded applications. However, current security architectures for TPDs do not provide any mechanism to control resource usage by downloaded applications, despite TPDs being subject to stringent resource constraints. Therefore, TPDs are particularly vulnerable to denial-of-service attacks, since executing a downloaded application may potentially lead to resource exhaustion.

Several approaches have been suggested to date to enforce memory consumption policies for programs, see Section 2. All approaches are automatic, but none of them is ideally suited for TPDs, either for their lack of precision, or for the runtime penalty they impose on programs.

The objective of this work is to explore an alternative approach that favours precision of the analysis at the cost of automation, but without runtime penalty. The approach is based on program logics and allows users to perform a precise analysis of resource consumption for Java bytecode programs (for the clarity of the explanations all examples in the introduction deal with source code). In order to illustrate the principles of our approach, let us consider the following program:

```
public void m (A a) {
  if (a == null)
    { a = new A();  }
  a.b = new B(); }
```

In order to model the memory consumption of this program, we introduce a *ghost* (or, *model*) variable `Mem` that accounts for memory consumption; more precisely, the value of `Mem` at any given program point is meant to provide an upper bound to the amount of memory consumed so far. To keep track of the memory consumption, we perform immediately after every bytecode that allocates memory an increment of `Mem` by the amount of memory consumed by the allocation. Thus, if the pro-

* Partially supported by the RNTL French project CASTLES, by the IST Project Inspired, and by the ACI Sécurité SPOPS.

grammer specifies that ka and kb is the memory consumed by the allocation of an instance of class A and B respectively, the program must be annotated as:

```
public void m (A a) {
 if (a == null)
   {a = new A(); //set Mem = Mem + ka;}
 a.b = new B(); //set Mem = Mem + kb; }
```

Such annotations allow to compute at run-time the memory consumed by the program. However, we are interested in static prediction of memory consumption, and resort to pre- and postconditions to this end.

Even for a simple example as above, one can express the specification at different levels of granularity. For example, fixing the amount of memory that the program may use, Max, one can specify that the method will use at most ka + kb memory units and will not overpass the authorised limit Max, with the following specification:

$$\begin{array}{ll} \texttt{//@ requires} & \texttt{Mem} + \texttt{ka} + \texttt{kb} \leq \texttt{Max} \\ \texttt{//@ ensures} & \texttt{Mem} \leq \backslash\text{old}(\texttt{Mem}) + \texttt{ka} + \texttt{kb} \end{array}$$

```
public void m (A a) { ... }
```

Or try to be more precise and relate memory consumption to inputs with the following specification:

$$\begin{array}{l} \texttt{//@ requires a==null} \Rightarrow \texttt{Mem} + \texttt{ka} + \texttt{kb} \leq \texttt{Max} \\ \qquad\qquad \wedge\ \texttt{!(a==null)} \Rightarrow \texttt{Mem} + \texttt{kb} \leq \texttt{Max} \\ \texttt{//@ ensures} \\ \quad \backslash\text{old}\texttt{(a)==null} \Rightarrow \texttt{Mem} \leq \backslash\text{old}(\texttt{Mem}) + \texttt{ka} + \texttt{kb} \\ \quad \wedge\ \texttt{!}(\backslash\text{old}\texttt{(a)==null}) \Rightarrow \texttt{Mem} \leq \backslash\text{old}(\texttt{Mem}) + \texttt{kb} \end{array}$$

```
public void m (A a) { ... }
```

More complex specifications are also possible: one can take into account whether the program will throw an exception or not by using (possibly several) exceptional postconditions stating that $\texttt{k}_\texttt{E}$ memory units are allocated in case the method exits on exception E.

The main characteristics of our approach are:

