Science of Computer Programming 78 (2013) 1427–1448

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Heap space analysis for garbage collected languages

Elvira Albert, Samir Genaim *, Miguel Gómez-Zamalloa

DSIC, Complutense University of Madrid (UCM), Spain

ARTICLE INFO

Article history:
Received 1 March 2011
Received in revised form 29 September 2012
Accepted 1 October 2012
Available online 12 October 2012

Keywords:
Static analysis
Live heap space analysis
Peak memory consumption
Low-level languages
Java bytecode
Garbage collection

ABSTRACT

Accurately predicting the dynamic memory consumption (or *heap space*) of programs can be critical during software development. It is well-known that garbage collection (GC) complicates such problem. The *peak heap consumption* of a program is the maximum size of the data on the heap during its execution, i.e., the minimum amount of heap space needed to safely run the program. Existing heap space analyses either do not take deallocation into account or adopt specific models of garbage collectors which do not necessarily correspond to the actual memory usage. This paper presents a novel static analysis for garbage-collected imperative languages that infers accurate *upper bounds* on the peak heap usage, including exponential, logarithmic and polynomial bounds. A unique characteristic of the analysis is that it is *parametric* on the notion of *object lifetime*, i.e., on when objects become *collectible*.

1. Introduction

Predicting the dynamic memory (heap) required to run a program is crucial in many contexts such as in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. Due in part to the difficulty of predicting the heap usage of programs, real-time and embedded software typically uses only statically allocated data, which is known to have disadvantages. It is also widely recognized that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime.

On the other hand, garbage collection (GC) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes difficult to predict the amount of memory required to run a program. A first approximation to this problem is to simply ignore the GC and infer bounds on the *total heap consumption*, i.e., the *accumulated* amount of memory dynamically allocated by a program. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual heap consumption. In this article, we propose a novel *peak heap space analysis*, also known as *live heap space analysis*, which aims at approximating the maximum size of the data on the heap during a program's execution, which provides a much tighter estimation.

Whereas analyzing the total heap consumption needs to observe the consumption at the *final* state only, peak heap consumption analysis has to reason on the heap consumption at *all program states* along the execution. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [37] will be directly applicable only to infer the

* Corresponding author.
E-mail addresses: elvira@sip.ucm.es (E. Albert), genaim@gmail.com, samir.genaim@fdi.ucm.es (S. Genaim), mzamalloa@fdi.ucm.es (M. Gómez-Zamalloa).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.10.008

total memory allocation. Intuitively, given a program, this approach produces a *cost relation system* (*CR* for short) which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the *CR*. This approach leads to very accurate cost bounds (including polynomial, logarithmic, exponential consumption bounds) and, besides, it can be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, it is not suitable to infer peak heap consumption because the heap usage is not an accumulative resource of a program's execution as *CRs* capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate *CRs*, peak heap consumption analysis is currently restricted to polynomial bounds [19], non-recursive methods [10], to linear bounds dealing with recursion [14] or are not fully automatic [21].

When considering GC, several techniques exist which differ on:

(1) *what* can be collected, i.e., the *lifetime* of objects;
(2) *when* GC is performed.

As regards (1), a *GC strategy* classifies objects in the heap into two categories: those which are collectible and those which are not. Most types of garbage collectors determine *unreachable* objects as collectible, i.e., they eliminate those objects to which there is no variable in the program environment pointing directly or indirectly. The more precise alternative is to rely on the notion of *liveness*. An object is said to be *not live* (or *dead*) at some state if it is not used from that point on during the execution.

As regards (2), in this paper, we consider several possibilities. One is *scope-based* GC in which deallocation of unreachable objects takes place on method's return and only those objects created during the method's execution can be freed. The scope assumption is motivated by the notion of stack reference liveness [31] in Java-like languages, according to which some objects which the local variables point to, become unreachable upon exit from methods, i.e., when the corresponding call stack frames are removed. Another possibility is the so-called *ideal* GC in which objects are collected as soon as they become collectible. The third one assumes a given limit on the heap, and applies GC only when we are about to exceed this limit.

The language we consider to develop our analysis is an imperative *bytecode*. Programming languages that are compiled to *bytecode* and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode [25] and .NET. The execution model based on virtual machines has the important advantage when compared to classical machine code that bytecode is *platform-independent*, i.e., the same compiled code can be run on multiple platforms.

In this article, we present a general framework to infer accurate bounds on the peak heap consumption of bytecode programs which improves the state-of-the-art in that:

- it is not restricted to any complexity class and deals with all bytecode language features including recursion,
- it is parametric w.r.t the *lifetime* of objects and,
- it can be instantiated with different GC strategies, e.g., the scope-based and ideal GC discussed above.

1.1. Motivating example

Let us motivate our work on a contrived example which is depicted in Fig. 1 (to the left). To the right, we show an intermediate representation of the program that will be explained later. Because the program has simple (constant) memory consumption, it is useful to describe intuitively the differences among the different approximations to memory consumption and, later, to explain the main technical parts of the paper.

Example 1. In Fig. 1 (to the bottom) we provide four possible approximations inferred by our analysis for the memory consumption of executing method m_1, where the notation s(X) means the memory required to hold an instance of class X.

First, we consider a scope-based garbage collector in which object lifetimes are inferred by an *escape* analysis. In this case, we can take advantage of the knowledge that at ④ (i.e., upon exit from m_2) the object to which "c" refers can be freed, i.e., it does not *escape* from the method. Hence, the *upper bound* (UB) S is obtained. The important point is that s(A) and s(B) are always accumulated, plus the largest of the consumption of m_2 (i.e., s(C) + s(E)) and the memory *escaped* from m_2 (i.e., s(E)) plus the continuation (i.e., s(D)).

As another instance, we consider a reachability-based GC but without the assumption of being scope-based, rather we assume an ideal GC. Then, our method is able to obtain the UB R in Fig. 1. This is due to the fact that the object to which "a.f" points becomes unreachable at program point ③, the object to which "c" points becomes unreachable upon exit from m_2, and the object created immediately before ① becomes unreachable at ④. We can observe that this information is reflected in R by taking the maximum between: the consumption up to the first allocation instruction in m_2; the consumption up to the end of m_2 taking into account that the object to which "a.f" points becomes unreachable, plus the consumption until the end of m_1 taking into account that both the object pointed by "a.f" and the object created immediately before ① become unreachable.

As the third instance, we consider the combination of an ideal garbage collector based on liveness, i.e., objects are reclaimed as soon as they become dead (i.e, will not be used in the future). Then, we obtain the UB L by taking advantage of the fact that the object created immediately before ① and those to which "a.f" and "c" point are dead at program point ③, and that the object created at the end of m_2 is dead at program point ④. This information is reflected in the elements of

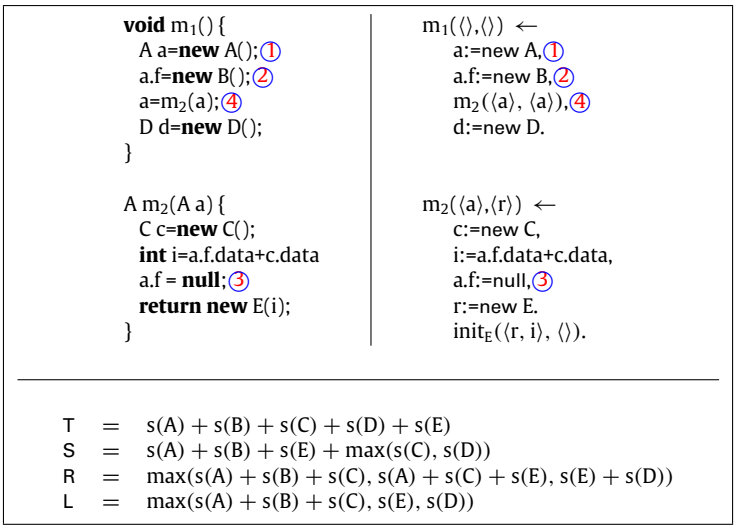

Fig. 1. A Java program and its memory requirements: T = total-allocation; S = scope-based; R = reachability-based; L = liveness-based.

the max similarly to what we have seen for R. Note that, in theory, the peak heap consumption inferred in L is indeed the minimal memory requirement for executing the method.

1.2. Summary of contributions and applications

Our overall contribution is a flexible and powerful approach to infer the heap space consumption of object-oriented imperative bytecode programs. Technically, the article makes the following important contributions:

1. We first present a novel application of the cost analysis framework in [3] to infer bounds on the total memory allocation of sequential Java bytecode programs. This requires developing 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 they consume. For instance, the cost of creating a new object is the number of heap units allocated to that object.
2. In a next step, we develop a new analysis to infer UBs on the *active memory* at a program point, i.e., the memory that has been allocated *and* that cannot be collected by the GC at that program point. The key idea is to infer first an UB for the total memory allocation of the method. Then, this bound can be manipulated, by relying on information pre-computed on object lifetimes in order to extract from it an UB on the active memory.
3. As our main contribution, we present a novel approach to accurately estimating the peak heap consumption of object-oriented imperative programs which is parametric w.r.t the lifetime of objects and it can be instantiated with different GC strategies. The main challenge is to integrate into the framework object lifetime information where heap data might be garbage collected at any program state. This is non-trivial since we need to generate recurrence relations that capture the memory requirements at a *program point* level, rather than at a method level as all previous approaches do.
4. Finally, in order to improve the accuracy of the resulting UBs, we propose to use the technique of *partial evaluation* during the process of generating the recurrence equations which, as our experimental results show, allows us to accurately infer the memory requirements at the different program states.
5. We report on a prototype implementation that is integrated in the COSTA system [4] and experimentally evaluate it on the JOlden benchmark suite by using different models for GC. Preliminary results demonstrate that our system obtains reasonably accurate peak consumption UBs in a fully automatic way.

The resulting framework has the following important applications:

- Our work is the first to model accurately and safely the actual memory usage in Java-like languages under only the assumption that GC will work before exceeding the memory limit. In particular, the only requirement is that the heap size limit be fixed to the obtained UB and that GC be activated when new memory is to be allocated and the limit is reached. We believe this assumption is practical and realistic, since the least that one can expect from GC (i.e., the least restrictive assumption) is that it frees memory when no more memory is available.
- Our work can be used to produce very accurate results of the heap usage in *compile-time garbage collection*. Essentially, in compile-time GC, the compiler determines the lifetime of the variables that are created during the execution of the program, and thus also the memory that will be associated with these variables. Whenever the compiler can guarantee that a variable, or more precisely, parts of the memory resources that this variable points to at run-time, will never ever be accessed beyond a certain program instruction, then the compiler can add instructions to deallocate these resources

at that particular instruction without compromising the correctness of the resulting code. If the program instruction is followed by a series of instructions that require the allocation of new memory cells, then the compiler can replace the sequence of deallocation and allocation instructions, by instructions updating the garbage cells, hence reusing these cells. The information on the variables lifetime can be used by our analysis and achieve very accurate heap usage bounds.

