Heap Space Analysis for Java Bytecode

Elvira Albert
DSIC, Complutense University of Madrid, Spain
elvira@sip.ucm.es

Samir Genaim
CLIP, Technical University of Madrid, Spain
samir@clip.dia.fi.upm.es

Miguel Gómez-Zamalloa
DSIC, Complutense University of Madrid, Spain
mzamalloa@fdi.ucm.es

Abstract

This article presents a heap space analysis for (sequential) Java bytecode. The analysis generates *heap space cost relations* which define at compile-time the heap consumption of a program as a function of its data size. These relations can be used to obtain upper bounds on the heap space allocated during the execution of the different methods. In addition, we describe how to refine the cost relations, by relying on *escape analysis*, in order to take into account the heap space that can be safely deallocated by the garbage collector upon exit from a corresponding method. These refined cost relations are then used to infer upper bounds on the *active heap space* upon methods return. Example applications for the analysis consider inference of constant heap usage and heap usage proportional to the data size (including polynomial and exponential heap consumption). Our prototype implementation is reported and demonstrated by means of a series of examples which illustrate how the analysis naturally encompasses standard data-structures like lists, trees and arrays with several dimensions written in object-oriented programming style.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

Keywords Heap Space Analysis, Heap Consumption, Low-level Languages, Java Bytecode

1. Introduction

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [24, 17, 25, 19] in the context of functional programming and [18, 13] for high-level imperative programming languages). However, there are situations where one has only access to compiled code and not to the source code. An example of this is *mobile code*, where the code consumer receives code to be executed. In this context, Java bytecode [20] is widely used, mainly due to its security features and the fact that it is platform-independent. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [14, 4, 5, 16, 12] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered. A general framework for the cost analysis of sequential Java bytecode has been proposed in [2]. Such analysis statically generates *cost relations* which define the cost of a program as a function of its input data size. The cost relations are expressed by means of *recursive equations* generated by abstracting the recursive structure of the program and by inferring size relations between arguments. Cost relations are parametric w.r.t. a *cost model*, i.e., the cost unit associated to the bytecode b appears as an abstract value T_b within the equations.

This article develops a novel application of the cost analysis framework of [2] to infer bounds on the heap space consumption of sequential Java bytecode programs. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units it consumes. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space cost relations which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program. In a second step, we refine this cost model to consider the effect of garbage collection. This is done by relying on escape analysis [15, 8] to identify those memory allocation instructions which create objects that will be

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.

garbage collected upon exit from the corresponding method. With this information available, we can generate heap space cost relations which contain annotations for the heap space that will be garbage collected. The annotated cost relations in turn are used to infer upper bounds on the *active heap space* upon exit from methods, i.e., the heap space consumed and that might not be garbage collected upon exit.

A distinguishing feature of the approach presented in this article w.r.t. previous type-based approaches (e.g., [5, 17]) is that it is not restricted to linear bounds since the generated cost relations can in principle capture any complexity class. Moreover, in many cases, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run. The approach has been assessed by means of a prototype implementation, which originates from the one of [3]. It should be noted that the examples in [3] are simple imperative algorithms which did not make use of the heap, since they were aimed at demonstrating that traditional complexity schemata can be handled by the cost analysis of [2]. In contrast, we demonstrate our heap analysis by means of a series of example applications written in an object-oriented style which make intensive use of the heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java bytecode.

The rest of the paper is structured as follows: Sec. 2 presents an example that illustrates the ideas behind the analysis. Sec. 3 briefly describes the Java bytecode language. Sec. 4 defines a cost model for heap consumption and describes the analysis framework. Sec. 5 demonstrates the different features of the analysis by means of examples. In Sec.6, we extend our cost model to consider the effect of garbage collection. Sec. 7 reports on a prototype implementation and some experimental results. Finally, Sec. 8 concludes and discusses the related work.

2. Worked Example

Consider the Java classes and their corresponding (structured) Java bytecode depicted in Fig. 1 which define a linked-list data structure in an object-oriented style, as it appears in [18]. The class *Cons* is used for data nodes and the class *Nil* plays the role of *null* to indicate the end of a list. Both classes define a copy function which is used to clone the corresponding object. In the case of *Nil* the copy method just returns *this* since it is the last element of the list, and in the case of *Cons* it clones the current object and its successors recursively (by calling the *copy* method of *next*). The rest of this section describes the different steps applied by the analyzer to approximate the heap consumption of the program depicted in Fig. 1. Note that the Java program is provided here just for clarity, the analyzer works directly on the bytecode which is obtained, for example, by compiling the Java program.

Step I: In the first step, the analyzer recovers the structure of the Java bytecode program by building a control flow graph (CFG) for its methods. The CFG consists of basic blocks which contain a sequence of non-branching bytecode instructions, these blocks are connected by edges that describe the possible flows that originate from the branching instructions like conditional jumps, exceptions, virtual method invocation, etc. In Fig. 1, the CFG of the method *Nil.copy* consists of the single block $Block_0^{Nil}$ and the CFG of the method *Cons.copy* consists of the rest of the blocks. $Block_0^{Cons}$ corresponds to the bytecode of *Cons.copy* up to the recursive method call *this.next.copy()*. Then, depending on the type of the object stored in *this.next* the execution is transferred to either *Nil.copy* or *Cons.copy*. This is expressed by the (guarded) branching to $Block_1^{Cons}$ and $Block_2^{Cons}$. In both cases, the control returns to $Block_3^{Cons}$ which corresponds to the rest of the statements.

Step II: In the second step, the analyzer builds an intermediate representation for the CFG and uses it to infer information about the changes in the sizes of the different data-structures (or in the values of integer variables) when the control passes from one part of the program (e.g., a block or a method) to another part. For example, this step infers that when *Nil.copy* or *Cons.copy* are called recursively, the length of the list decreases by one. This information is essential for defining the heap consumption of one part of the program in terms of the heap consumption of other parts.