- *Precision:* our analysis allows to specify and enforce precise memory consumption policies, including those that take into account the results of branching statements or the values of parameters in method calls. Being based on program logics, which are very versatile, the precision of our analysis can be further improved by using it in combination with other analysis, such as control flow analysis and exception analysis;
- *Correctness:* our analysis exploits existing program logics which are (usually) already known to be sound. In fact, it is immediate to derive the soundness of our analysis from the soundness of the program logic, provided ghost annotations that update memory consumption variables are consistent with an instrumented semantics that extends the language operational semantics with a suitable cost model that reflects resource usage;
- *Language coverage:* our analysis relies on the existence of a verification condition generator for the programming language at hand, and is therefore scalable to complex programming features. In the course of the paper, we shall illustrate applications of our approach to programs featuring recursive methods, method overriding and exceptions;
- *Usability:* our approach can be put to practise immediately using existing verification tools for program logics. We have applied it to annotated Java bytecode programs using a verification environment developed in [6]. It is also possible to use our approach on JML annotated Java source code [7], and more generally on programs that are written in a language for which appropriate support for contract-based reasoning exists;
- *Annotation and proof generation:* in contrast to other techniques discussed above, our approach requires user interaction, both for specifying the program and for proving that it meets its specification. In order to reduce the burden of the user, we have developed heuristics that infer automatically part of the annotations, and use automatic procedures to help discharge many proof obligations automatically.

Furthermore, our analysis may be used to guarantee that no memory allocation is performed in undesirable states of the application, namely after initialisation or during a transaction in a Java Card.

On the negative side, our method does not deal with garbage collection nor arrays (see Section 6).

Contents The paper is organised as follows: the next Section discusses related work while Section 3 provides a brief introduction to Java bytecode programs and to the modelling language and weakest precondition calculus used to specify and verify such programs. Section 4 describes in some detail program logics can be used to specify and verify precise memory consumption policies. Section 5 is devoted to inference of annotations. We conclude in Section 6 with directions for future work.

2. Related Work

- *Static analysis and abstract interpretations:* in such an approach, one performs an abstract execution

of an approximation of the program. The approximation is chosen to be coarse enough to be computable, as a result of which it yields automatically bounds on memory consumption, but at the cost of precision. Such methods are not very accurate for recursive methods and loops, and often fail to provide bounds for programs that contain dynamic object creation within a loop or a recursive method.

In a series of papers including [3], M. Hofmann and co-authors have investigated the use of type systems for estimating memory consumption. For example, Hofmann and Jost [15] propose an automatic heap space usage static analysis for first-order functional programs. The analysis determines both the amount of free cells necessary before execution as well as a safe (under)-estimate of the size of a *free-list* after successful execution of a function. These numbers are obtained as solutions to a set of linear programming (LP) constraints derived from the program text. Automatic inference is obtained by using standard polynomial-time algorithms for solving LP constraints. The type systems are then used within a proof-carrying code architecture for enforcing resource control policies.

In a similar line of work, R. Amadio and coworkers have been studying the resource bounds problem using type, size and termination verifications for programs that execute in a simple stack machine. Their early work [24] defines an analysis at the level of the bytecode for a sequential language, while their later work [1] extends their result to cooperative threads, but works at the level of source code.

Another related work is [2], which introduces a first-order linearly typed assembly language that allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints.

A different technique, based on the computation of linear invariants which relate program variables to memory consumption, is presented in [5]. In a nutshell, the amount of memory consumed by a program is the number of integer points satisfying the invariant; such number is a polynomial where the unknowns are method input parameters. Many experiments are presented showing the precision of the technique.

Building up on [24] (no mechanical proof nor implementation is provided in such work), the third author and co-workers [8] have also developed a certified static analysis for a Java-like bytecode language. Their analysis uses a constraint-based algorithm to check the existence of `new` instructions inside intra- and inter-procedural loops; the analyser has been automatically extracted from its Coq's correctness proof.

- *Run-time monitoring:* here the program also comes equipped with a specification of its memory consumption, but the verification is performed at runtime, and interrupted if the memory consumption policy is violated. Such an approach is both precise and automatic, but incurs a runtime overhead which makes it unsuitable for TPDs.

  L.-A. Fredlund [12] implements a runtime monitor that controls the execution of a Java card applet. In order to guarantee the desired memory allocation property, a call to a monitor method is added before a `new` instruction; the monitor method has as parameter the size of the allocation request and it halts the execution of the applet if a predefined allocation bound is exceeded.

  A method for analysing, monitoring and controlling dynamic memory allocation using pointer and scope analysis is presented in [14]. A Java program is automatically instrumented using the information given by the pointer and escape analysis and a region-based memory manager is synthesised, which dynamically maps "creation sites to the region stack at runtime via a registering mechanism".

A hybrid (i.e., static and dynamic) resource bound checker for an imperative language designed is presented in [9]. The verifier is based on a variant of Dijkstra's weakest precondition calculus using "generalised predicates", which keeps track of the resource units available. Besides adding loop invariants, pre- and post-conditions, the programmer must insert "acquires" annotations to reserve the resource units to be consumed. The checker is designed to admit decidable verification, and has been used on a realistic case study.