- Also, if we base our analysis on the same liveness information, our approach is the first one to obtain UBs on the memory consumption for:
  1. GC schemes (for Java-like languages) that take advantage of liveness information inferred at compile time [31].
  2. Languages with region-based memory management [33] in which programs are instrumented with explicit region (de)allocation annotations by relying on a liveness analysis [12].
- The resulting heap space UBs provide information for understanding/debugging the memory usage of programs, which can be a critical resource.

1.3. Organization of the article

The rest of the article is organized as follows. In Section 2, we first introduce the language and the semantics on which our analysis is developed. We use an intermediate imperative language to which Java bytecode programs can be automatically translated. Then, we provide the concrete definitions related to the notions of memory consumption we want to infer statically later.

Section 3 presents the application of the cost analysis framework in [3] for inferring total memory consumption. We first recall some basic concepts related to the underlying size analysis. These concepts are used then to formalize the notion of total memory allocation equations, from which the UB on the memory consumption is obtained.

In Section 4, we present the notion of *collectible types*, which approximates the set of objects that can be garbage collected. This notion is parametric on the notion of object lifetime, which could be inferred by a reachability analysis or by a liveness analysis. Collectible types are used in order to then obtain the *peak heap consumption* of executing the program for any input data, as presented in Section 5. The accuracy of the analysis can be greatly improved by relying on the well-known technique of partial evaluation [22]. This is described in Section 6.

Experimental results showing the accuracy and efficiency of our method are presented in Section 7. Finally, Section 8 reviews related work and Section 9 concludes.

2. Basic concepts: language, semantics and memory consumption

To formalize our analysis, we consider a simple *rule-based* imperative language (in the style of any of the languages in [6,36,23]). The key features of the rule-based language which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* blocks guarded by a type check. It has been shown that Java bytecode (and hence Java) can be compiled into this intermediate language [3]. Moreover, the translation preserves the *heap* memory consumption of the original program.

2.1. The language

A *rule-based program* consists of a set of *procedures* and a set of classes. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to this grammar:

$$
\begin{aligned}
\mathit{rule} &::= p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \ \leftarrow g, b_1, \ldots, b_n \\
g &::= \mathit{true} \mid \mathit{exp}_1 \ \mathit{op} \ \mathit{exp}_2 \mid \mathsf{type}(x, C) \\
b &::= x := \mathit{exp} \mid x := \mathsf{new}\ \mathsf{C}^i \mid \mathsf{x} := \mathsf{y.f} \mid \mathsf{x.f} := \mathsf{y} \mid \mathsf{q}(\langle \bar{\mathsf{x}} \rangle, \langle \bar{\mathsf{y}} \rangle) \\
\mathit{exp} &::= \mathsf{null} \mid \mathit{aexp} \\
\mathit{aexp} &::= x \mid n \mid \mathit{aexp} - \mathit{aexp} \mid \mathit{aexp} + \mathit{aexp} \mid \mathit{aexp} * \mathit{aexp} \mid \mathit{aexp} / \mathit{aexp} \\
\mathit{op} &::= > \mid < \mid \leq \mid \geq \mid = \mid \neq
\end{aligned}
$$

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call. The language supports class definition and includes instructions for object creation and field manipulation. A class C is a finite set of typed field names, where the type can be integer or a class name. The superscript i on a class C is a unique identifier which associates objects with the program points where they have been created. For ease of notation, we assume that there are no two different procedures with the same name (even if they have different number of parameters). A program point $[k, j]$ corresponds to the point j of the k-th program rule. The points in a rule are assigned as follows: Let $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \leftarrow g, b_1^k, \ldots, b_n^k$ be a program rule which has $n+1$ program points, then $[k, 0]$ is the point after the execution of the guard g and before the execution of b_1, and $[k, n]$ the one after the execution of b_n.

Due to dynamic method resolution, in the case of a method invocation, the actual method to be called is only known at runtime. The compilation to the rule-based program is made easier by approximating this information and introducing it

```
class Test {

  static Tree m(int n) {
    if ( n>0 ) return new
      Tree[1](f(n,g(n)), m(n-1),m(n-1));
    else return null;
  }

  static List g(int n) {
    if ( n <= 0 ) return null
    else return new List[2](n,g(n-1));
  }

  static int f(int n, List l) {
    int r=0;
    while (l != null) {
      r += (new Long[3](l.data)).intValue();
      l = l.next;
    } // List[2] is not live

    for (int i=n; i>0; i--)
      r *= (new Integer[4](i)).intValue();
    return r;
  }
}
```

```
m(⟨n⟩, ⟨r⟩) ←
    n > 0, (a)
    s0 := new Tree[1], (b)
    g(⟨n⟩, ⟨s1⟩), (c)
    f(⟨n, s1⟩, ⟨s1⟩),
    s2 := n − 1, (d)
    m(⟨s2⟩, ⟨s2⟩),
    s3 := n − 1, (e)
    m(⟨s3⟩, ⟨s3⟩),
    init_Tree(⟨s0, s1, s2, s3⟩, ⟨⟩),
    r = s0.
m(⟨n⟩, ⟨r⟩) ←
    n ≤ 0,
    r := null.

f(⟨n, l⟩, ⟨r⟩) ←
    r := 0, (f)
    f1(⟨l, r⟩, ⟨l, r⟩),
    i := n, (g)
    f2(⟨i, r⟩, ⟨i, r⟩).

f1(⟨l, r⟩, ⟨l, r⟩) ←
    l ≠ null, (h)
    s0 := new Long[3],
    s1 := l.data,
    init_Long(⟨s0, s1⟩, ⟨⟩),
    intValue_Long(⟨s0⟩, ⟨s0⟩),
    r := r + s0,
    l := l.next, (i)
    f1(⟨l, r⟩, ⟨l, r⟩).
f1(⟨l, r⟩, ⟨l, r⟩) ←
    l = null.

f2(⟨i, r⟩, ⟨i, r⟩) ←
    i > 0, (j)
    s0 := new Integer[4],
    init_Int(⟨s0, i⟩, ⟨⟩),
    intValue_Int(⟨s0⟩, ⟨s0⟩),
    r := r ∗ s0,
    i := i − 1, (k)
    f2(⟨i, r⟩, ⟨i, r⟩).
f2(⟨i, r⟩, ⟨i, r⟩) ←
    i ≤ 0.

g(⟨n⟩, ⟨r⟩) ←
    n ≤ 0,
    r := 0.
g(⟨n⟩, ⟨r⟩) ←
    n > 0,
    s0 := n − 1, (l)
    g(⟨s0⟩, ⟨s0⟩), (m)
    s1 := new List[2],
    init_List(⟨s1, n, s0⟩, ⟨⟩),
    r := s1.
```

Fig. 2. Java code of our second running example and rule-based representation of m, f and g. Method m is the entry method.

explicitly in the control flow graph (see [3]). This is done by adding new blocks, which are called dispatch blocks, containing calls to the actual methods which might be called at runtime. The access to these blocks is guarded by mutually exclusive conditions on the runtime class of the object whose method is called. This is represented in the rule-based program by a type comparison through the instruction type(x, C), which succeeds if the runtime class of x is exactly C. Our analysis will compute a safe approximation by taking all such possible runtime methods into account.

The translation from (Java) bytecode to the rule-based form is performed in two steps [3]. First, a control flow graph is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. For simplicity, our language does not include advanced features of Java such as exceptions, interfaces, static fields, access control and primitive types besides integers and references, but our implementation deals with full (sequential) Java bytecode.

Example 2. Fig. 2 shows the Java source (at the top) of a second example that we will use in the paper that has interesting memory consumption, namely exponential and polynomial bounds. The source code is shown only for clarity as the analyzer generates the rule-based representation (at the bottom) from the corresponding bytecode only. The first two rules correspond to method m. Each of them is guarded by a corresponding condition, resp. $\mathbf{n} > \mathbf{0}$ and $\mathbf{n} \leq \mathbf{0}$. Variable names of the form s_i indicate variables that originate from stack positions. For instance, the "new Tree[1]" instruction creates an object of type Tree allocated at the allocation site 1 (the superscript 1 is the unique identifier for such an allocation site) and assigns the corresponding reference to the variable s_0. This corresponds to pushing the reference on the stack in the original bytecode. Next, methods g and f are invoked. Then, the local variable n is decremented by one and the result is assigned to s_2 and a recursive call is done. A similar recursive invocation follows. Constructor methods are named init (as in Java bytecode). In both rules, the return value is r which in the first one contains the object reference and in the second

$$(1)\quad \frac{b \equiv x := exp,\ \ v = eval(exp, tv)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A;\ h \rightsquigarrow \langle p, bc, tv[x \mapsto v]\rangle{\cdot}A;\ h}$$

$$(2)\quad \frac{b \equiv x := \text{new } C^i,\quad o{=}\text{newobject}(C^i),\quad r{\notin}dom(h)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A;\ h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}A;\ h[r \mapsto o]}$$

$$(3)\quad \frac{b \equiv x := y.f,\ \ tv(y) \neq \text{null},\ \ o = h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A;\ h \rightsquigarrow \langle p, bc, tv[x \mapsto o.f]\rangle{\cdot}A;\ h}$$

$$(4)\quad \frac{b \equiv x.f := y,\ \ tv(x) \neq \text{null},\ \ o = tv(x)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A;\ h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}A;\ h[o.f \mapsto tv(y)]}$$

$$(5)\quad \frac{\begin{array}{c} b \equiv q(\langle\bar{x}\rangle, \langle\bar{y}\rangle),\ \text{there is a program rule } q(\langle\bar{x}'\rangle, \langle\bar{y}'\rangle){:=}g, b_1, \ldots, b_k \\ \text{such that } tv'{=}\text{newenv}(q),\ \forall i.tv'(x'_i) = tv(x_i),\ eval(g, tv') = true \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A;\ h \rightsquigarrow \langle q, b_1 \cdot \ldots \cdot b_k, tv'\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv\rangle{\cdot}A;\ h}$$

$$(6)\quad \frac{}{\langle q, \epsilon, tv\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv'\rangle{\cdot}A;\ h \rightsquigarrow \langle p, bc, tv'[\bar{y} \mapsto tv(\bar{y}')]\rangle{\cdot}A;\ h}$$

Fig. 3. Operational semantics of bytecode programs in rule-based form.

one contains null. In the rule-based representation for f, loops are extracted as separate procedures which are treated by the analysis as methods; in our example, f_1 and f_2 are intermediate procedures that correspond, resp., to the **while** and **for** loops in f. We refer to each procedure as a *scope*, which can be a method definition or an intermediate block.