Step III: In the third step, the intermediate representation and the size information are used together with the cost model for heap consumption to generate a set of cost relations which describe the heap consumption behaviour of the program. The following equations are the ones we get for the example in Fig. 1:

Heap Space Cost Equations			Size relations
$C_{copy}^{Nil}(a)$	=	0	$\{a{=}1\}$
$C_{copy}^{Cons}(a)$	=	$C_0(a)$	
$C_0(a)$	=	$size(Cons) + CC_0(a,b)$	$\{a{\geq}1, b{\geq}0, a{=}b{+}1\}$
$CC_0(a,b)$	=	$\begin{cases} C_1(a,b) & \hat{b} \in Nil \\ C_2(a,b) & \hat{b} \in Cons \end{cases}$	
$C_1(a,b)$	=	$C_{copy}^{Nil}(b) + C_3(a)$	$\{a{=}1\}$
$C_2(a,b)$	=	$C_{copy}^{Cons}(b) + C_3(a)$	$\{a{\geq}2\}$
$C_3(a)$	=	0	

Each of these equations corresponds to a method entry, block or branching in the CFG. An equation is composed by the left hand side which indicates the block or the method it represents, and the right hand side which defines its heap consumption behaviour. In addition, size relations might be attached to describe how the data size changes when using another equation.

```
abstract class List {
    abstract List copy();
}
class Nil extends List {
    List copy() {
        return this;
    }
}
class Cons extends List {
    int elem;
    List next;
    List copy(){
        Cons aux = new Cons();
        aux.elem = this.elem;
        aux.next = this.next.copy();
        return aux;
    }
}
```

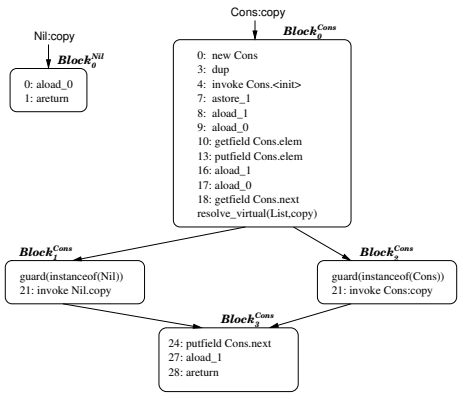

Figure 1. Java source code and CFG bytecode of example

The equation $C^{Nil}_{copy}(a)$ defines the heap consumption of *Nil.copy* in terms of (the size of) its first argument a which corresponds to its *this* reference variable (in Java bytecode the *this* reference variable is the first argument of the method). In this case the heap consumption is zero since the method does not allocate any heap space. The equation $C^{Cons}_{copy}(a)$ defines the heap consumption of *Cons.copy* as the heap consumption of $Block^{Cons}_0$ using the corresponding equation C_0, which in turn defines the heap consumption as the amount of heap units allocated by the *new* bytecode instructions, namely $size(Cons)$, plus the heap consumption of its successors which is defined by the equation CC_0. All other instructions in $Block^{Cons}_0$ contribute zero to the heap consumption. Note that in C_0, the variable b corresponds to *this.next* of *Cons.copy* and that the size analysis is able to infer the relation $a{=}b{+}1$ (i.e., the list a is longer than b by one). The equation CC_0 corresponds to the heap consumption of the branches at the end of $Block^{Cons}_0$, depending on the type of b (denoted as $\hat{b}$) it is equal to the heap consumption of $Block^{Cons}_1$ or $Block^{Cons}_2$ which are respectively defined by the equations C_1 and C_2. The equation C_1 defines the heap consumption of $Block^{Cons}_1$ as the heap consumption of *Nil.copy* (since it is called in $Block^{Cons}_1$) plus the heap consumption of $Block^{Cons}_3$ (using the equation C_3). Similarly C_2 defines the heap consumption of $Block^{Cons}_2$ in terms of the heap consumption of *Cons.copy*. The equation C_3 defines the heap consumption of $Block^{Cons}_3$ to be zero since it does not allocate any heap space.

Step IV: In the fourth step, we can simplify the equations and try to obtain an upper bound in closed form for the cost relation by applying the method described in [1]. In particular, assuming that $size(Cons)$ equals 8 (4 bytes for the integer field *data* and 4 bytes for the reference field *next*), we obtain the following simplified equations:

Equation			Size relations
$C^{Nil}_{copy}(a)$	=	0	$\{a{=}1\}$
$C^{Cons}_{copy}(a)$	=	8	$\{a{=}2\}$
$C^{Cons}_{copy}(a)$	=	$8 + C^{Cons}_{copy}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

and then obtain an upper bound in closed form $C^{Cons}_{copy}(a) = 8 * (a-1)$.

The main focus of this paper is on the generation of heap space cost relations, as illustrated in Step III. Steps I and II are done as it is proposed in [2] and Step IV as it is described in [1] and hence we will not give many details on how they are performed in this paper.

3. The Java Bytecode Language

Java bytecode [20] is a low-level object-oriented programming language with unstructured control and an *operand stack* to hold intermediate computational results. Moreover, objects are stored in dynamic memory: the *heap*. A Java bytecode program consists of a set of *class files*, one for each class or interface. A class file contains information about its *name* $c \in Class\_Name$, the class it extends, the interfaces it implements, and the fields and methods it defines. In particular, for each method, the class file contains: a method signature which consists of its name and its type; its bytecode $bc_m = \langle pc_0{:}b_0, \ldots, pc_{n_m}{:}b_{n_m}\rangle$, where each b_i is a *bytecode instruction* and pc_i is its address; and the method's exceptions table. In this work we consider a subset of the JVM [20] language which is able to handle operations on integers and references, object creation and manipulation (by accessing fields and calling methods), arrays of primitive and reference types, and exceptions (either generated by abnormal execution or explicitly thrown by the program). For simplicity, we omit static fields and initializers and primitive types different from integers. Such features could be handled by making the underlying abstract interpretation support them by assuming the worst case approximation for them. Thus, our bytecode instruction set ($bcInst$) is:

$bcInst$ ::=
  push x | istore v | astore v | iload v | aload v | iconst a
| iadd | isub | imul | idiv | if$\diamond$ pc | goto pc | ireturn | areturn
| return | new $Class\_Name$ |
| newarray int | anewarray $Class\_Name$ | iaload | aaload
| iastore | aastore | athrow | dup
| invokevirtual/invokespecial $Class\_Name.Meth\_Sig$
| getfield/putfield $Class\_Name.Field\_Sig$

where $\diamond$ is a comparison operator (ne,le,_icmpgt, etc.), v a local variable, a an integer, pc an instruction address, and x an integer or the special value `null`.

4. The Heap Space Analysis Framework

Cost analysis of a low-level object-oriented language such as Java bytecode is complicated mainly due to its unstructured control flow (e.g., the use of goto statements rather than recursive structures), its object-oriented features (e.g., virtual method invocation) and its stack-based model. The recent work of [2] develops a generic framework for the automatic cost analysis of Java bytecode programs. Essentially, the complications of dealing with a low-level language are handled in this framework by abstracting the *recursive structure* of the program and by inferring *size relations* between arguments. As we have seen in Sect. 2, this analysis framework is based on transforming the Java bytecode program to an intermediate representation which fits inside the same setting all possible forms of loops. Then, using this intermediate representation, the analysis infers information about the change in the sizes of the relevant data structures as the program goes through its loops (Steps I and II). Finally, this information is used to set up a cost relation which defines the cost of the program in terms of the sizes of the corresponding data structures.

In this section, we present a novel application of this generic cost analysis framework to infer bounds on the heap space consumption of sequential Java bytecode programs. So far, this framework has been only used in [3] to infer the complexity of some classical algorithms while in this paper our purpose is completely different: we aim at computing bounds on the heap usage for programs written in object-oriented programming style which make intensive use of the heap. In Sect. 4.1 and Sect. 4.2, we briefly present the notions of recursive representation and calls-to size-relation in a rather informal style. Then, we introduce our cost model for heap consumption and our notion of heap space cost relation in Sect. 4.3.