Type systems and program logics have been used to enforce other resource consumption policies: for example Vanderwaart and Crary [28] describe a type theory for certified code, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. In earlier work, Crary and Weirich [11] define a logic for reasoning about execution time of programs. Finally, some earlier work has shown how general purpose logics can be used to enforce security properties of Java programs, including confidentiality [4] and high-level security rules [21].

Works defining bytecode logics comprise [22] and [29]. In [22] a Hoare logic for bytecode is defined; the approach is based on searching structure in the bytecode programs which is not very natural for unstructured bytecode programs. In [29], on the other hand, a Hoare bytecode logics is defined in terms of weakest precondition calculus over the Jinja language (subset of Java). The logics is used for verifying bytecode against arithmetic overflow.

3. Preliminaries

3.1. Java class files

The standard format for Java bytecode programs is the so-called class file format which is specified in the Java Virtual Machine Specification [18]. For the purpose of this paper, it is sufficient to know that class files contain the definition of a single class or interface, and are structured into a hierarchy of different attributes that contain information such as the class name, the name of its superclass or the interfaces it implements, a table of the methods declared in the class. Moreover an attribute may contain other attributes. For example the attribute that describes a single method contains a `Local_Variable_Table` attribute that describes the method parameters and its local variables.

In addition to these attributes which provide all the information required by a standard implementation of the JVM, class files can accommodate user-defined attributes. We take advantage of this possibility and introduce additional attributes given in the Bytecode Specification Language, described below.

3.2. The Bytecode Specification Language

The *Bytecode Specification Language* (BCSL) [6] is a variant of the Java Modelling Language (JML) [17] tailored to Java bytecode. For our purposes, we only need to consider a restricted fragment of BCSL, which is given in Fig. 1; we let $\mathcal{E}$ and $\mathcal{P}$ denote respectively the set of BCSL expressions and predicates. As for JML, BCSL specifications contain different forms of statements, in the form of predicates tagged with appropriate keywords. BCSL predicates are built from expressions using standard predicate logic; furthermore BCSL expressions are bytecode programs that correspond to effect-free Java expressions, or BCSL specific expressions. The latter include expressions of the form `\old(exp)` which refers to the value of the expression `exp` at the beginning of the method, or $\texttt{exp}^{\texttt{pc}}$ which refers to the value of the expression `expr` at program point `pc`. Note that the latter is not standard in JML but can be emulated introducing a ghost variable $\texttt{exp}^{\texttt{pc}}$ and performing the ghost assignment `set` $\texttt{exp}^{\texttt{pc}}$`= exp` at program point `pc`.

Statements can be used for the following purposes:

- Specifying method preconditions, which following the design by contract principles, must be satisfied upon method invocation. They are formulated using statements of the form `requires` $\mathcal{P}$;
- Specifying method postconditions, which must be guaranteed upon returning normally from the method. Such postconditions are formulated using statements of the form `ensures` $\mathcal{P}$;
- Specifying method exceptional postconditions, which must be guaranteed upon returning exceptionally from the method. Such postconditions are formulated using statements of the form `exsures(Exception)`$\mathcal{P}$, that record the reason for exceptional termination;
- Stating loop invariants, which are predicates that must hold every time the program enters the loop: `invariant` $\mathcal{P}$;
- Guaranteeing termination of loops and recursive methods, using statements of the form `variant` $\mathcal{E}$ which provide a measure (in the case of BCSL, a positive number) that strictly decreases at each iteration of the loop/recursive call;
- Local assertions, using `assert` $\mathcal{P}$, which asserts that $\mathcal{P}$ holds at the program point immediately after the assertion;
- Declaring and updating ghost variables, using statements of the form `declare Model` *Type name* and `set` $\mathcal{E} = \mathcal{E}$;
- Keeping track of variables that are modified by a method or in a loop, using declarations of the form `modifies` *var*. During the generation of verification conditions, one checks that variables that are not declared as modifiable by the clause above will not be modified during the execution of the method/loop. This information is also used to generate the verification conditions.