2.2. Semantics

The execution of bytecode in rule-based form is the same as that of standard bytecode; a thorough explanation is outside the scope of this paper (see [25]). The *operational semantics* for rule-based bytecode is shown in Fig. 3. An *activation record* is of the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a sequence of instructions and tv a variable mapping. Executions proceed between *configurations* of the form $A; h$, where A is a stack of activation records and h is the *heap*, which is a partial map from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by the memory location r in h, $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$, and $\text{dom}(h)$ to denote the set of memory locations in the heap h. An object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the type of the fields.

Intuitively, rule (1) accounts for all instructions in the bytecode semantics that perform arithmetic and assignment operations. The evaluation $eval(exp, tv)$ returns the value of the arithmetic or Boolean expression exp for the values of the corresponding variables from tv in the standard way, and for reference variables, it returns the reference. Rules (2), (3) and (4) deal with objects. We assume that $\text{newobject}(C^i)$ creates a new object of class C and initializes its fields to either 0 or null, depending on their types. Note that rules (3) and (4) require that the dereferenced variable is different from null, we assume that the program aborts when dereferencing a nullpointer. Rule (5) (resp., (6)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}, \bar{y}']$ records the association between the formal and actual return variables. It is assumed that newenv creates a new mapping of local variables for the corresponding method, where each variable is initialized as newobject does.

A complete execution starts from an *initial configuration* $\langle\bot, p(\langle\bar{x}\rangle, \langle\bar{y}\rangle), tv\rangle; h$ and ends in a *final configuration* of the form $\langle\bot, \epsilon, tv'\rangle; h'$ where tv and h are initialized to suitable initial values, tv' and h' include the final values, and $\bot$ is a special symbol indicating an initial state. Complete executions can be regarded as *traces* $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_n$, denoted $S_0 \rightsquigarrow^* S_n$, where S_n is a final configuration. Infinite traces correspond to non-terminating executions. Traces that correspond to complete or infinite executions are referred to as complete traces.

2.3. Basic notions of memory consumption

We use s(C) to denote the amount of memory required to hold an instance object of class C, s(o) denotes the amount of memory occupied by an object o, and s(h) denotes the amount of memory occupied by all objects in the heap h, namely $\Sigma_{r \in \text{dom}(h)} s(h(r))$. Since in the semantics of Fig. 3, there is no deallocation, given a finite complete trace $t \equiv S_0 \rightsquigarrow^* S_n$, its *total memory* allocation is defined as $total(t) = s(h_n) - s(h_0)$. If the derivation is infinite, then $total(t) = \max(\{s(h_i) \mid S_i \in t \wedge S_i = (A_i; h_i)\}) - s(h_0)$.

Languages with automatic memory management aim at automatically reclaiming memory (freeing it) when its content can no longer affect future computations. Therefore, in the presence of any GC, the size of the heap might also decrease.

Hence, the *peak heap* consumption of an execution is defined as the maximum size of all intermediate heaps. More formally, given a complete trace t, and assuming that the initial heap h_0 contains initial data that will not be deallocated during the execution, the peak memory usage of t in the presence of GC is defined as $peak(t) = \max(\{s(h_i) \mid S_i \in t \wedge S_i = (A_i; h_i)\}) - s(h_0)$. This is the notion that our analysis aims at approximating *statically*, i.e., our goal is to obtain a sound and tight UB on the peak heap usage for any input data and without having to run the program. Note that $total(t)$ and $peak(t)$ might be undefined for infinite derivations, in such case we assume it is ∞.

3. Total memory allocation

Any heap space analysis aims at approximating the memory usage of the program as a function of the input *data sizes*. As customary, the *size* of data is determined by its variable type [3]: the size of an integer variable is its value; the size of an array is its length; and the size of a reference variable is the length of the longest path that can be traversed through the corresponding object (e.g., length of a list, depth of a tree, etc.). We use the original variable names (possible primed) to refer to the corresponding size variables; but we write the size in *italic*, e.g., since variable l of procedure f_1 (in Fig. 2) is a reference to a list, then l represents its length. The size measure is mainly used for estimating the number of iterations of recursive procedures. Note that, the size measure of data structures, as defined here, is unrelated to the notation s(C) which measures the actual space occupancy, as defined in the previous section. When we need to compute the sizes $\bar{v}$ of a tuple of variables $\bar{x}$, we use the notation $\bar{v} = \alpha(\bar{x}, tv, h)$, which means that the integer value v_i is the size of the variable x_i in the context of the variables table tv and the heap h. For example, we need to access the heap, h, where the list l is allocated to compute its length v. If x is an integer variable, then its size (value) is obtained from the variable table tv.

Standard *size analysis* is used in order to obtain relations between the sizes of the program variables at different program points [15]. For instance, associated with the recursive rule f_1, we infer the size relation $l > l'$ which indicates that the length of l decreases when calling f_1 recursively, where l' refers to the size of l at the program point where f_1 is called recursively. We denote by φ_r the conjunction of linear constraints that describes the size relations between the abstract variables of a rule r (see [15,3] for more information). The rest of our analysis is parametric w.r.t. the size relations, which are an external component, and can also admit user-defined size relations as [18].

Given a program P and the relations φ for its rules, a recurrence relation (RR) system for total memory allocation is generated by applying the following definition to all rules in P.

Definition 3 (*Total Memory Allocation Equations*)**.** Let r be a rule of the form $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \leftarrow g, b_1, \ldots, b_n$ and φ_r its corresponding size relations. Then, the *total memory allocation equation* of r is defined as:

$$p(\bar{x}) = \Sigma_{i=1}^{n} \mathcal{M}(b_i)\ \ \varphi_r$$

where $\mathcal{M}(x := \text{new } C^i) = s(C^i)$, $\mathcal{M}(q_i(\langle\bar{x}_i\rangle, \langle\bar{y}_i\rangle)) = q_i(\bar{x}_i)$; otherwise $\mathcal{M}(b_i) = 0$.

Note that each call in the rule $q_i(\langle\bar{x}_i\rangle, \langle\bar{y}_i\rangle)$ has a corresponding abstract version $q_i(\bar{x}_i)$ where $\bar{x}_i$ are the size abstractions of $\bar{x}_i$ at the corresponding program point. The output variables are ignored in the RR as the cost is a function of the *input* data sizes, but the relation they impose on other variables is kept in φ_r. The same procedure name is used to define its associated cost relation, but in *italic* font. An important point is that the RR must keep the order of the corresponding *size* constants and the calls in their right hand sides (rhs) exactly as they appear in the rule they are generated from. This is required in order to make RR capture the heap space usage at a program point level.

Example 4. The total allocation equations for the rules in Fig. 2 are:

$$\begin{array}{lll}
m(n) & = s(\text{Tree}^1) + g(n) + f(n, s_1) + m(s_2) + m(s_3) & \{n{>}0, s_1{=}n, s_2{=}n{-}1, s_3{=}n{-}1\} \\
m(n) & = 0 & \{n{=}0\} \\
f(n, l) & = f_1(l, r) + f_2(i, r') & \{r{=}0, i{=}n\} \\
f_1(l, r) & = s(\text{Long}^3) + f_1(l', r') & \{l \geq 1, l' < l\} \\
f_1(l, r) & = 0 & \{l = 0\} \\
f_2(i, r) & = s(\text{Integer}^4) + f_2(i', r') & \{i > 0, i' = i - 1\} \\
f_2(i, r) & = 0 & \{i \leq 0\} \\
g(n) & = g(s_0) + s(\text{List}^2) & \{n > 0, s_0 = n - 1\} \\
g(n) & = 0 & \{n \leq 0\}.
\end{array}$$

For simplicity we ignore calls to constructors $\text{init}_{\text{Long}}$, $\text{init}_{\text{Integer}}$, $\text{init}_{\text{List}}$, and $\text{init}_{\text{Tree}}$, assuming they do not consume any heap memory. The total allocation of m is defined by the first two equations. The first one states that the total memory consumption when executing m with an input value $n > 0$ is the size of an object of type Tree^1, plus the consumption of the corresponding calls to g, f and the recursive calls to m. The attached constraints describe the size relations between the local variables and the input variable n. The second equation corresponds to the base-case of m, i.e., when $n \leq 0$. The total allocation of f is defined by the third equation. It is the sum of the total allocations of loops f_1 and f_2. The total allocation of f_1 is defined in the fourth and fifth equations. The fourth equation states that a call to f_1 with a non-empty list of length l occupies the size of an object of type Long^3, plus the occupation of the recursive call with a list of length l' which is smaller than l (due to the instruction $l := l.next$). The fifth rule corresponds to the case of calling f_1 with an empty list, i.e., $l = 0$.

Similarly, the sixth and seventh equations define the total allocation of f_1. The last two equations define the total allocation of g. The total allocation equations for methods m_1 and m_2 in Fig. 1 are:

$$\begin{aligned} m_1 &= s(A) + s(B) + m_2(a) + s(D) \\ m_2(a) &= s(C) + s(E) \end{aligned}$$

which, as expected, are simpler than those for the program Fig. 2 since the corresponding program is not recursive and does not contain any loop.

Once the RR are generated, a worst-case cost analyzer uses a solver in order to obtain closed-form UBs, i.e., cost expressions without recurrences. The technical details of the process of obtaining a cost expression from the RR are not explained in the paper as our analysis does not require any modification to this part. In what follows, we rely on the RR solver of [1] to obtain UBs for our examples, sometimes simplified by removing constants to facilitate understanding. Given a RR $p(\bar{x})$, we denote by $p^{ub}(\bar{x})$ its UB. The UBs that [1] can infer from the above RR are *cost expressions* of the following form:

$$e \equiv n \mid s(C) \mid \mathsf{nat}(l) \mid \log(\mathsf{nat}(l)+1) \mid e * e \mid e_1 + e_1 \mid 2^{\mathsf{nat}(l)} \mid \max(\{e_1, \ldots, e_k\})$$

where n is an integer, l is a linear expression and C is a class. Function nat is defined as $\mathsf{nat}(v) = \max(\{v, 0\})$ to avoid negative values. A cost expression must evaluate to a non-negative value for any input.