4.1 Recursive Representation

Cost relations can be elegantly expressed as systems of *recursive* equations. In order to automatically generate them, we need to capture the iterative behaviour of the program by means of recursion. One way of achieving this is by computing the CFG of the program. Also, advanced features like virtual invocation and exceptions are simply dealt as additional nodes in the graph. To analyze the bytecode, its CFG can be represented by using some auxiliary recursive representation (see, e.g., [2]). In this approach, a *bytecode* is transformed into a set of *guarded rules* of the form $\langle head \leftarrow guard, body \rangle$ where the *guard* states the applicability conditions for the rule. Rules are obtained from blocks in the CFG and *guards* indicate the conditions under which each block is executed. As it is customary in deterministic imperative languages, guards provide mutually exclusive conditions because paths from a block are always exclusive (i.e., alternative) choices.

DEFINITION 4.1 (rec. representation). *Consider a block* $\mathtt{p}$ *in a CFG, which contains a sequence of bytecode instructions* $\mathtt{B}$ *guarded by the condition* $\mathtt{G_b}$ *and whose successor blocks are* $\mathtt{q_1}, \cdots, \mathtt{q_n}$. *The* recursive representation *of* $\mathtt{p}$ *is:*

$$\mathtt{p}(\bar{\mathtt{l}}, \bar{\mathtt{s}}, \mathtt{r}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}); \cdots; \mathtt{q_n}(\bar{\mathtt{l}}, \bar{\mathtt{s}}', \mathtt{r}))$$

where:

- $\bar{l}$ *is a tuple of variables which corresponds to the method's local variables,*
- $\bar{s}$ *and* $\bar{s}'$ *are tuples of variables which respectively correspond to the active stack elements at the block's entry and exit,*
- r *is a single variable which corresponds to the method's return value (omitted if there is not return value),*
- G_p *and* B *are obtained from the block's guard and bytecode instructions by adding the local variables and stack elements on which they operate as explicit arguments.*

We denote by $\mathtt{calls}(\mathtt{B})$ *the set of method invocation instructions within* $\mathtt{B}$ *and by* $\mathtt{bytecode}(\mathtt{B})$ *the other instructions.* □

The formal translation of bytecode instructions in `B` to calls within the recursive rules is presented in [2]. In this translation, it is interesting to note that the stack positions are visible in the rules by explicitly defining them as local variables. This intermediate representation is convenient for analysis as in one pass we can eliminate almost all stack variables which results in a more efficient analysis.

EXAMPLE 4.2. *The rules that correspond to the blocks* $Block_0^{Cons}$, $Block_1^{Cons}$ *and* $Block_2^{Cons}$ *in Fig. 1 are:*

$$\begin{array}{l}
\mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{r}) \leftarrow \\
\quad \mathtt{new}(\mathtt{Cons}, \mathtt{s}_0), \mathtt{dup}(\mathtt{s}_0, \mathtt{s}_1), \mathtt{Cons.<init>}(\mathtt{s}_1), \\
\quad \mathtt{astore}(\mathtt{s}_0, \mathtt{aux}'), \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0'), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'), \\
\quad \mathtt{getfield}(\mathtt{Cons.elem}, \mathtt{s}_1', \mathtt{s}_1''), \\
\quad \mathtt{putfield}(\mathtt{Cons.elem}, \mathtt{s}_0', \mathtt{s}_1''), \\
\quad \mathtt{aload}(\mathtt{aux}', \mathtt{s}_0''), \mathtt{aload}(\mathtt{this}, \mathtt{s}_1'''), \\
\quad \mathtt{getfield}(\mathtt{Cons.next}, \mathtt{s}_1''''), \\
\quad (\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}) ; \\
\qquad \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r})).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Nil})), \\
\quad \mathtt{Nil.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

$$\begin{array}{l}
\mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1, \mathtt{r}) \leftarrow \\
\quad \mathtt{guard}(\mathtt{instanceof}(\mathtt{s}_1, \mathtt{Cons})), \\
\quad \mathtt{Cons.copy}(\mathtt{s}_1, \mathtt{s}_1'), \\
\quad \mathtt{copy}_3^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}, \mathtt{s}_0, \mathtt{s}_1', \mathtt{r}).
\end{array}$$

The rule $\mathtt{copy}_0^{\mathtt{Cons}}$ *is not guarded and has two continuation blocks, while the other rules are guarded by the type of*

the object of s_1 (the top of the stack) and have only one successor. The bytecode instructions were transformed to include explicitly the stacks elements and the local variables on which they operate, moreover, all variables are in single static assignment *form. Note that calls to methods take the same form as calls to blocks, which makes all different forms of loops to fit in the same setting.* □

4.2 Size Analysis

A size analysis is then performed on the recursive representation in order to infer the *calls-to size-relations* between the variables in the head of the rule and the variables used in the calls (to rules) which occur in the body for each program rule. Derivation of constraints is a standard abstract interpretation over a constraints domain such as Polyhedra [2, 3]. Such relations are essential for defining the cost of one block in terms of the cost of its successors. The analysis is done by abstracting the bytecode instructions into the linear constraints they impose on their arguments, and then computing a fixpoint that collects *calls-to* relations.

DEFINITION 4.3 (calls-to size-relations). *Consider the rule in Def. 4.1, its calls-to* size-relations *are triples of the form*

$$\langle \mathtt{p}(\bar{\mathtt{x}}), \mathtt{p}'(\bar{\mathtt{z}}), \varphi \rangle \quad \textit{where } \mathtt{p}'(\bar{\mathtt{z}}) \in \mathtt{calls(B)} \cup \mathtt{q_1}(\bar{\mathtt{y}}) \cup \ldots \cup \mathtt{q_n}(\bar{\mathtt{y}})$$

The size-relation φ *is given as a conjunction of linear constraints. The tuples of variables $\bar{x}$, $\bar{y}$ and $\bar{z}$ correspond to the variables of the corresponding block.* □

In Java bytecode, we consider three cases within size relations: for integer variables, *size-relations* are constraints on the possible values of variables; for reference variables, they are constraints on the length of the longest reachable paths [21], and for arrays they are constraints on the length of the array. Note that using the path-length notion cyclic structures are not handled since to guarantee soundness the corresponding references are abstracted to "unknown-length" and therefore cost that depends on them cannot be inferred.