Note that, as alluded above, annotations are not inserted directly into bytecode; instead they are gathered into appropriate user defined attributes of an extended class file. Such extended class files can be obtained either through direct manipulation of standard class files, or using an extended compiler that outputs extended class files from JML annotated programs, see [6].

$$
\begin{array}{rcl}
\text{BCSL} - \mathsf{stmt} & = & \texttt{requires}\ \mathcal{P} \\
& | & \texttt{ensures}\ \mathcal{P} \\
& | & \texttt{exsures(Exception)}\ \mathcal{P} \\
& | & \texttt{assert}\ \mathcal{P} \\
& | & \texttt{invariant}\ \mathcal{P} \\
& | & \texttt{variant}\ \mathcal{E} \\
& | & \texttt{declare Model}\ \mathit{Type}\ \mathit{name} \\
& | & \texttt{modifies}\ \mathit{var} \\
& | & \texttt{set}\ \mathcal{E} = \mathcal{E}
\end{array}
$$

Figure 1. Specification language

3.3. Verification of annotated bytecode

In order to validate annotated Java bytecode programs, we resort to a verification environment for Java bytecode, which is an adaptation by L. Burdy and the second author [6] of JACK [7]. The environment consists of two main components:

- A verification condition generator, which takes as input an annotated applet and generates a set of verification conditions which are sufficient to guarantee that the applet meets its specification;
- A proof engine that attempts to discharge the verification conditions automatically, and then sends the remaining verification conditions to proof assistants where they can be discharged interactively by the user.

3.3.1. Generating the Verification Conditions The verification condition generator (VCGen) takes as input an extended class file and returns as output a set of proof obligations, whose validity guarantees that the program satisfies its annotations. The VCGen proceeds in a modular fashion in the sense that it addresses each method separately, and is based on computing weakest preconditions. More precisely, for every method m, postcondition ψ that must hold after normal termination of m, and exceptional postcondition ψ' that must hold after exceptional termination of m (for simplicity we consider only one exception in our informal discussion), the VCGen computes a predicate ϕ whose validity at the onset of method execution guarantees that ψ will hold upon normal termination, and ψ' will hold upon exceptional termination. The VCGen will then return several proof obligations that correspond, among other things, to the fact that the precondition of m given by the specification entails the predicate ϕ that has been computed, and to the fact that variants and invariants are correct.

The procedure for computing weakest preconditions is described in detail in [6]. In a nutshell, one first defines for each bytecode a predicate transformer that takes as input the postconditions of the bytecode, i.e. the predicates to be satisfied upon execution of the bytecode (different predicates can be provided in case the bytecode is a branching instruction), and returns a predicate whose validity prior to the execution of bytecode guarantees the postconditions of the bytecode. The definition of such functions is based on a single instruction, so the next step is to use these functions to compute weakest preconditions for programs. This is done by building the control flow graph of the program, and then by computing the weakest preconditions of the program, using the graph.

Note that the verification condition generator operates on BCSL statements which are built from extended BCSL expressions. Indeed, predicate transformers for instructions need to refer to the operand stack and must therefore consider expressions of the form `st(i)` which represent the `i`-element of the stack `st`.

3.3.2. Discharging verification conditions Verification conditions are expressed in an intermediate language and then translated to automatic theorem provers and proof assistants. In our examples, we have used Simplify [25] as automatic prover and Coq [10] as proof assistant. The Coq plug-in for Jack was developed by J. Charles, and adapted to Java bytecode by L. Burdy.

3.4. Correctness of the method

The verification method is correct in the sense that one can prove that for all methods m of the program the (exceptional) postcondition of the method holds upon (exceptional) termination of the method provided the method is called in a state satisfying the method precondition and provided all verification conditions can be shown to be valid.

The correctness of the verification method is established relative to an operational semantics that describes the transitions to be taken by the virtual machine depending upon the state in which the machine is executed. There are many formalisations of the operational semantics of the JVM, see e.g. [13, 16, 26, 27].

We have proved the correctness of our method for a fragment of the JVM that includes the following constructs: Stack manipulation: `push`, `pop`, `dup`, `dup2`, `swap`, `numop`, etc; Arithmetic instructions: type_`add`, type_`sub`, etc; Local variables manipulation: type_`load`, type_`store`, etc; Jump instructions: `if`, `goto`; Object creation and object manipulation: `new`, `putfield`, `getfield`, `newarray`, etc; Ar-

ray instructions: `arraystore`, `arrayload`, etc; Method calls and return: `invokevirtual`, `return`; Subroutines: `jsr` and `ret`.