Example 5. The total memory UBs obtained from the first RR of Example 4 are:

$$\begin{aligned} g^{ub}(n) &= \mathsf{nat}(n) * s(\mathrm{List}^2) \\ f_2^{ub}(i, r) &= \mathsf{nat}(i) * s(\mathrm{Integer}^4) \\ f_1^{ub}(l, r) &= \mathsf{nat}(l) * s(\mathrm{Long}^3) \\ f^{ub}(n, l) &= \mathsf{nat}(l) * s(\mathrm{Long}^3) + \mathsf{nat}(n) * s(\mathrm{Integer}^4) \\ m^{ub}(n) &= (2^{\mathsf{nat}(n)} - 1) * (s(\mathrm{Tree}^1) + \mathsf{nat}(n) * (s(\mathrm{List}^2) + s(\mathrm{Long}^3) + s(\mathrm{Integer}^4))). \end{aligned}$$

Intuitively, for method f (and its intermediate procedure f_1 and f_2), observe that the first (resp. the second) loop is executed $\mathsf{nat}(l)$ (resp. $\mathsf{nat}(n)$) times and at each iteration a Long^3 (resp. $\mathrm{Integer}^4$) object is allocated. For m, we have an exponential number of recursive calls, at each one: an object Tree^1 is allocated, g allocates $\mathsf{nat}(n)$ objects List^2 and f contributes with its allocation. The inferred UBs capture exactly this intuition. The total memory UBs obtained from the second RR of Example 4 are:

$$\begin{aligned} m_1^{ub} &= s(A) + s(B) + s(C) + s(D) + s(E) \\ m_2^{ub}(a) &= s(C) + s(D). \end{aligned}$$

The following theorem states the soundness of the total memory allocation analysis. The proof is based on the soundness of the generic cost analysis framework of [3] and soundness of the UB solver [1].

Theorem 6 (*Soundness*)**.** *Let P be a program with an entry procedure p, and let $p^{ub}(\bar{x})$ be an UB for the corresponding total memory allocation equations generated in Definition 3. Then, for any complete trace t that starts from an initial state $S_0 = \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$ it holds that $p^{ub}(\bar{v}) \geq total(t)$ where $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$.*

4. Inference of object lifetime

A *GC strategy* classifies objects in the heap of a given configuration into two categories: those which are collectible and those which are not. Different strategies have different criteria to determine when an object is collectible. In this paper we apply our formalization on the following three GC strategies.

- *Reachability-based GC.* In this strategy an object is determined to be collectible if it is unreachable, namely, if there is no variable in the program environment (activation records stack) pointing to it directly or indirectly. This strategy is widely used in practice. We denote it by $\mathcal{G}_r$.
- *Scope-based GC.* In this strategy an object is determined to be collectible if it has been created during a *method call* and is unreachable upon exit from that call. We denote this GC strategy by $\mathcal{G}_s$.
- *Liveness-based GC.* In this strategy an object is determined to be collectible if it is *not live* (i.e., it is *dead*), namely, if it is not accessed or modified from that point on during the execution. We denote this GC strategy by $\mathcal{G}_l$.

Due to the soundness of the translation, the above strategies are equivalent when considered at the level of a Java program or its corresponding rule-based program [5]. However, in the scope-based strategy we have to distinguish procedure calls (in the rule-based program) that correspond to method calls (e.g., f, m and g in Fig. 2) from those calls to intermediate procedures representing loops (e.g, f_1 and f_2 in Fig. 2).

A common approach to statically over-approximate the above collectible information, is to provide information on the *types* (i.e., the class name with allocation site) of collectible objects instead of the actual objects. In this case a type C^i is classified as collectible only if all instances of C^i are collectible. This approximation on types itself is also undecidable. Therefore, the corresponding analysis might say that a type is not collectible while it is. Note that computing the collectible information is typically done w.r.t. an *entry* procedure (such as `main` in Java). In what follows we introduce two notions of collectible types and apply them to the above strategies:

- *collectible types for a procedure* which approximates the set of types that are created during the execution of a given procedure and are collectible upon exit from that procedure; and
- *collectible types at a program point* which approximates the collectible types whenever a given program point is reached.

The distinction between these two notions is crucial, not only for covering more GC strategies as the reader might first expect (e.g., for scope-based), but also to improve the precision loss introduced by considering collectible types instead of collectible objects, mainly when dealing with recursive procedures. This point is explained below in further detail once all notions become clear. In what follows, by *applying a GC strategy on a configuration* we mean the process of removing all collectible objects from the corresponding heap. For simplicity, we assume that *all* collectible objects are collected whenever the corresponding garbage collector is activated. In practice, this can be adjusted to the actual underlying GC strategy. For ease of notation, we say that a type A is reachable (resp. live) to indicate that there is a reachable (resp. live) object of type A.

Definition 7 (*Collectible Types for a Procedure*)**.** Let P be a program with an entry procedure p, $\mathcal{G}$ be a GC strategy, C^i a type, and q a procedure defined in P. We say that type C^i is *collectible* for q (or upon exit from q) w.r.t. $\mathcal{G}$, if

- the instruction new C^i is reachable from q, i.e., the procedure can actually create an object of type C^i; and
- for any complete trace t that starts from an initial state $\langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$, if $\langle m, q(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \cdot bc, tv_j \rangle \cdot A; h_j \rightsquigarrow^* \langle m, bc, tv_k \rangle \cdot A; h_k$ is a sub-trace of t, then applying $\mathcal{G}$ on $\langle m, bc, tv_k \rangle \cdot A; h_k$ results in a heap which does not have any object of type C^i which was not in heap h_j.

The set of all collectible types for q are denoted by $\mathcal{C}(\mathcal{G}, q)$.

Such set of collectible types can be approximated by first applying *points-to* analysis [38], and then *reachability* or *heap-liveness* [12] analysis (depending on the GC strategy). The latter analyses use the points-to information to determine whether the lifetime of objects of a given type can be proven to be restricted only to the method where they have been created. This is known as *escape analysis* [30,9].

Example 8. The following table summarizes the collectible types for the different procedures in Figs. 1 and 2, for the different GC strategies:

q	$\mathcal{C}(\mathcal{G}_r, q)$	$\mathcal{C}(\mathcal{G}_s, q)$	$\mathcal{C}(\mathcal{G}_l, q)$
m	{Long^3, Integer^4, List^2}	{Long^3, Integer^4, List^2}	{Long^3, Integer^4, List^2}
g	$\emptyset$	$\emptyset$	$\emptyset$
f	{Long^3, Integer^4}	{Long^3, Integer^4}	{Long^3, Integer^4}
f_1	{Long^3}	$\emptyset$	{Long^3}
f_2	{Integer^4}	$\emptyset$	{Integer^4}
m_1	{A, B, C, D, E}	{A, B, C, D, E}	{A, B, C, D, E}
m_2	{C}	{C}	{C, E}

For procedures f_1 and f_2, the types Integer^4 and Long^3 are not collectible when considering the scope-based strategy $\mathcal{G}_s$. This is because these procedures do not correspond to methods in the corresponding Java program but rather to intermediate procedures for the loops. In m_2, note that E is collectible when considering $\mathcal{G}_l$ since there is no execution (starting from the entry m_1) in which such type is used after executing m_2.

Definition 9 (*Collectible Types at a Program Point*)**.** Let P be a program with an entry procedure p, C^i a type, $[k, j]$ a program point, and $\mathcal{G}$ a GC strategy. We say that the type C^i is *collectible at* $[k, j]$ *w.r.t.* $\mathcal{G}$, if for any complete trace t that starts from an initial state $S_0 = \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$ and any state $S_l = \langle q^k, b_j^k \cdot bc, tv_l \rangle \cdot A; h_l$ in t, applying $\mathcal{G}$ on S_l results in a heap which does not have any object of type C^i. The set of all collectible types at $[k, j]$ w.r.t. $\mathcal{G}$ is denoted $\mathcal{C}(\mathcal{G}, [k, j])$.

The set of collectible types at a program point $[k, j]$ w.r.t. a reachability-based GC strategy $\mathcal{C}(\mathcal{G}_r, [k, j]))$ can be approximated using points-to analysis [38].

Example 10. The following collectible types information w.r.t. $\mathcal{G}_r$ is obtained for the program points in Fig. 2. At ⓐ, ⓑ, ⓓ and ⓔ, the set reachable types is {*Tree*^{1}}; and at ⓒ, ⓕ, ⓖ, ⓗ, ⓘ, ⓙ, ⓚ, ⓛ and ⓜ the set of reachable types is {Tree^1, List^2}. Likewise, the reachability information for the program points in Fig. 1 is that at ① the set of reachable types is {A}, at ② is {A, B}, at ③ is {A, C}, and at ④ is {E}. The complementary sets are the unreachable types and therefore collectible w.r.t. $\mathcal{G}_r$.

The set of collectible types at a program point $[k, j]$ w.r.t. a scope-based GC strategy $\mathcal{C}(\mathcal{G}_s, [k, j]))$ can be approximated using points-to [38] and escape [30,9] analysis.

Example 11. The following collectible types information w.r.t. $\mathcal{G}_s$ is obtained for the program points in Fig. 2. At ⓐ, ⓑ, ⓒ, ⓓ and ⓔ the types Long^3 and *Integer*^{4} are collectible. This is because at any configuration that corresponds to one of these program points, the instances of these types were created in calls to f that have been already completed, and they were unreachable upon exit from f. Note that, for example, *List*^{2} is not collectible at ⓓ (w.r.t. $\mathcal{G}_s$) even though it is unreachable. This is because the instances created in g were still reachable upon exit from g and became unreachable only when s_1 is

overwritten (output argument of f). For the remaining program points, nothing is collectible. Since f_1 and f_2 are intermediate rules and not procedures, the objects created during calls to these methods are not collectible upon exit. Likewise, for the program points in Fig. 1, we have that type C is collectible at ④. For the remaining program points nothing is collectible.

The set of all collectible types at a program point $[k, j]$ w.r.t. a liveness-based GC strategy $\mathcal{C}(\mathcal{G}_l, [k, j])$ can be approximated by using points-to analysis and a backwards heap-liveness inference similar to [12].