EXAMPLE 4.4. *The calls-to-size relation for the first rule in Ex. 4.2 is formed by the triples:*

$$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_1^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$$
$$\langle \mathtt{copy}_0^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}), \mathtt{copy}_2^{\mathtt{Cons}}(\mathtt{this}, \mathtt{aux}', \mathtt{s}_0'', \mathtt{s}_1'''', \mathtt{r}), \varphi \rangle$$

where φ includes, among others, the constraint $this{=}s_1''''{+}1$ which states that the list that $this$ points to is longer by one than the list that s_1'''' points to (s_1'''' corresponds to $this.next$). The meaning of the above relations is explained in Section 2. Note that the call to the constructor Cons.<init> *is ignored for simplicity.* □

4.3 Heap Space Cost Relations

In order to define our heap space cost analysis, we start by defining a cost model which defines the cost of memory allocation instructions (e.g., new, newarray and anewarray) as the the number of heap (memory) units they consume. The remaining bytecode instructions do not add any cost.

DEFINITION 4.5 (cost model for heap space). *We define a cost model $\mathcal{M}_{heap}$ which takes a bytecode instruction* bc *and returns a positive expression as follows:*

$$\mathcal{M}_{heap}(\mathtt{bc}) = \begin{cases} size(\mathtt{Class}) & \textit{if bc=}\mathtt{new(Class,\_)} \\ S_{\mathtt{PrimType}} * \mathtt{L} & \textit{if bc=}\mathtt{newarray(PrimType,L,\_)} \\ S_{\mathtt{ref}} * \mathtt{L} & \textit{if bc=}\mathtt{anewarray(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ denote, respectively, the heap consumption of primitive types and references. Function size is defined as follows:

$$size(O) = \begin{cases} \sum_{F \in \mathtt{Class.field}} size(type(F)) & \textit{if O=}\mathtt{Class} \\ S_{\mathtt{PrimType}} & \textit{if O is a primitive type} \\ S_{\mathtt{ref}} & \textit{if O is a reference type} \end{cases}$$

where the type of a field in a Class *(i.e.,* Class.field*) can be either primitive or reference.* □

In Java bytecode, types are classified into primitive (its size is represented by $S_{\mathtt{PrimType}}$ in our model) and reference types ($S_{\mathtt{ref}}$). In a particular assessment, one has to set the concrete values for $S_{\mathtt{PrimType}}$ and $S_{\mathtt{ref}}$ of the JVM implementation.

For each rule in the recursive representation of the program and its corresponding size relation, the analysis generates the cost equations which define the heap consumption of executing the block (or possibly a method call) by relying on the above cost model. A *heap space cost relation* is defined as the set of cost equations for each block of the bytecode (or rule in the recursive representation).

DEFINITION 4.6 (heap space cost relation). *Consider a rule $\mathcal{R}$ of the form $\mathtt{p}(\bar{\mathtt{x}}) \leftarrow \mathtt{G_p}, \mathtt{B}, (\mathtt{q_1}(\bar{\mathtt{y}}); \cdots; \mathtt{q_n}(\bar{\mathtt{y}}))$ and let the linear constraints φ be a conjunction of all call-to size-relations within the rule. The* heap space cost *equations for $\mathcal{R}$ are generated as follows:*

$$C_p(\bar{x}) = \sum_{b \in bytecode(B)} \mathcal{M}_{heap}(b) + \sum_{r(\bar{z}) \in calls(B)} + C_r(\bar{z}) + C_{p\_cont}(\bar{y}) \quad \varphi$$
$$C_{p\_cont}(\bar{y}) = C_{q_1}(\bar{y}) \quad G_{q_1}$$
$$\ldots$$
$$C_{p\_cont}(\bar{y}) = C_{q_n}(\bar{y}) \quad G_{q_n}$$

where $\mathtt{G_{q_i}}$ is the guard of $\mathtt{q_i}$. The heap space cost relation associated to the recursive representation of a method is defined as the set of cost equations for its blocks. □

When the rule has multiple *continuation*s, it is transformed into several equations. We specify the cost of each continuation in a separate equation because the guards for determining the alternative path $\mathtt{q_i}$ that the execution will take (with $i = 1, \ldots, n$) are only known at the end of the execution of the bytecode B; thus, they cannot be evaluated before B is executed. The guards appear also decorating the equations. In the implementation, when a rule has only one continuation, it gives rise to a single equation which contains the size relation φ as an attachment.

Java source code

```
abstract class Data{
  abstract public Data copy();
}
class Polynomial extends Data{
  private int deg;
  private int[] coefs;
  public Polynomial() {
    coefs = new int[11];}
  public Data copy() {
    Polynomial aux = new Polynomial();
    aux.deg = deg;
    for (int i=0;i<=deg && i<=10;i++)
      aux.coefs[i] = coefs[i];
    return aux;}}
```

```
class Vector3D extends Data{
  private int x;
  private int y;
  private int z;
  public Vector3D(int x,int y,int z) {
    this.x = x;
    this.y = y;
    this.z = z;}
  public Data copy() {
    return new Vector3D(x,y,z);
  }
}
```

```
class Results{
  Data[] rs;
  public Results() {
    rs = new Data[25];
  }
```

```
public Results copy() {
  Results aux = new Results();
  for (int i = 0;i < 25;i++)
    aux.rs[i] = rs[i].copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = S_{ref} + 25{*}S_{ref} + C_0(a,0)$		
$C_0(a,i) = \underbrace{S_{int} + S_{ref}}_{size(Polyn)} + 11{*}S_{int} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Polyn \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = \underbrace{3{*}S_{int}}_{size(Vect3D)} + C_0(a,j)$	$\langle \hat{a}.rs[i] \in Vect3D \rangle$	$\{i{<}25,\ j = i{+}1\}$
$C_0(a,i) = 0$		$\{i{>=}25\}$

Figure 2. Constant heap space example

EXAMPLE 4.7. *The heap space cost equations generated for the rule* $copy_0^{Cons}$ *of Ex. 4.2 and the size relation of Ex. 4.4 are (see Sect. 2):*

$$C_0^{Cons}(this, aux) = size(Cons) + CC_0^{Cons}(this, aux', s_0'', s_1'''')\{this = s_1'''' + 1, \ldots\}$$

$$CC_0^{Cons}(this, aux', s_0'', s_1'''') = \begin{cases} C_1^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Nil \\ C_2^{Cons}(this, aux', s_0'', s_1'''') & \hat{s}_1'''' \in Cons \end{cases}$$

The cost of $Block_0^{Cons}$ *is captured by* C_0^{Cons}*, among all bytecode instructions in* $Block_0^{Cons}$*, we count only the creation of the object of class* `Cons`*. The continuation of* $Block_0^{Cons}$ *is captured in the relation* CC_0^{Cons}*, where depending on the type of the object* s_1''''*, we choose between two* mutually exclusive *equations* C_1^{Cons} *or* C_2^{Cons}*.* □

In addition, the analyzer performs a slicing step, which aims at removing variables that do not affect the cost. And also tries to simplify the equations as much as possible by applying unfolding steps. These steps lead to simpler cost relations. Due to lack of space, during the rest of the paper we will apply them without giving details on how they were performed.

5. Example Applications of Heap Space Analysis

In this section, we show the most salient features of our heap space analysis by means of a series of examples. All examples are written in object-oriented style and make intensive use of the heap. We intend to illustrate how our analysis is able to deal with standard data-structures like lists, trees and arrays with several dimensions as well as with multiple inheritance, class fields, virtual invocation, etc. We show examples which present heap usage which depends proportionally to the *data size*, namely in some cases it depends on class fields while in another one on the input arguments. An interesting point is that heap consumption is, in the different examples, constant, linear, polynomial or exponentially proportional to the data sizes.

For each example, we show the Java source code and its heap space cost relation. Each relation consists of three parts: the equations, the guards and the size relations. The applicability conditions of each equation are defined by the guards and the size relations. Guards usually provide non-numeric conditions while size relations provide conditions on the sizes of the corresponding variables. In addition, size relations describe how the data changes when the control moves from one to another part of the program. Since our

Java source code