Note however that our method imposes some mild restrictions on the structure of programs: for example, we require that `jsr` and `throw` instructions are not entry for loops in the control flow graph in order to prevent pathological recursion. Lifting such restrictions is left for future work.

4. Modelling memory consumption

The objective of this section is to demonstrate how the user can annotate and verify programs in order to obtain an upper bound on memory consumption. We begin by describing the principles of our approach, then turn to establish its soundness, and finally show how it can be applied to non-trivial examples involving recursive methods and exceptions.

4.1. Principles

Let us begin with a very simple memory consumption policy which aims at enforcing that programs do not consume more than some fixed amount of memory `Max`. To enforce this policy, we first introduce a ghost variable `Mem` that represents at any given point of the program the memory used so far. Then, we annotate the program both with the policy and with additional statements that will be used to check that the application respects the policy.

The precondition of the method m should ensure that there must be enough free memory for the method execution. Suppose that we know an upper bound of the allocations done by method m in any execution. We will denote this upper bound by `mthdCon(`m`)`. Thus there must be at least `mthdCon(`m`)` free memory units from the allowed `Max` when method m starts execution. Thus the precondition for m is:

$$//@\ \texttt{requires}\ \texttt{Mem} + \texttt{mthdCon(}m\texttt{)} \leq \texttt{Max}.$$

The precondition of the program entry point (i.e., the *main* method from which an application may start its execution) should also give the initial memory used by the virtual machine, i.e. require that variable `Mem` is equal to some fixed constant.

The normal postcondition of the method m must guarantee that the memory allocated during a normal execution of m is not more than some fixed number `mthdCon(`m`)` of memory units. Thus for the method m the postcondition is:

$$//@\ \texttt{ensures}\ \texttt{Mem} \leq \backslash\text{old}(\texttt{Mem}) + \texttt{mthdCon(}m\texttt{)}.$$

The exceptional postcondition of the method m must specify that the memory allocated during an execution of m terminating by throwing an exception `Exception` is not more than `mthdCon(`m`)` units. Thus for the method m the exceptional postcondition is:

$$//@\ \texttt{exsures(Exception)}\ \texttt{Mem} \leq \backslash\text{old}(\texttt{Mem}) + \texttt{mthdCon(}m\texttt{)}.$$

For every instruction that allocates memory the ghost variable `Mem` must be updated accordingly. For the purpose of this paper, we only consider dynamic object creation with the bytecode `new`; arrays are left for future work and briefly discussed in the conclusion.

In order to perform the update for `new` bytecodes, we assume given a function `allocInst` : $Class \rightarrow int$ gives an estimation of the memory used by an instance of a class. Then at every program point where a bytecode `new A` is found, the ghost variable `Mem` must be incremented by `allocInst(A)`. This is achieved by inserting a ghost assignment associated with any `new` instruction, as shown below:

$$\texttt{new A} \quad //\texttt{set Mem} = \texttt{Mem} + \texttt{allocInst(A)}.$$

4.2. Correctness

An important question is whether our approach guarantees that the memory allocated by a given program conforms to the memory consumption policy imposed by BCSL annotations. We can prove that our approach is correct by instrumenting the operational semantics of the bytecode language to reflect memory consumption. Concretely, this is achieved by extending states with the special variable `Mem`, and describing for each bytecode and for ghost assignments the effect of the weakest precondition calculus on `Mem` (in the fragment of the language considered, the only instruction to modify memory is `new`, thus the only instruction whose weakest precondition calculus has an effect on `Mem` is `new`).

We can then prove the correctness of the annotations w.r.t. the instrumented operational semantics, under the proviso that ghost assignments triggered by object creation are compatible with the instrumented operational semantics.

4.3. Examples

We illustrate hereafter our approach by several examples, coping with recursive and overridden methods and with exceptions.

Specification of method m in class A:

```
//@ requires  Mem + k ≤ Max
//@ modifies  Mem
//@ ensures   Mem ≤ \old(Mem) + k
```

Specification for method m in class B:

```
//@ requires  Mem + l ≤ Max
//@ modifies  Mem
//@ ensures   Mem ≤ \old(Mem) + l
```

```
public void n (A a)
...
//Prove  Mem + max(l,k) <= Max
invokevirtual <A.m>
//Assume  Mem <= \old(Mem) + max(l,k)
...
```

Figure 2. EXAMPLE OF OVERRIDDEN METHODS

4.3.1. Inheritance and overridden methods Overriding methods are treated as follows: whenever a call to a method m is performed, we require that there is enough free memory space for the maximal consumption by all the methods that override or are overridden by m. In Fig. 2 we show a class `A` and its extending class `B`, where `B` overrides the method m from class `A`. Method m is invoked by method n. Given that the dynamic type of the parameter passed to n is not known, we cannot know which of the two methods will be invoked. This is the reason for requiring enough memory space for the execution of any of these methods.