Example 12. The following collectible types information w.r.t. $\mathcal{G}_l$ is obtained for the program points in Fig. 2: at ⓐ, ⓑ, ⓓ, ⓔ, ⓖ, ⓙ, and ⓚ the set of live types is $\{\text{Tree}^1\}$; and at ⓒ, ⓕ, ⓗ and ① it is $\{\text{Tree}^1, \text{List}^2\}$. The important point is that, at ⓕ, objects of type List^2 are still live since their field *data* still has to be accessed, but at ⓖ the access has already been performed and $List^2$ is not live anymore. Similarly, the liveness information obtained for the program points in Fig. 1 is that at ① the set of live types is $\{A\}$ and at ② is $\{A, B\}$, and at ③ and ④ is $\emptyset$. The complementary sets are the dead types and therefore collectible w.r.t. $\mathcal{G}_l$.

The following example demonstrates the need for two different notions of collectible objects.

Example 13. Consider the following recursive Java program:

```
void m(int n) {
  if (n <= 0) return;
  A x = new A(n);
  x = new B(x.f());
  m(n−1);
  m(n−1);
  System.out.println(x.f());
}
```

Its peak memory consumption w.r.t. strategy $\mathcal{G}_r$ is $n * \text{s(B)} + \text{s(A)}$. This means that we only need space for one object of type A and n objects of type B. In order to infer this bound we need to automatically prove the following conditions: (1) the object of type A becomes unreachable when variable x is overwritten; and (2) all objects of type B that are created during the first recursive call m(n − 1) are unreachable upon exit from this call. The first condition is easy to infer, however, the second is not straightforward. This is because not all objects of type B are unreachable upon exit from m(n − 1), the one created in the current scope is still reachable till the end of the method. Since we do not distinguish between objects of the same type it is not correct that type B is collectible after the first recursive call. In order to overcome this limitation we introduced the above two notions of collectible types. We will come back to this program in Example 16, once the above notions are formally defined, in order to show how our techniques handle such programs.

The notion of collectible types (both for procedures and program points) as defined above is context-insensitive, i.e., for a type to be collectible it must be collectible in all calling contexts. Obtaining context-sensitive information would require using context-sensitive [38] or object-sensitive [27] points-to analysis. This could be essential when dealing with programs with object-oriented features such as the factory pattern. Note that using such points-to analysis would not require substantial changes in the underlying technical details of our approach. The most important change is to modify the analysis such that it gives different names to objects of the same type that are created in a different context. This can be done in a similar manner to what we do with allocation sites in order to distinguish objects of the same type that are created at different program points. Incorporating such analysis in our approach is left for future work.

5. Heap space analysis

Let us first explain the intuition behind the analysis. Suppose that n_1 is the actual memory usage at some program point, and that n_2 is the amount of memory that corresponds to objects that can be freed by the garbage collector at that point. Then, obviously $n_1 - n_2$ is the current memory usage after freeing the memory that corresponds to those collectible objects. In order to obtain an UB for $n_1 - n_2$, one can obtain an UB for n_1 and a lower bound for n_2. In our context, we have a symbolic UB e_1 for n_1. However, the information about collectible classes is an over-approximation. Hence, it cannot be used to obtain a lower bound on n_2. The main idea is that, since the collectible classes sets in Section 4 provide the information that *all* instances of specific classes at that program point can be freed and, given that e_1 is a symbolic UB, we can obtain a sound UB for $n_1 - n_2$ by replacing in e_1 all occurrences of s(C) by 0 for all C which are in the set of collectible classes.

Example 14. Let us assume that $\text{nat}(l) * \text{s}(\text{Long}^3) + \text{nat}(n) * \text{s}(\text{Integer}^4) + \text{nat}(l) * \text{s(A)}$ is an UB on the total memory consumption of method f of Fig. 2 (note that we added $\text{nat}(l) * \text{s(A)}$ just for the sake of explaining the idea). In Example 8, we have seen that objects of type Long^3 and Integer^4 do not escape from f, thus syntactically replacing each $\text{s}(\text{Long}^3)$ and $\text{s}(\text{Integer}^4)$ by 0 results in $\text{nat}(l) * \text{s(A)}$, which is an UB on the amount of the memory still in use upon exit from f.

By relying on this basic idea, we will estimate the *active memory* at each program point and then choose the maximum among them.

5.1. Inference of active memory

We start by computing an UB on the *active memory* associated to an expression $e_1 + \cdots + e_n$, where each e_i is either a call to a procedure $q(\bar{x})$ or an expression of the form $\mathrm{s}(\mathrm{C}^{\mathrm{i}})$. Such expression corresponds to the memory consumption of executing a sequence of instructions (from left to right), where $\mathrm{s}(\mathrm{C}^{\mathrm{i}})$ is the memory consumed by an instruction that creates an object of type C^i and $q(\bar{x})$ that of calling a procedure. The active memory for such expression is the memory that has been created during the corresponding execution and cannot be garbage collected at the end of the execution. In what follows, given a cost expression e and a set of types X, we denote by $e|_X$ the cost expression that results from replacing each $\mathrm{s}(\mathrm{C}^{\mathrm{i}})$ in e by 0 if $C^i \in X$.

Definition 15 (*Active Memory*)**.** Given an expression $e \equiv e_1 + \cdots + e_n$, the *active memory* of e at a program point $[k, j]$ w.r.t. a GC strategy $\mathcal{G}$, denoted $\mathcal{A}(e, [k, j], \mathcal{G})$, is defined as follows: first obtain e' by replacing each $e_i \equiv q(\bar{x})$ by $q^{ub}(\bar{x})|_{\mathcal{C}(\mathcal{G},q)}$ and then $\mathcal{A}(e, [k, j], \mathcal{G}) = e'|_{\mathcal{C}(\mathcal{G},[k,j])}$.

The intuition behind this definition is that, in order to eliminate the collectible types from $e_1 + \cdots + e_n$, we first eliminate those that do not escape from procedure calls (i.e., when $e_i \equiv q(\bar{x})$) and, in a second phase, those that are collectible at the corresponding program point. This two phase elimination, as defined in Definition 15, is fundamental as we have explained in Example 13.

Example 16. Let us see the importance of the two phase elimination by reconsidering the program of Example 13 w.r.t. $\mathcal{G}_r$. We want to compute the active memory at the program point immediately after the first recursive call. Let us denote this program point by ①. First note that (1) the total memory UB for m is $m^{ub}(n) = (\mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B})) * (2^{\mathrm{nat}(n)} - 1)$; (2) objects of type A and B are collectible upon exit from m, i.e., $\mathcal{C}(\mathcal{G}_r, m) = \{A, B\}$; (3) at ① there is a reachable object of type B but all those of type A are unreachable, i.e., $\mathcal{C}(\mathcal{G}_r, ①) = \{A\}$; and (4) the total memory at ① is $e = \mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}) + m(n-1)$. According to Definition 15, $\mathcal{A}(e, ①, \mathcal{G}_r)$ is computed in two steps:

1. In the first step we compute e' by replacing $\mathrm{m}(\mathrm{n}-1)$ in e by $m^{ub}(n-1)|_{\mathcal{C}(\mathcal{G}_r,m)}$. Note that $m^{ub}(n-1)|_{\mathcal{C}(\mathcal{G}_r,m)} = 0$, since both A and B are in $\mathcal{C}(\mathcal{G}_r, m)$, and thus $e' = \mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B})$.
2. In the second step we compute $e'|_{\mathcal{C}(\mathcal{G}_r,①)}$, which results in $\mathrm{s}(\mathrm{B})$ since A is collectible at ①.

Thus, $\mathcal{A}(e, ④, \mathcal{G}_r) = \mathrm{s}(\mathrm{B})$. Now let us see what we get if we compute $\mathcal{A}(e, ①, \mathcal{G}_r)$ only in one phase, i.e., we generate e' from e replacing $\mathrm{m}(\mathrm{n}-1)$ by $m^{ub}(n-1)$, without removing those objects that are collectible upon exit from m, and then compute $e'|_{\mathcal{C}(\mathcal{G}_r,①)}$: we first get $e' = \mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}) + (\mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B})) * (2^{\mathrm{nat}(n)} - 1)$, and then $e'|_{\mathcal{C}(\mathcal{G}_r,①)} = \mathrm{s}(\mathrm{B}) * 2^{\mathrm{nat}(n-1)}$, which is much more imprecise than what we get with the two phase elimination approach. This is because, when computing $e'|_{\mathcal{C}(\mathcal{G}_r,①)}$, we cannot distinguish between the single reachable object of type B (that comes from e) and those unreachable objects of type B that come from $m^{ub}(n-1)$, and thus, for soundness, we would not eliminate any of them.

Example 17. The total memory consumption at program point ④ is described by the expression $e = \mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}) + m_2(a)$. The total memory UB for m_2, as computed in Example 5, is $m_2^{ub}(a) = \mathrm{s}(\mathrm{C}) + \mathrm{s}(\mathrm{E})$. Using the collectible information of Examples 8 and 10–12, we obtain $\mathcal{A}(e, ④, \mathcal{G}_r) = \mathrm{s}(\mathrm{E})$, $\mathcal{A}(e, ④, \mathcal{G}_s) = \mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}) + \mathrm{s}(\mathrm{E})$, and $\mathcal{A}(e, ④, \mathcal{G}_l) = 0$. Note that type A is not eliminated from e when considering $\mathcal{G}_s$. This is because it has been created in the same method as program point ④. Therefore, according to this GC strategy, it can be collected only upon exit from m_1. Also, $\mathrm{s}(\mathrm{E})$ is eliminated from e when considering $\mathcal{G}_l$ because, even if it is reachable through variable a, it is never accessed or modified afterwards.

5.2. Peak heap space cost relations

The main idea behind our heap space analysis is to produce disjunctive equations which capture the active memory at the program points where we know that the memory usage can change, i.e., at the memory allocation instructions. Since the RR generated in Definition 3 allow us to identify exactly these points by means of their associated s constants, we generate the parametric peak heap consumption equations from them.