```
abstract class List{
  abstract public List copy();
}
class Nil extends List{
  public List copy() {
    return this;
}
```

```
class Cons extends List
  private Data elem;
  private List next;
  public List copy() {
    Cons aux = new Cons();
    aux.elem = this.elem.copy();
    aux.next = this.next.copy();
    return aux;}
```

Heap space cost equations

Equation	Guard	Size rels.
$C_{copy}(a) = \overbrace{2{*}S_{ref}}^{size(Cons)} + \overbrace{S_{int}+S_{ref}+11{*}S_{int}}^{this.elem.copy()}$	$\langle \hat{a}.elem \in Polyn \wedge \hat{a}.next \in Nil \rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+S_{int}+S_{ref} +11{*}S_{int}+C_{copy}(b)$	$\langle \hat{a}.elem \in Polyn \wedge \hat{a}.next \in Cons \rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$
$C_{copy}(a) = \underbrace{2{*}S_{ref}}_{size(Cons)} + \underbrace{3{*}S_{int}}_{this.elem.copy()}$	$\langle \hat{a}.elem \in Vect3D \wedge \hat{a}.next \in Nil \rangle$	$\{a=2\}$
$C_{copy}(a) = 2{*}S_{ref}+3{*}S_{int}+C_{copy}(b)$	$\langle \hat{a}.elem \in Vect3D \wedge \hat{a}.next \in Cons \rangle$	$\left\{ \begin{array}{l} a{\geq}3, b{\geq}2 \\ a{>}b \end{array} \right\}$

Figure 3. Generic list example

system only deals with integer primitive types, we use the cost model presented in Sect. 4.3 with the constants S_{int} and S_{ref} to denote the basic sizes for integers and reference types, respectively. Also note that we provide the Java source code instead of the bytecode just for clarity and space limitations. The analyzer works directly on the bytecode which can be found in the appendix.

5.1 Constant Heap Space Usage

In the first example we consider a method with constant heap space usage, i.e., its heap consumption does not depend on any input argument. Fig. 2 shows both the source code and the heap space cost equations generated by the analyzer. The program implements a data hierarchy which will be used throughout the section. It consists of an abstract class, *Data* and two subclasses, *Polynomial* and *Vector3D*. The class *Polynomial* defines a polynomial expression of degree up to 10 with integer coefficients, the coefficients are stored in the array field *coefs* and the degree in the integer field *deg*. Its *copy* method returns a deep copy of the corresponding polynomial by creating a new array of 11 integers and copying the first $deg{+}1$ original coefficients. The class *Vector3D* represents an integer vector with 3 dimensions. The class *Results* stores 25 objects of type *Data*, which in execution time will be *Polynomial* or *Vector3D* objects. Its *copy* method produces a deep copy of the whole structure where each of the 25 elements is copied by its corresponding *copy* method (hence dynamically resolved).

The cost equations generated by the analyzer for the method *Results.copy* are shown in Fig. 2 (at the bottom left). The first equation $C_{copy}(a)$ defines the heap consumption of the method in terms of its first argument a which corresponds to the abstraction of its *this* reference variable (i.e., its size). It counts the heap space allocated for the creation of an object of type *Results*, namely S_{ref}; the space allocated by its constructor, namely $25{*}S_{ref}$; and the space allocated when executing the loop. The heap space allocated by the loop is captured by C_0 and it depends on the type of the object at the current position of the array (which is specified in the guards by checking the class of $\hat{a}.rs[i]$) such that the call to its corresponding *copy* method contributes $S_{int}+S_{ref}{+}11{*}S_{int}$ if it is an instance of *Polynomial* and $3{*}S_{int}$ if it is an instance of *Vector3D*.

As already mentioned, a further issue is how to automatically infer *closed form* solutions (i.e., without recurrences) from the generated cost relations. In our examples, we can directly apply the method of [1] to compute an upper bound in closed form. However, we will not go into details of this process as it is not a concern of this paper and we will simply show the asymptotic complexity that can be directly obtained from such upper bounds. We can observe from the equations that the asymptotic complexity is $O(1)$, as equation C_{copy} is a constant plus C_0, and C_0 is called a constant number of times (in this case 25 times). By assuming that $S_{int}=4$ and $S_{ref}=4$, we can obtain the following upper bound $C_{copy}=4+25*4+25*52=1404$.

5.2 Bounds Proportional to the Input Data Size

For the second example, we consider a generic data structure of type *List*. Both the source code and the heap space cost equations obtained by our analyzer are depicted in Fig. 3. The list is implemented taking advantage of the polymorphism as in the style of the example in Sect. 1, but in this case the elements of the list are objects extending from *Data*

Java source code