4.3.2. Recursive Methods In Fig. 3 the bytecode of the recursive method m and its specification are shown. For simplicity we show only a simplified version of the bytecode; we assume that the constructors for the class `A` and `C` do not allocate memory. Besides the pre- and the postcondition, the specification also includes information about the termination of the method: `variant localVar`(1), meaning that the local variable `localVar`(1) decreases on every recursive call down to 0, guaranteeing that the execution of the method will terminate.

We explain first the precondition. If the condition of line `1` is not true, the execution continues at line `2`. In the sequential execution up to line `7`, the program allocates at most `allocInst(A)` memory units and decrements by 1 the value of `localVar`(1). The instruction at line `8` is a recursive call to m, which either will take the same branch if `localVar`(1) $>$ 0 or will jump to line `12` otherwise, where it allocates at most `allocInst(A)`+`allocInst(C)` memory units.

```
public class D {
 public void m (int i) {
   if (i > 0) { new A(); m(i-1); new A(); }
   else { new C(); new A(); } } }
```

```
//@ requires Mem + allocInst(A) + allocInst(C))
        +localVar(1) * 2 * allocInst(A) ≤ Max
//@ variant localVar(1)
//@ ensures localVar(1) ≥ 0 ∧ Mem ≤ \old(Mem) +
      \old(localVar(1)) * 2 * allocInst(A) +
      allocInst(A) + allocInst(C))
```

```
public void m()
0 load_1 //Local var. loaded on the stack of m
1 ifle 12 //If localVar(1) <= 0 jump
2 new <A> //Here localVar(1) > 0
//set Mem = Mem + allocInst(A)
3 invokespecial <A.<init>>
4 aload_0
5 iload_1
6 iconst_1
7 isub //localVar(1) decremented with 1
8 invokevirtual <D.m> //Recursive call
9 new <A>
//set Mem = Mem + allocInst(A)
10 invokespecial <A.<init>>
11 goto 16
12 new <A> //Target of the jump at 1
//set Mem = Mem +  allocInst(A)
13 invokespecial <A.<init>>
14 new  <C>
//set Mem = Mem + allocInst(C)
15 invokespecial <C.<init>>
16 return
```

Figure 3. EXAMPLE OF A RECURSIVE METHOD

On returning from the recursive call one more allocation will be performed at line `9`. Thus m will execute, `localVar`(1) times, the instructions from lines `2` to `7`, and it finally will execute all the instructions from lines `12` to `16`.

The postcondition states that the method will perform no more than \old(`localVar`(1)) recursive calls (i.e., the value of the register variable in the pre-state of the method) and that on every recursive call it allocates no more than two instances of class `A` (one corresponding to line `2` and the other to line `9`) and that it will finally allocate one instance of class `A` (line `12`) and another of class `C` (line `14`).

For proving the correctness of this method, 18 proof obligations were generated with Jack, most of which were automatically proved in Coq using its standard tactics.

Loops must also be annotated with appropriate invariants, and with variants that guarantee their termination. If we know that some expression e bound by `0` decreases at every iteration of the loop, and that each loop iteration will not allocate more than k units, then we can strengthen the loop invariant to:

```
//@ modifies i, Mem
//@ invariant 0 ≤ e ∧ Mem ≤ Mem^{Before_l} + e * k
//@ variant e
```

$\texttt{Mem}^{Before_l}$ is a special variable denoting the value of the consumed memory just before entering for the first time the loop l. At every iteration the consumed memory must not go beyond the upper bound given for the body of loop.

4.3.3. More precise specification We can be more precise in specifying the precondition of a method by considering the field values of an instance, for example. Let m be the method shown in Fig. 4 and assume no allocations are done in the constructor of the class `A`. The first line of m initialises one of the fields of field `b`. Since nothing guarantees that `b` is not `null`, the execution may terminate with `NullPointerException`. Depending on the values of the parameters passed to m, the memory allocated will be different. The precondition specifies the required memory space depending on whether the field `b` is `null` or not. In the normal postcondition we state that the method has allocated an object of class `A`. The exceptional postcondition states that no allocation is performed if `NullpointerException` causes the termination.