Definition 18 (*Peak Heap Space Equations*)**.** Given a total memory allocation equation "$p(\bar{x}) = e_1 + \cdots + e_n,\ \varphi$" and a GC strategy $\mathcal{G}$, the corresponding *peak heap space equation* is defined as: $\hat{p}(\bar{x}) = max(f_1, \ldots, f_n),\ \varphi$, where each $\mathrm{f_i}$ is defined as $\mathrm{f_i} = \mathcal{A}(\mathrm{e_1} + \cdots + \mathrm{e_{i-1}}, [\mathrm{k}, \mathrm{j}], \mathcal{G}) + \hat{\mathrm{e}}_\mathrm{i}$ such that $[k, j]$ is the program point that corresponds to e_i and $\hat{e}_i$ is equal to $\mathrm{s}(\mathrm{C}^{\mathrm{j}})$ when $e_i \equiv \mathrm{s}(\mathrm{C}^{\mathrm{j}})$ and to $\hat{q}(\bar{x})$ when $e_i \equiv q(\bar{x})$.

An important point in the above definition is that, when computing the active memory of $e_1 + \cdots + e_{i-1}$, procedure calls are replaced by their escaped memory UBs and hence the result is a symbolic cost expression. In contrast, we have that $\hat{e}_i$, when e_i is a call, will be defined by corresponding peak heap space equations when applying Definition 18 to the equations defining e_i. As mentioned in Section 3, it is crucial for the above definition to maintain the order (and program point information) in the expressions of the rhs of the total allocation equations in order to be able to apply the program point collectible classes information into them.

Example 19. Consider again the total memory equations of Example 5 for the program in Fig. 1. According to Definition 18, the following peak heap space equations are obtained for a generic GC strategy $\mathcal{G}$:

$$\begin{array}{lll} \hat{m}_1 & = max(\mathrm{s(A)}, & \\ & \quad \mathcal{A}(\mathrm{s(A)}, ①, \mathcal{G}) + \mathrm{s(B)}, & \\ & \quad \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)}, ②, \mathcal{G}) + \hat{m}_2(a), & \\ & \quad \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + m_2^{ub}(a), ④, \mathcal{G}) + \mathrm{s(D)}) & \\ \hat{m}_2(a) & = max(\mathrm{s(C)}, \mathcal{A}(\mathrm{s(C)}, ③, \mathcal{G}) + \mathrm{s(E)}). & \end{array}$$

The disjunctive information is handled in the solver by replacing the max operator by non-deterministic equations and finding an UB using [1]. Below, we show to the left (resp. right) the non-deterministic equations for the reachability-based strategy $\mathcal{G}_r$ before (resp. after) the elimination of the unreachable classes computed in Example 10. The notation ~~s(C)~~ indicates setting s(C) to zero every C that belongs to collectible classes:

$$\begin{array}{lll} \hat{m}_1 = \mathrm{s(A)} & & \\ \hat{m}_1 = \mathcal{A}(\mathrm{s(A)}, ①, \mathcal{G}_r) + \mathrm{s(B)} & = \mathrm{s(A)} + \mathrm{s(B)} & \\ \hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)}, ②, \mathcal{G}_r) + \hat{m}_2(a) & = \mathrm{s(A)} + \mathrm{s(B)} + \hat{m}_2(a) & \\ \hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + m_2^{ub}(a), ④, \mathcal{G}_r) + \mathrm{s(D)} & = \sout{\mathrm{s(A)}} + \sout{\mathrm{s(B)}} + \sout{\mathrm{s(C)}} + \mathrm{s(E)} + \mathrm{s(D)} & \\ \hat{m}_2(a) = \mathrm{s(C)} & & \\ \hat{m}_2(a) = \mathcal{A}(\mathrm{s(C)}, ③, \mathcal{G}_r) + \mathrm{s(E)} & = \mathrm{s(C)} + \mathrm{s(E)}. & \end{array}$$

The UBs obtained from these equations using [1] are:

$$\begin{array}{ll} \hat{m}_2^{ub}(a) = & \mathrm{s(C)} + \mathrm{s(E)} \\ \hat{m}_1^{ub} = & \mathrm{s(E)} + \max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(D)}). \end{array}$$

This UB is still not as good as R of Fig. 1. This is because we need the partial evaluation transformation that will be explained in the next section. If we consider the scope-based GC strategy $\mathcal{G}_s$ we obtain equations as the case of $\mathcal{G}_r$ except for the fourth equation which becomes:

$$\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + m_2^{ub}(a), ④, \mathcal{G}_s) + \mathrm{s(D)} = \mathrm{s(A)} + \mathrm{s(B)} + \sout{\mathrm{s(C)}} + \mathrm{s(E)} + \mathrm{s(D)}.$$

Unlike the case of reachability, here types s(A) and s(B) are not collected since they are created in m_1. The UB obtained for m_2 is equal to the one of case $\mathcal{G}_r$ and, for m_1, it is:

$$\hat{m}_1^{ub} = \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(E)} + \max(\mathrm{s(C)}, \mathrm{s(D)})$$

which is exactly S of Fig. 1. Regarding the case of the liveness-based strategy $\mathcal{G}_l$, we obtain also the same equation as the case of $\mathcal{G}_r$ except for the fourth and sixth equations which are as follows:

$$\begin{array}{lll} \hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + m_2^{ub}(a), ④, \mathcal{G}_l) + \mathrm{s(D)} & = \sout{\mathrm{s(A)}} + \sout{\mathrm{s(B)}} + \sout{\mathrm{s(C)}} + \sout{\mathrm{s(E)}} + \mathrm{s(D)} \\ \hat{m}_2(a) = \mathcal{A}(\mathrm{s(C)}, ③, \mathcal{G}_l) + \mathrm{s(E)} & = \sout{\mathrm{s(C)}} + \mathrm{s(E)}. \end{array}$$

In contrast to the case of $\mathcal{G}_r$ and $\mathcal{G}_s$, in the equation of m_1 we eliminate type E since it becomes dead at program point ④, and, in the equation of m_2 we eliminate C since it is already dead when reaching program point ③. The UBs obtained from these equations are:

$$\begin{array}{lll} \hat{m}_2^{ub}(a) & = & \max(\mathrm{s(C)}, \mathrm{s(E)}) \\ \hat{m}_1^{ub} & = & \max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(E)}, \mathrm{s(D)}). \end{array}$$

Again, this UB is less precise than L of Fig. 1, a partial evaluation transformation is needed in order to obtain the same one. In summary, we get the following UBs for the peak memory consumption of m_1 w.r.t scope-based, reachability-based and liveness-based GC strategies:

S	=	$\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(E)} + \max(\mathrm{s(C)}, \mathrm{s(D)})$
R	=	$\mathrm{s(E)} + \max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(D)})$
L	=	$\max(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(E)}, \mathrm{s(D)})$.

Recall that R and L are still not as precise as those of Fig. 1.

Informally, the following soundness theorem ensures that our analysis correctly approximates the peak of a procedure's execution for any GC scheme $\mathcal{G}$ in the two scenarios explained in Section 1.2.

Theorem 20 (*Soundness*)**.** *Let P be a program with an entry procedure p, $\mathcal{G}$ be the GC strategy, and $\hat{p}^{ub}(\bar{\mathrm{x}})$ an UB for the corresponding peak heap space equations generated in Definition 18. Assuming that we start the execution from an initial state $S_0 = \langle \bot, p(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle), tv_0 \rangle; h_0$ then, for any complete trace t, it holds $\hat{p}^{ub}(\bar{v}) \geq peak(t)$ where $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_0, h_0)$ if one of the conditions hold:*

(i) *$\mathcal{G}$ is applied as soon as objects become collectible; or*
(ii) *the heap size is fixed to $\hat{p}^{ub}(\bar{v})$ and $\mathcal{G}$ is applied when we reach this limit.*

Example 21. By applying Definition 18 to the first peak heap space equation generated for m in Example 4, and splitting the max operator, we obtain

$$\begin{array}{rcl}
\hat{m}(n) & = & \max(\mathrm{s}(\mathrm{Tree}^1), \\
& = & \quad \mathcal{A}(\mathrm{s}(\mathrm{Tree}^1), \textcircled{b}, \mathcal{G}) + \hat{g}(n), \\
& = & \quad \mathcal{A}(\mathrm{s}(\mathrm{Tree}^1) + g^{ub}(n), \textcircled{c}, \mathcal{G}) + \hat{f}(n, s_1), \\
& = & \quad \mathcal{A}(\mathrm{s}(\mathrm{Tree}^1) + g^{ub}(n) + f^{ub}(n, s_1), \textcircled{d}, \mathcal{G}) + \hat{m}(s_2), \\
& = & \quad \mathcal{A}(\mathrm{s}(\mathrm{Tree}^1) + g^{ub}(n) + f^{ub}(n, s_1) + m^{ub}(s_2), \textcircled{e}, \mathcal{G}) + \hat{m}(s_3)) \\
\hat{m}(n) & = & 0 \\
\hat{f}(n, l) & = & \max(\hat{f}_1(n, r), \mathcal{A}(f_1^{ub}(n, r), \textcircled{g}, \mathcal{G}) + \hat{f}_2(l, r')) \\
\hat{f}_1(i, r) & = & \max(\mathrm{s}(\mathrm{Long}^3), \mathcal{A}(\mathrm{s}(\mathrm{Long}^3), \textcircled{i}, \mathcal{G}) + \hat{f}_1(i', r')) \\
\hat{f}_1(i, r) & = & 0 \\
\hat{f}_2(l, r) & = & \max(\mathrm{s}(\mathrm{Integer}^4), \mathcal{A}(\mathrm{s}(\mathrm{Integer}^4), \textcircled{i}, \mathcal{G}) + \hat{f}_2(l', r')) \\
\hat{f}_2(l, r) & = & 0 \\
\hat{g}(n) & = & \max(\hat{g}(s_0), \mathcal{A}(g^{ub}(s_0), \textcircled{i}, \mathcal{G}) + \mathrm{s}(\mathrm{List}^2)) \\
\hat{g}(n) & = & 0.
\end{array}$$

These equations can be specialized to a specific garbage collection strategy, as we have done in Example 19, using the total UBs of Example 5 and the collectible information from Examples 8 and 10–12. After solving the resulting equations we get the following UBs for the cases of $\mathcal{G}_r$ and $\mathcal{G}_l$:

$$\begin{array}{ll}
\hat{m}^{ub}(n) & =(2^{\mathsf{nat}(n)} - 1) * \mathrm{s}(\mathrm{Tree}^1) + \max(\mathrm{s}(\mathrm{Long}^3), \mathrm{s}(\mathrm{Integer}^4)) + \mathsf{nat}(n) * \mathrm{s}(\mathrm{List}^2) \\
\hat{f}^{ub}(n) & =\max(\mathrm{s}(\mathrm{Long}^3), \mathrm{s}(\mathrm{Integer}^4)) \\
\hat{f}_1^{ub}(l, r) & =\mathrm{s}(\mathrm{Long}^3) \\
\hat{f}_2^{ub}(i, r) & =\mathrm{s}(\mathrm{Integer}^4) \\
\hat{g}^{ub}(n) & =\mathsf{nat}(n) * \mathrm{s}(\mathrm{List}^2).
\end{array}$$

and for the case of $\mathcal{G}_s$ the peak consumption is equal to the total allocation bound of Example 5 except for m which is:

$$\hat{m}^{ub}(n)=(2^{\mathsf{nat}(n)} - 1) * (\mathrm{s}(\mathrm{Tree}^1) + \mathsf{nat}(n) * \mathrm{s}(\mathrm{List}^2)) + \mathsf{nat}(n) * \max(\mathrm{s}(\mathrm{Long}^3), \mathrm{s}(\mathrm{Integer}^4)).$$

The difference between $\mathcal{G}_s$ and an ideal GC is that, in the latter, the memory required by g is accumulated only once to the memory requirement of m, while scope-based GC requires space for allocating g an exponential number of times. This is because the List^2 objects are created in g and become unreachable (resp. dead) in a different scope m (resp. f).