```
class Score{
  private int gt1, gt2;
  public Score() {
    gt1 = 0;
    gt2 = 0;
  }
class Scoreboard{
  private Score[][][] scores;
public Scoreboard(int a,int b) {
  scores = new Score[a][][];
  for (int i = 1;i <= a;i++) {
    scores[i-1] = new Score[i][];
    for (int j = 0;j < (i-1);j++) {
      scores[i-1][j] = new Score[b];
      for (int k = 0;k < b;k++)
        scores[i-1][j][k] = new Score();}}}}
```

Heap space cost equations

Equation			Guard	Size rels.
$C_{<init>}(a,b)$	=	$a{*}S_{ref} + C_1(a,b,1)$		
$C_1(a,b,i)$	=	$i{*}S_{ref} + C_2(b,i,0) + C_1(a,b,d)$		$\{i{\leq}a,\ d = i{+}1\}$
$C_1(a,b,i)$	=	0		$\{i > a\}$
$C_2(b,i,j)$	=	$b{*}S_{ref} + C_3(b,0) + C_2(b,i,d)$		$\{j{<}(i{-}1),\ d = j{+}1\}$
$C_2(b,i,j)$	=	0		$\{j \geq (i{-}1)\}$
$C_3(b,k)$	=	$2{*}S_{int} + C_3(b,c)$		$\{k{<}b,\ c = k{+}1\}$
$C_3(b,k)$	=	0		$\{k \geq b\}$

Figure 4. Multi-dimensional arrays example

(see the classes in Fig. 2) rather than integer primitive types. The *List.copy* method returns a deep copy of the list which, in addition to copying the whole list structure, it copies each element by using the corresponding *Data.copy* method (resolved at execution time).

At the bottom of Fig. 3, we show the heap space cost equations our analyzer generates for the method *List.copy* of class *Cons*. The equation $C_{copy}(a)$ defines the heap consumption of the whole method in terms of its first argument a which represents the size of its *this* reference variable. There are four equations for C_{copy}, two of them (the second and the fourth one) are recursive and correspond to the case in which the rest of the list is not empty, i.e., $\hat{a}.next \in Cons$. Note that, in such recursive equations, the size analysis is able to infer the constraint $a > b$, thus ensuring that recursive calls are made with a strictly decreasing value. The other two equations are constant and correspond to the base case (i.e. the rest of the list is empty). This is abstracted in the size relations with the constraint $a = 2$. Note that the heap usage depends on whether we invoke the *copy* method of a *Polynomial* or a *Vector3D* object. By considering the worst cases for all equations, we can infer the upper bound $C_{copy}(a){\leq}(5{*}S_{ref}{+}15{*}S_{int}){*}a \equiv O(a)$ which describes a heap consumption linear in a, the size of the list.

5.3 Multi-Dimensional Arrays

Let us consider the example in Fig. 4. The class *Scoreboard* is instrumental to show how our heap space analysis deals with complex multi-dimensional array creation. The class has a 3-dimensional array field. The constructor takes two integers a and b and creates an array such that: the first dimension is a; the second dimension ranges from 1 to a; and the third dimension is b. Each array entry $scores[i][j][k]$ stores an object of type *Score*.

At the bottom of Fig. 4 we can see the heap space cost equations generated by the analyzer for the constructor of class *Scoreboard*. The equation $C_{<init>}(a,b)$ represents the heap space consumption of the constructor where a and b correspond to the size of its input parameters. It counts the heap consumed by constructing the first array dimension, $a{*}S_{ref}$, plus the heap consumption when executing the outermost loop which is represented by the call $C_1(a,b,1)$. The heap consumption modeled by C_1 includes the amount of heap allocated for the second array dimension in each iteration, $i{*}S_{ref}$, and the consumption of executing the middle loop which is represented by the call $C_2(b,i,0)$. Note that size analysis infers that within C_1, the value of i increases by 1 at each iteration ($d{=}i{+}1$) until it converges to a ($i{\leq}a$). The equation C_2 defines the heap consumption of the middle loop, which includes the heap allocated for the third array dimension, $b{*}S_{ref}$, plus the consumption of executing the innermost loop which is represented by the call $C_3(b,0)$. Finally, C_3 models the heap space required for creating $b{-}k$ *Score* objects by the innermost loop. In this case, we can infer the upper bound $C_{<init>}(a,b) \leq (((2{*}S_{int}{*}b) + b{*}S_{ref}){*}a + a{*}S_{ref}){*}a + a{*}S_{ref} \equiv O(b{*}a^2)$.

5.4 Complex Data Structures

For the last example, let us consider a more complex tree-like data structure which is depicted in Fig. 5. The class *MultiBST* implements a binary search tree data structure where each node has an object of type List (from Fig. 3) and two successors of type *MultiBST* which correspond to the right and left branches of the tree. The constructor method creates an empty tree whose *data* field is initialized to an empty list, i.e., an instance of class *Nil*. The *copy* method performs a deep copy of the whole tree by relying on the *copy* method of class *List*.

Java source code

```
class BST {
  private List data;
  private MultiBST lc;
  private MultiBST rc;
  public MultiBST() {
    data = new Nil();
    lc = null; rc = null; }
```

```
public MultiBST copy() {
  MultiBST aux = new MultiBST();
  aux.data = data.copy();
  if (l==null) aux.lc=null; else aux.lc=lc.copy();
  if (r==null) aux.rc=null; else aux.rc=rc.copy();
  return aux;}}
```

Heap space cost equations

Equation	Guard	Size rels.
$C(a) = 3{*}S_{ref} + D(d)$	$\langle \hat{a}.lc = null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc = null \rangle$	$\{a{>}0, a{>}d, a{>}l\}$
$C(a) = 3{*}S_{ref} + D(d) + C(r)$	$\langle \hat{a}.lc = null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}r\}$
$C(a) = 3{*}S_{ref} + D(d) + C(l) + C(r)$	$\langle \hat{a}.lc \neq null, \hat{a}.rc \neq null \rangle$	$\{a{>}0, a{>}d, a{>}l, a{>}r\}$

Figure 5. Multi binary search tree example

The heap space cost equations generated by the analyzer for the method *MultiBST.copy* are depicted in Fig. 5 (at the bottom). The equations defining $C(a)$ represent the heap space usage of the whole method in terms of the parameter a, which corresponds to the maximal path-length in the tree. There are four cases which correspond to the different possible values for the left and right branches (equal or different from null). Consider for example the last equation: $3 * S_{ref}$ is the heap allocated by the new instruction; $D(d)$ is the heap consumption for copying an object of type *List* which corresponds to C_{copy} from Fig. 3; $C(l)$ and $C(r)$ correspond to the heap consumption of copying the left and right branches respectively. From the cost relation, we infer the upper bound $C(a){\leq}(3{*}S_{ref}{+}D(a)){*}2^a$ where $D(a)$ corresponds to the cost of copying the *data* field (see Sec. 5.2).

6. Active Heap Space with Garbage Collection

One of the safety principles in the Java language is ensured by the use of a garbage collector which avoids errors by the programmer related to deallocation of objects from the heap. The aim of this section is to furnish the heap usage cost relations with *safe annotations* which mark the heap space that will be deallocated by the garbage collector upon exit from the corresponding method. The annotations are then used to infer heap space upper bounds for methods upon exit.

In order to generate such annotations, we rely on the use of *escape analysis* (see, e.g., [8, 15]). Essentially, we assume that the heap allocation instructions `new`, `newarray` and `anewarray` have been respectively transformed by new instructions `new_gc`, `newarray_gc` and `anewarray_gc` as long as it is guaranteed that the lifetime of the corresponding allocated heap space does not exceed the instruction's static scope. In this case, the heap space can be safely deallocated upon exit from the corresponding method. This preprocessing transformation can be done in a straightforward way by using the information inferred by escape analysis. In the following, we refer by *transformed* bytecode instructions to the above transformation performed on the heap allocation instructions. Also, we use $gc(H)$ to denote that the heap space H will safely be garbage collected upon exit from the corresponding method (according to escape analysis) and $ngc(H)$ to denote that it might not be garbage collected.

DEFINITION 6.1. *We define a cost model for heap space with garbage collection which takes a* transformed *bytecode instruction* bc *and returns a positive symbolic expression as:*

$$\mathcal{M}^{gc}_{heap}(\mathsf{bc}) = \begin{cases} ngc(size(\texttt{Class})) & \textit{if bc=}\texttt{new(Class,\_)} \\ gc(size(\texttt{Class})) & \textit{if bc=}\texttt{new\_gc(Class,\_)} \\ ngc(S_{\texttt{PrimType}} * \texttt{L}) & \textit{if bc=}\texttt{newarray(PrimType,L,\_)} \\ gc(S_{\texttt{PrimType}} * \texttt{L}) & \textit{if bc=}\texttt{newarray\_gc(PrimType,L,\_)} \\ ngc(S_{\texttt{ref}} * \texttt{L}) & \textit{if bc=}\texttt{anewarray(Class,L,\_)} \\ gc(S_{\texttt{ref}} * \texttt{L}) & \textit{if bc=}\texttt{anewarray\_gc(Class,L,\_)} \\ 0 & \textit{otherwise} \end{cases}$$

where $S_{\texttt{PrimType}}$, $S_{\texttt{ref}}$ *and* $size()$ *are as in Def. 4.5.* □

The above cost model returns a *symbolic* positive expression which contains the annotations gc and ngc as described above. Therefore, when generating the heap space cost relations as described in Def. 4.6 w.r.t. $\mathcal{M}^{gc}_{heap}$, the cost relations will be of the following form:

$$C(\bar{x}) = gc(H_{gc}) + ngc(H_{ngc}) + \sum C_r(\bar{z}) \quad \varphi$$

where we assume that all symbolic expressions wrapped by gc (resp. by ngc) are grouped together within each cost equation and denote the total heap space H_{gc} that will be garbage collected (resp. H_{ngc} which might not be) after the application of such equation.

EXAMPLE 6.2. *Suppose we add the following methods*