5. Inferring memory allocation

In the previous section, we have described how the memory consumption of a program can be modelled in BCSL and verified using an appropriate verification environment. While our examples illustrate the benefits of our approach, especially regarding the precision of the analysis, the applicability of our method is hampered by the cost of providing the annotations manually. In order to reduce the burden of manually annotating the program, one can rely on annotation assistants that infer automatically some of the program annotations (indeed such assistants already exist for loop invariants [20] and class invariants [19]). In this section, we describe an implementation of an annotation assistant dedicated to the analysis of memory consumption, and illustrate its functioning on an example.

```
//@ requires !(localVar(1) == null) ⇒
                    Mem + allocInst(A) ≤ Max
//@ modifies Mem
//@ ensures Mem ≤ \old(Mem) + allocInst(A)
//@ exsures(NullPointerException)
                          Mem == \old(Mem)

0 aload_0                  public class C     {
1 getfield<C.b>              B b;
2 iload_2                    public void
3 putfield <B.i>               m (A a, int i){
4 new <A>                        b.i = i ;
//set Mem = Mem +                a = new A();}}
      allocInst(A)
5 dup
6 invokespecial <A.<init>>
7 astore_1
8 return
```

Figure 4. EXAMPLE OF A METHOD WITH POSSIBLE EXCEPTIONAL TERMINATION

5.1. Annotation assistant

The annotation assistant performs two tasks. First, it inserts the ghost assignments on appropriate places; for this task, the user must provide annotations about the memory required to create objects of the given classes.

Second, it inserts pre- and postconditions for each method. In this case, variants for loops and recursive methods may be given by the user or be synthesised through appropriate mechanisms. Based on this information, the annotation assistant recursively computes the memory allocated on each loop and method. Essentially, it finds the maximal memory that can be allocated in a method by exploring all its possible execution paths.

The function `mthdCon(.)` is defined as follows:

- **Input:** Annotated bytecode of a method m, and memory policies for methods that are called by m;
- **Output:** Upper bound of the memory allocated by m;
- **Body:** The first step is to compute the loop structure of the method, then to compute an upper bound to the memory allocated by each loop using its variant, and then to compute an upper bound to the memory allocated along each execution path.

The annotation assistant currently synthesises only simple memory policies (i.e., whenever the memory consumption policy does not depend on the input). Furthermore, it does not deal with arrays, subroutines, nor exceptions, and is restricted to loops with a unique entry point. The latter restriction is not critical because it accommodates code produced by non-optimising compilers. However, a pre-analysis could give us all the entry points of more general loops, for instance by the algorithms given in [8]; our approach may be thus applied straightforwardly. How to treat arrays is briefly discussed in the conclusion.

5.2. Example

Let us consider the bytecode given in Fig. 5, which is a simplified version of the bytecode corresponding to the source code given in the right of the figure. For simplicity of presentation, we do not show all the instructions (the result of the inference procedure is not affected). Method m has two branching instructions, where two objects are created: one instance of class `A` and another of class `B`. Our inference algorithm gives that `mthdCon(`m`)` = `allocInst(A)`+ `mthdCon(A.init)` + `allocInst(B)` + `mthdCon(B.init)`.

```
0 aload_1                        public void
1 ifnonnull 6                     m (A a , B b )   {
2 new <A>                          if (a == null) {
...                                  a = new A(); }
4 invokespecial <A.<init>>         if (b == null) {
6 aload_2                            b = new B(); }}
7 ifnonnull 12
8 new <B>
...
10 invokespecial <B.<init>>
...
12 return
```

Figure 5. EXAMPLE

6. Conclusion

Program logics have traditionally been used to verify functional properties of applications, but we have shown that such logics are also appropriate to enforce security properties including memory consumption policies. We have shown that program logics complement nicely existing methods to verify memory consumption, over which they are superior in terms of the precision of the analysis (and inferior in terms of automation).