The above example illustrates the power of our method in the kind of upper bounds we infer: we capture exponential, logarithmic and polynomial memory bounds. This improves over type-based memory usage analyses [21] which are often restricted to linear upper bounds.

6. A partial evaluation transformation of recurrence relations

The technique in Section 5 obtains precise UBs when objects become collectible in the same rule in which they have been created. However, if an object becomes collectible in another rule (e.g., of a called method), the effect of removing it might be delayed until it becomes visible in the rule in which they have been created, which might result in a loss of precision. This happens in the program of Fig. 1, when reaching program point ③ in m_2, the object to which the variable "a" refers is dead, and the object to which "a.f" refers is both dead and unreachable. However, the equations that we generate for m_1 in Example 19 (both for $\mathcal{G}_l$ and $\mathcal{G}_r$) do not take advantage of this information but rather, they use the information that such objects are dead and unreachable at ④, i.e., only upon exit from m_2. This prevents us from obtaining the precise UBs R and L in Fig. 1.

The well-known technique of partial evaluation [22] (PE for short) gives us a leeway to solve this accuracy problem. PE is an automatic program transformation technique whose goal is to specialize programs by propagating static information by means of *unfolding*. In our context, the notion of unfolding corresponds to the intuition of replacing a call to a relation by the definition of the corresponding relation, and therefore merging the corresponding rules into the same equation and making more program points visible. Observe that there is no static data w.r.t. the program is going to be specialized. Hence, we apply a very simple form of partial evaluation whose main component is the unfolding operator.

Example 22. Let us consider again the program depicted in Fig. 1, and the GC strategy $\mathcal{G}_l$. Note that the objects to which the variable a and the field *a.f* point (i.e., those of type *A* and *B*), which are created in m_1, become dead at program point ③. In particular, they become dead before creating the object of type *E*. This means that, in order to get the optimal peak memory consumption w.r.t. $\mathcal{G}_l$, one should take into account that those objects are collected before creating *E*, and not just accumulating s(E) to the current consumption. This would allow stating that the consumption after creating *E* is $\max(\mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}), \mathrm{s}(\mathrm{E}))$ instead of $\mathrm{s}(\mathrm{A}) + \mathrm{s}(\mathrm{B}) + \mathrm{s}(\mathrm{E})$.

With the techniques developed so far, this is not possible because of the following reasons: (1) when generating the equations of m_1, we do not look into the program points of m_2, but only on what happens upon exit from m_2; and (2) when generating the equations of m_2, since the analysis is modular, we do not consider the caller and thus we cannot collect *A* and *B* before creating *E*, we even do not know how many objects of type *A* and *B* we have at that point.

As we have explained above, one way to solve this problem is to unfold m_2 inside m_1. This makes them belong to the same rule. Thus, when generating the equations of m_1 we can take into account that *A* and *B* are collected before creating *E*, since the program point ③ now belongs to the same rule in which *A* and *B* are created. It is important to note that this unfolding is not applied to the source program, but rather to the equations generated for the total consumption since they are the ones used to generate the peak consumption equations. In what follows we develop the details of this approach for the program of Fig. 1.

Consider the total memory allocation equations of m_1 and m_2 of Example 4. Unfolding the call to m_2 into its calling context results in the following equation:

$$m_1 = \mathrm{s(A)} + ①\mathrm{s(B)} + ②\mathrm{s(C)} + ③\mathrm{s(E)} + ④\mathrm{s(D)}.$$

From it, by applying Definition 18 and splitting the max into several equations, we obtain the following equations

$$\begin{array}{ll}
\hat{m}_1 = \mathrm{s(A)}, & = \mathrm{s(A)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)}, ①, \mathcal{G}_r) + \mathrm{s(B)} & = \mathrm{s(A)} + \mathrm{s(B)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)}, ②, \mathcal{G}_r) + \mathrm{s(C)} & = \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, ③, \mathcal{G}_r) + \mathrm{s(E)} & = \mathrm{s(A)} + \cancel{\mathrm{s(B)}} + \mathrm{s(C)} + \mathrm{s(E)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(E)}, ④, \mathcal{G}_r) + \mathrm{s(D)} & = \cancel{\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}} + \mathrm{s(E)} + \mathrm{s(D)}.
\end{array}$$

Solving this equation results exactly in the optimal UB R of Fig. 1. The key point is to incorporate the reachability information at program point ③ in the equation of m_1. We could not do it in Example 19 since it was in a different rule. Similarly, for $\mathcal{G}_l$ we obtain the following equations:

$$\begin{array}{ll}
\hat{m}_1 = \mathrm{s(A)}, & = \mathrm{s(A)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)}, ①, \mathcal{G}_l) + \mathrm{s(B)} & = \mathrm{s(A)} + \mathrm{s(B)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)}, ②, \mathcal{G}_l) + \mathrm{s(C)} & = \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}, ③, \mathcal{G}_l) + \mathrm{s(E)} & = \cancel{\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)}} + \mathrm{s(E)} \\
\hat{m}_1 = \mathcal{A}(\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(E)}, ④, \mathcal{G}_l) + \mathrm{s(D)} & = \cancel{\mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(E)}} + \mathrm{s(D)}.
\end{array}$$

Solving this equation results exactly in the optimal UB L of Fig. 1. Note that UBs of Fig. 1 are optimal, i.e., they describe the exact peak memory consumption w.r.t. the corresponding GC strategy.

The unfolding process could be defined on the programming language, however, defining it on the RR has the main advantage of being much simpler. This is because RR are made up only of constants s(C), calls to other equations and linear constraints. This kind of unfolding is basically the same as that of clauses in constraint logic programming [16] and that of the UB solver of [1].

Definition 23 (*Unfolding Step*)**.** Let *E* be the recurrence equation "$p(\bar{x}) = b_1 + \cdots + b_{i-1} + q(\bar{x}_i) + b_{i+1} + \cdots + b_n, \varphi$", and E' a renamed apart equation $q(\bar{y}) = c_1 + \cdots + c_m, \varphi'$" defining q such that $vars(E) \cap vars(E') = \emptyset$. Then, the unfolding of *E* w.r.t. $q(\bar{x}_i)$ and E' is "$p(\bar{x}) = b_1 + \cdots + b_{i-1} + c_1 + \cdots + c_m + b_{i+1} + \cdots + b_n, \varphi \wedge \varphi' \wedge \{\bar{x}_i = \bar{y}\}$".

The unfolding step basically generates a new equation by: substituting the (renamed apart) definition of q in its calling site, joining the constraints of both p and q ($\varphi \wedge \varphi'$), and unifying the variables of the caller and the variables of the renamed apart definition ($\{\bar{x}_i = \bar{y}\}$). When the call we want to unfold is defined by several equations, the above operation is repeated for each of them, possibly generating several equations. When $\varphi' \wedge \varphi \wedge \{\bar{x}_i = \bar{y}\}$ is unsatisfiable, no equation is generated since this does not correspond to a valid execution.

Note that when we unfold an equation, we lose the scope in which the corresponding objects were created. This in turn might result in a loss of precision when eliminating the collectible information for a procedure. Fortunately, this can be easily solved by symbolically keeping the scope boundaries in order to know where the objects come from (since our bounds are symbolic it is possible to do so). We can then slightly modify Definition 15 to handle this syntax. For example, the unfolded rule for m_1 would be:

$$m_1 = \mathrm{s(A)} + ①\mathrm{s(B)} + ②m_2[\mathrm{s(C)} + ③\mathrm{s(E)}] + ④\mathrm{s(D)}$$

where $m_2[\cdots]$ is used to syntactically keep the scope boundaries of the unfolded procedure. For simplicity, in what follows we ignore this extension.

Performing an unfolding step solves this precision problem when the object is created in a procedure p and becomes collectible in the unfolded procedure q which is called from p. However, there are scenarios where more steps are required. For example, an object might become collectible not during an immediate call but rather in a transitive one. Even more, an object can be created and become collectible in procedures that do not have a caller/callee relation. In general, unfolding steps should be applied repetitively until the program points that correspond to the creation and collection of an object are as close as possible in the equations. This process in the presence of recursive relations (coming from loops) might be non-terminating. Fortunately, the problem has been well studied in the PE field and we can adopt any terminating strategy [24].

For instance, in [1], the strategy is to leave one relation per *recursive* strongly connected component (SCC) and unfold the remaining ones. Note that the purpose of unfolding recursive relations is to transform mutual recursion into direct recursion, when possible, by eliminating all relations but one in the given SCC. In order to take more advantage of collectible classes, it is even possible to unfold a recursive SCC into other SCCs, which corresponds to loop unrolling. In PE terminology, a *binding-time analysis* is an analysis which is used to infer a set of predicates which cannot be unfolded, either because it could endanger termination or because it would not be profitable (e.g., it would not make the points of interest closer in the rules). The set of predicates resulting from such analysis is a *binding-time annotation* (BT) Our definition of partially evaluated equations can be used with any BT which ensures termination.

Definition 24 (*Partially Evaluated RR*)**.** Given a set of RRs and a BT, the *partially evaluated RRs* are obtained by iteratively unfolding (Definition 23) all calls in the rhs of the equations that do not belong to the set of predicates in BT w.r.t. its defining equations.