```
abstract List map(Func o);                    // List
List map(Func o) { return this; }             // Nil
List map(Func o) {                            // Cons
    List tail = this.next.map(o);
    Cons head = new Cons();
    head.next = tail;
    head.elem = o.f(new Integer(this.elem));
    return head;
}
```

respectively to the classes List*,* Nil *and* Cons *which are depicted in Fig. 1. The method* map *clones the corresponding list structure, but the value of the field* elem *in the clone is the result of applying the method* o.f *on the corresponding value in the cloned list. Note that the method* o.f*, takes as input an object of type* Integer*, therefore* this.elem *(which is of type* int*) is first converted to* Integer *by creating a temporary corresponding* Integer *object. For simplicity, we do not give a specific definition for* Func*, but we assume that its method* f *(which is called using* o.f*) does not allocate any heap space and that it returns a value of type* int*. Using* escape analysis*, the creation of the temporary* Integer *object can be annotated as local to* map*, therefore we replace the corresponding* `new` *instruction by* `new_gc`*. Assuming that the size of an* Integer *object is 4 bytes, and using the cost model of Def. 6.1, we obtain the following cost equations:*

Equation	*Size relations*
$C^{Nil}_{map}(a) = 0$	$\{a{=}1\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8)$	$\{a{=}2\}$
$C^{Cons}_{map}(a) = gc(4) + ngc(8) + C^{Cons}_{map}(b)$	$\{a{\geq}3, b{\geq}1, a{=}b{+}1\}$

The symbolic expression $gc(4)$ *in the above equations corresponds to the heap space allocated for the temporary* Integer *object which can be garbage collected upon exit from* map*, and* $ngc(8)$ *corresponds to the heap space allocated for the* Cons *object. As before,* a *corresponds to the size of the* this *reference variable (i.e., the list length) and* b *to* this.next. □

Using the refined cost relations we can infer different information about the heap space usage depending on the interpretation given to the gc and ngc annotations. Let us first consider the following definitions:

$$\boxed{\forall\, H,\, gc(H) = 0 \text{ and } ngc(H) = H} \tag{1}$$

where we do not count the heap space that will be deallocated upon exit from the corresponding method. By applying Eq. (1) to a cost relation C_m of a method m, we can infer an upper bound U^{gc}_m of the *active heap space* upon the exit from m, i.e., the heap space consumed by m which might not be deallocated upon exit. In this setting, for the cost relations of Ex. 6.2 we infer the closed form $U^{gc}_{map} \equiv C^{Cons}_{map}(a) = 8 * (a-1)$. It is important to note that, in general, such upper bound does not ensure that the heap space required for executing m does not exceed U^{gc}_m, i.e., it is not an upper bound of the heap usage *during* the execution of m but rather only *after* its execution. Actually, in this simple example, we can observe already that during the execution of the method *map*, if all objects are heap allocated, we need more than $8 * (a-1)$ heap units (as the objects of type *Integer* will be heap allocated and they are not accounted in the upper bound). However, one of the applications of escape analysis is to determine which objects can be stack allocated instead of heap allocated in order to avoid invoking the garbage collector which is time consuming [8]. For instance, in the above example, the objects of type *Integer* can be safely stack allocated. When this stack allocation optimization is performed, then U^{gc}_m is indeed an upper bound for the heap space required to execute m.

In order to infer upper bounds for the heap space required during the execution of m, we define gc and ngc as follows:

$$\boxed{\forall\, H,\, gc(H) = H \text{ and } ngc(H) = H} \tag{2}$$

In this case, we obtain the same cost relations as in Def. 4.6 which correspond to the worst case heap usage in which we do not discount any deallocation by the garbage collector. In this setting, for the cost relation of Ex. 6.2 we infer the closed form $U_{map}(a) \equiv C^{Cons}_{map}(a) = 12 * (a-1)$.

Analysis for finding upper bounds on the memory high-watermark cannot be directly done using cost relations as introducing decrements in the equations requires computing lower bounds. As a further issue, the *active heap space* upper bound, U^{gc}_m, can be used to improve the accuracy of the upper bound on the heap space required for executing a sequence of method calls. For example, an upper bound of the heap space required for executing a method m_1 and upon its return immediately executing a method m_2 can be approximated by $max(U_{m_1}, U^{gc}_{m_1} + U_{m_2})$ which is more precise than taking $U_{m_1} + U_{m_2}$ as it takes into account that after executing m_1 we can apply garbage collection and only then executing m_2. This idea is the basis for a post-processing that could be done on the program in order to obtain more accurate upper bounds on the heap usage at a *program point* level. This is a subject of ongoing research.

7. Experiments

In order to assess the practicality of our heap space analysis, we have implemented a prototype inter-procedural analyzer in `Ciao` [10] as an extension of the one in [3]. We still have not incorporated an escape analysis in our implementation and hence the upper bounds inferred correspond to those generated using Eq. (2) of Sect. 6. The experiments have been performed on an Intel P4 Xeon 2 GHz with 4 GB of RAM, running GNU Linux FC-2, 2.6.9. Table 1 shows the run-times of the different phases of the heap space analysis process. The name of the main class to be analyzed is given in the first column, *Benchmark*, and its size (the sum of all its *class* file sizes) in KBytes is given in the second column, *Size*. Columns 3-6 shows the runtime of the different phases in milliseconds, they are computed as the arithmetic mean of five runs: *RR* is the time for obtaining the recursive representation (building CFG, eliminating stack elements, etc., as outlined in Sec. 4.1); *Size An.* is the time for the abstract-interpretation based size analysis for computing size relations; *Cost* is the time taken for building the heap space cost relations for the different blocks and representing them in a simplified form; and *Total* shows the total times of the whole analysis process. In the last column, *Complexity*, we depict the asymptotic complexity of the (worst-case) heap space cost obtained from the cost relations.

Benchmark	Size	RR	Size An.	Cost	Total	Complexity	
ListInt	0.86	24	53	7	83	$O(n)$	$n \equiv$ list length
Results	1.31	83	275	15	374	$O(1)$	–
BSTInt	0.48	37	113	5	156	$O(2^n)$	$n \equiv$ tree depth
List	1.79	71	207	16	293	$O(n)$	$n \equiv$ list length
Queue	1.93	219	570	24	813	$O(n)$	$n \equiv$ queue length
Stack	1.38	89	643	17	749	$O(n)$	$n \equiv$ stack length
BST	1.43	97	238	14	349	$O(2^n)$	$n \equiv$ tree depth
Scoreboard	0.65	280	1539	12	1830	$O(a^2 * b)$	$\{a, b\} \equiv$ input args.
MultiBST	2.35	166	510	34	709	$O(n * 2^n)$	$n \equiv$ tree depth

Table 1. Measured time (in ms) of the different phases of cost analysis