We intend to pursue our work in four directions. Firstly, we would like to extend our approach to arrays. In principle, it should be reasonably easy to extend the verification method to arrays; however, it seems more complicated to extend our inference algorithm to arrays. The main difficulty here is to provide an estimate of the size of an array, as it is given by the top value on the operand stack at the time of its creation. Our intuition is that this can be done using an abstract interpretation or a symbolic evaluation of the program. If we look at the example code below (in source code):

```
void m(int s)
   { int len = s;  int[] i = new int[len] }
```

where `len` is a local variable to the method, one can infer by symbolic computation that its value is the value of the method parameter. Thus the method can be given the precondition `Mem + s.sizeof(int) <= Max`. In a similar line of work, we would like to extend our results to concurrency using recent advances in program logics for multi-threaded Java programs [23]. Providing an appropriate treatment of arrays and multi-threading is an important step towards applying our results to mobile phone applications.

Secondly, we would like to adapt our approach to account for explicit memory management. More precisely, we would like to consider an extended language with a special instruction `free(o)` that deallocates the object `o`, and establish the correctness of our method under the assumption that deallocation is correct, i.e. that the object `o` is not reachable from the program point where `free(o)` is inserted. By combining our approach with existing compile-time analysis that infers for each program point which objects are not reachable, we should be able to provide more precise estimates of memory consumption.

Thirdly, we intend to apply our technique to other resources such as communication channels, bandwidth, and power consumption, as well as to more refined analysis that distinguish between different kinds of memory, such as RAM or non-volatile EEPROM. As suggested by the MRG project [3], it seems also interesting to consider policies that enforce limits on the interaction between the program and its environment, for example w.r.t. the number of system calls or the bounds on parameters passed to them.

Finally, we envisage to experiment with more complex applets and to compare the results with other approaches.

Acknowledgements The authors are grateful to the anonymous referees for their comments on the paper, to

Lilian Burdy and Julien Charles for making available a Coq plug-in for verifying annotated bytecode programs, and to the members of the CASTLES project for useful discussions on memory consumption.

References

[1] R. Amadio and S. Dal Zilio. Resource Control for Synchronous Cooperative Threads. In P. Gardner and N. Yoshida, editors, *Proceedings of CONCUR'04*, volume 3170 of *Lecture Notes in Computer Science*, pages 68–82. Springer-Verlag, August 2004.

[2] D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *Journal of Automed Reasoning*, 31(3-4):261–302, 2003.

[3] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 1–27. Springer-Verlag, 2005.

[4] G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.

[5] V. Braberman, D. Garbervetsky, and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTfJP'05*, 2005. To appear.

[6] L. Burdy and M. Pavlova. Annotation carrying code. Manuscript, 2005.

[7] L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME'03*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[8] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Proceedings of FM'05*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106, 2005. To appear.

[9] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag, 2005.

[10] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.

[11] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of POPL'00*, pages 184–198. ACM Press, 2000.

[12] L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In K. Havelund and G. Rosu, editors, *Proceedings of RV'04*, volume 113 of *Electronic Notes in Theoretical Computer Science*, pages 217–233. Elsevier, 2004.

[13] S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.

[14] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In K. Havelund and G. Rosu, editors, *Proceedings of RV'04*, volume 113 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[15] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of POPL'03*, pages 185–197. ACM Press, 2003.

[16] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.

[17] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. O'Reilly, 1999.

[19] F. Logozzo. Automatic inference of class invariants. In G. Levi and B. Steffen, editors, *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*, pages 211–222. Springer-Verlag, 2004.

[20] J.W. Nimmer and M.D. Ernst. Automatic generation of program specifications. In *Proceedings of ISSTA'02*, volume 27, 4 of *Software Engineering Notes*, pages 232–242. ACM Press, 2002.

[21] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*. Kluwer, 2004.

[22] C.L. Quigley. A programming logic for java bytecode programs. In D. Basin and B. Wolff, editors, *Proceedings of TPHOLs'03*, volume 2758 of *Lecture Notes in Computer Science*, pages 41–54. Springer-Verlag, 2003.

[23] E. Rodriguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *Proceedings of ECOOP'05*, volume 3xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2005. To appear.

[24] G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, December 2004.

[25] Simplify. `http://research.compaq.com/SRC/esc/Simplify.html`.

[26] I. Siveroni. Operational semantics of the Java Card Virtual Machine. *J. Logic and Algebraic Programming*, 58(1-2), 2004.

[27] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer-Verlag, 2001.

[28] J. C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.

[29] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In S. Sagiv, editor, *Proceedings of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 326–341. Springer-Verlag, 2005.