Regarding the benchmarks we have used, on one hand, we have benchmarks implementing some classic data structures using an object-oriented programming style, which expose the analyzer's ability in handling such classical data structures as well as sophisticated object-oriented programming features. In particular, *ListInt*, *List*, *Queue*, *Stack*, *BSTInt*, *BST* and *MultiBST* implement respectively integer and generic lists, generic queues, generic stacks, integer and generic binary search trees which allow data repetitions. On the other hand, we have some benchmarks which expose more particular issues of heap space analysis, such as *Results* which has constant heap space usage and *Scoreboard* which presents a multidimensional arrays creation. For all benchmarks, we have analyzed the corresponding *copy* method which performs a deep copy of the corresponding structure.

We can observe in the table that computing size relations is the most expensive step as it requires a global analysis of the program, whereas *RR* and *Cost* basically involve a single pass on the code. Our prototype implementation supports the full instructions set of sequential Java bytecode, however, it is still preliminary, and there is plenty of room for optimization, mainly in the size analysis phase, which in addition assumes the absence of cyclic data structures, which can be verified using the non-cyclicity analysis [23].

8. Conclusions and Related Work

We have presented an automatic analysis of heap usage for Java bytecode, based on generating at compile-time cost relations which define the heap space consumption of an input bytecode program. By means of a series of examples which allocate lists, trees, trees of lists, arrays, etc. in the heap, we have shown that our analysis is able to infer non-trivial bounds for them (including polynomial and exponential complexities). We believe that the experiments we have presented show that our analysis improves the state of the practice in heap space analysis of Java bytecode.

Related work in heap space analysis includes advanced techniques developed in functional programming, mainly based on type systems with resource annotations (see, e.g., [24, 17, 25, 19]) and, hence, they are quite different technically to ours. But heap space analysis is comparatively less developed for low-level languages such as Java bytecode. A notable exception is the work in [11], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [5, 7], which focuses on building a proof-carrying code [22] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. The analysis is developed for a functional language which then compiles to a (subset of) Java bytecode and it is restricted to linear bounds. In [6] the Bytecode Specification Language is used to annotate Java bytecode programs with memory consumption behaviour and policies, and then verification tools are used to verify those policies.

For Java-like languages, the work of [18] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortised analysis (hence it is technically quite different to our work) and, unlike us, they do not present an inference method for heap consumption. On the other hand, the work of [9] deals also with Java source code, it is able to infer polynomial complexity though it does not handle recursion.

Some works consider explicit deallocation of objects by decreasing the cost by the size of the deallocated object (see, e.g., [18, 17]). This approach is interesting when one wants to observe the heap consumption at certain program points. However, it cannot be directly incorporated in our cost relations because they are intended to provide a *global* upper bound of a method's execution. Naturally, it should happen that allocated objects are correctly deallocated and hence our cost relations would provide zero as (global) upper bound. Other work which considers cost with garbage collection is [24]. Unlike ours, it is developed for pure functional programs where the garbage collection behaviour is easier to predict as programs do not have assignments.

In the future, we want to extend our work in several directions. On the practical side, we want to incorporate an escape

analysis to transform the bytecode as outlined in Sect. 6. Regarding scalability, it is a question of performance vs. precision trade-off and depends much on the underlying abstract domain used by the size analysis. We believe our analysis would scale without sacrificing precision if an efficient domain like octagons is used together with [1]. On the theoretical side, we plan to adapt our analysis to infer upper bounds on the heap usage at given program points in the presence of garbage collection. We also would like to develop an analysis which infers upper bounds on the call stack usage.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Cost Equation Systems. *Submitted*, 2007.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *Proc. of BYTECODE'07*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[4] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, number 3452 in LNAI, pages 380–397. Springer-Verlag, 2005.

[5] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, number 3362 in LNCS. Springer, 2005.

[6] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[7] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[8] Bruno Blanchet. Escape Analysis for Java$^{\text{tm}}$: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[9] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[10] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[11] D. Cachera, D. Pichardie T. Jensen, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[12] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proc. of ESOP'05*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.

[13] W. Chin, H. Nguyen, S. Qin, and M. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, LNCS 3672, pages 70–86. Springer, 2005.

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

[15] Patricia M. Hill and Fausto Spoto. Deriving Escape Analysis by Abstract Interpretation. *Higher-Order and Symbolic Computation*, (19):415–463, 2006.

[16] M. Hofmann. Certification of Memory Usage. In *Theoretical Computer Science, 8th Italian Conference, ICTCS*, volume 2841 of *Lecture Notes in Computer Science*, page 21. Springer, 2003.

[17] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *30th ACM Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM Press, 2003.

[18] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[19] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

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

[21] Patricia M.Hill, Etienne Payet, and Fausto Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*, 2006.

[22] G. Necula. Proof-Carrying Code. In *POPL'97*. ACM Press, 1997.

[23] S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In E. A. Emerson and K. S. Namjoshi, editors, *Proc. of the 7th workshop on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 95–110, Charleston, SC, USA, January 2006. Springer-Verlag.

[24] L. Unnikrishnan, S. Stoller, and Y. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, Lecture Notes in Computer Science, pages 70–85. Springer, 2003.

[25] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.