Once the RR have been partially evaluated, we apply Definition 18 to them in order to generate the corresponding peak heap space equations.

Example 25. For the simple example of Fig. 1, if the BT includes only m_1, we obtain the RR in Example 22 and the optimal R and L UBs of Fig. 1.

Theorem 26 (*Soundness*)**.** *Let P be a program with an entry procedure p, $\mathcal{G}$ the used GC technique, and $\hat{p}^{ub}(\bar{x})$ an UB for the corresponding peak heap space equations generated in Definition 18 after PE. Assuming that we start the execution from an initial state $S_0 = \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$ then, for any complete trace t, $\hat{p}^{ub}(\bar{v}) \geq peak(t)$ under the same conditions as in Theorem 20 where $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$.*

The key difference with the PE of our previous work [1] is that we apply PE *previously* to the generation of the peak heap space RR, while [1] uses PE only to solve them. This is an essential difference since we would not be able to obtain the propagation of collectible information that we need to obtain peak heap space bounds by using PE like [1]. As other differences, we can apply PE with any terminating BT while [1] requires checking further conditions on the associated graph.

7. Experimental evaluation

We have implemented our technique in COSTA [4], a cost and termination analyzer for Java bytecode. Our implementation can be tried out through its web interface at http://costa.ls.fi.upm.es by selecting the *memory* cost model. The system allows selecting the GC strategy among this set of options: *none*, *scope*, *reachability* and *liveness*, which respectively correspond to the total memory consumption and the peak consumption with $\mathcal{G}_s$, $\mathcal{G}_r$ or $\mathcal{G}_l$. Our reachability analysis is based on the context-insensitive points-to analysis of [38], and the heap liveness analysis is similar to the region-based liveness of [12]. The PE transformation leaves one relation per SCC as explained in Section 6, and the obtained *CRs* are solved using the PUBS *CRs* solver [1].

We assess the practicality of our proposal on the standardized set of benchmarks in the JOlden suite [11]. Indeed, the benchmarks used in the experiments have been slightly modified w.r.t. the original versions in order to avoid programming patterns like enumerators that require field-sensitive analysis which makes the overall cost analysis around twice more expensive (see [2]). Also, in few methods whose cost depends on the size of an array which is a field of an object, we had to explicitly pass such array as method argument. This is because the size abstraction used in COSTA, path-length, would lose this information. A more accurate size abstraction like the one used in [29] could handle these cases automatically. The modified benchmarks are available from the COSTA website above.

The JOlden benchmark suite was first used in [13] in the context of memory usage verification for a different purpose, namely for checking memory adequacy w.r.t. given specifications, but there is no inference of UBs as our analysis does. It has been also used in [10] for our same purpose, i.e., the inference of peak consumption. However, since [10] does not deal with memory-consuming recursive methods, the process is not fully automatic in their case and they have to provide manual annotations. Also, they require invariants which sometimes have to be manually provided. In contrast, our tool is able to infer accurate peak heap UBs in a fully automatic way, including logarithmic and exponential complexities.

Table 1 shows the UBs obtained by COSTA on a set of selected methods of the JOlden suite. The symbol "=" in Table 1 indicates that the corresponding UB is the same as the one in the cell to its left. Each row in the table corresponds to the results of one method. The first column indicates the package and class to which the method belongs and its name in the syntax "package/class.method" (with some abbreviations due to space limitations). For each method, we infer the total allocation UB (column $\mathbf{U}_\mathsf{T}$) and the peak heap consumption using the different GC strategies (column $\mathbf{U}_\mathsf{S}$ for $\mathcal{G}_s$, $\mathbf{U}_\mathsf{R}$ for $\mathcal{G}_r$, and $\mathbf{U}_\mathsf{L}$ for $\mathcal{G}_l$). The parameters in the UBs are abbreviations of the original variable or field names they represent, or `t`, that represents the `this` object. Note that $\mathbf{U}_\mathsf{T} \geq \mathbf{U}_\mathsf{S} \geq \mathbf{U}_\mathsf{R} \geq \mathbf{U}_\mathsf{L}$ holds since (i) the total consumption is clearly larger than the peak consumption under any GC strategy; and (ii) clearly $\mathcal{G}_l$ is more general than $\mathcal{G}_r$ which is more general than $\mathcal{G}_s$. We have considered all methods that consume memory that are invoked within the method `main` of the main class of each package in the JOlden. In general, this requires the analysis of the majority of the methods in the corresponding package. Method `parseCmdLine` is common in most packages and therefore it has only been considered once (last row). We have preferred to analyze the invoked methods within the `main`'s instead of analyzing the `main`'s themselves because: (1) In

Table 1
Total, scope, reachability and liveness upper bounds by COSTA on the JOlden.

Method	U_T	U_S	U_R	U_L
bh/Tree.<init>	4	=	=	=
bh/Tree.createTestData	24*nat(nbody) + 32	=	=	=
bh/Tree.stepSystem	not available	=	=	=
bh/MathVector.toString	8	=	2	=
bi/Value.createTree	3*nat(size-1)	=	=	=
bi/Value.inOrder	$4*2^{nat(t-1)} - 2$	2*nat(t-1) + 2	2	2
em3d/BiGraph.create	2*nat(nN-1)*nat(nD) + 2*nat(nN) + 14*nat(nN-1) + 2*nat(nD) + 19	=	=	=
em3d/Em3d.compute	4*nat(numIter)	=	=	=
em3d/BiGraph.toString	4*nat(t-1) + 2	=	4	=
hth/Village.createVill	$26/3*(4^{nat(level)} - 1)$	=	=	=
hth/Village.simulate	not available	=	=	=
hth/Village.getResults	$28/3*4^{nat(t)-1} - 7/3$	=	=	=
mst/Graph.<init>	$4*nat(nv)^2$ + nat(nv)*nat(nv/4) + 6*nat(nv)	=	=	=
mst/MST.computeMST	4*nat(nv-1)	=	4+2*nat(nv-1)	2
per/QTN.createTree	$28/3*4^{nat(levels)} - 7/3$	=	=	=
pow/Root.<init>	nat(nf)*(6*nat(nl)*nat(nb)*nat(nlv) + 9*nat(nl)*nat(nb) + 6*nat(nl)*nat(nlv) + 6*nat(nb)*nat(nlv) + 22*nat(nl) + 9*nat(nb) + 6*nat(nlv) + 23) + 4	=	=	=
pow/Root.toString	2	=	=	=
pow/Lateral.compute	$8*nat(t-3)^2*nat(nl)$ + 16*nat(t-3)*nat(nl) + 8*nat(nl)	8	=	=
tadd/TreeNode.<init>	$10*2^{nat(levels-1)} - 6$	=	=	=
tsp/Tree.buildTree	13*nat(2*n - 1)	7*nat(2*n - 1)	=	=
tsp/Tree.printVO	2*nat(t-1) + 4	=	4	=
vor/Vertex.createPs	5*nat(2*n - 1)	=	=	=
vor/Vertex.buildDelTr	not available	=	=	=
vor/Edge.outputVorDiag	not available	=	=	=
vor/Vertex.print	$4*2^{nat(t-1)} - 2$	2*nat(t-1) + 2	2	2
parseCmdLine	nat(args) + 4	=	=	4

many cases we obtain more interesting and useful UBs which are fully parametric. The `main` instead performs some calls with constant values thus producing less interesting UBs. For instance, this is the case of the call to `Root.<init>` within `pow.Power.main`. Method `Root.<init>` has four parameters all of which influence its memory consumption (see the UB in Table 1). However this parametricity is lost when analyzing `pow.Power.main` since the call to `Root.<init>` is performed with constant values (thus obtaining a constant UB for it). (2) This way it is clearer to understand the obtained UBs. Otherwise, expressions possibly involving parameters of different methods are all mixed in the same UB. (3) As mentioned below, there are some methods that COSTA cannot handle due to limitations unrelated to our analysis. This way we can concentrate on those methods which can be handled.

It can be observed that the obtained UBs are all numerical. This is done by substituting the symbolic expressions s(C) by some numeric measure. In particular, for objects we consider the number of fields C has, and for arrays we consider their sizes. This way the *CRs* solver of COSTA can perform mathematical simplifications and therefore we can get readable UBs. In addition to this measure, COSTA allows counting the number of objects and/or arrays, and number of bytes, as well as obtaining symbolic UBs with the concrete s(C) expressions. For instance, the symbolic UB for the total memory consumption of method `tsp/Tree.buildTree` is `(2*nat(n)-1)*(s(Tree) + 2*s(Random))`. Since class `Tree` has 7 fields and `Random` has 3, the expression is simplified to `13*nat(2*n - 1)`. Let us observe that the symbolic UB for the peak consumption with $\mathcal{G}_S$ does not include the `Random` objects and therefore we have `7*nat(2*n - 1)`. The UB obtained by the solver of COSTA is less precise than the one shown in the table. Only in this case, we have manually solved the equations in order to illustrate the real gains we can achieve. Obtaining this solution automatically requires more sophisticated techniques for solving recurrences.

Out of the twenty-six methods that have been considered, which in turn required the analysis of more than two-hundred methods, there are four methods for which COSTA cannot produce an UB. Method `bh/Tree.stepSystem` contains a loop whose termination condition does not depend on the size of the data structure, but rather on the particular value stored at certain locations within the data structure. In general, it is complicated to bound the number of iterations of this kind of loop. A similar behavior also occurs in `vor/Vertex.buildDelTr` and `vor/Edge.outputVorDiag`. In `hth.Village.simulate`, COSTA fails to estimate the path-length of the traversed data structure since it passes through fields of type array. The reason for this failure is that arrays are abstracted to their length (number of elements), and therefore once we pass through an array we lose the corresponding path-length information. This is easily solvable by maintaining an additional abstraction for arrays which approximates their path-length. It should be noted that, in these four cases, the

Table 2
Analysis times of COSTA on the benchmarks of Table 1.

Method	T_T	T_S	T_R	T_L
bh/Tree.<init>	0.110	0.112 (0.010)	0.112 (0.013)	0.117 (0.014)
bh/Tree.createTestData	9.104	9.145 (0.241)	27.945 (0.340)	27.832 (0.340)
bh/Tree.stepSystem	-	-	-	-
bh/MathVector.toString	0.283	0.286 (0.009)	0.317 (0.012)	0.317 (0.012)
bi/Value.createTree	0.190	0.193 (0.033)	0.246 (0.035)	0.246 (0.034)
bi/Value.inOrder	0.248	0.255 (0.011)	0.460 (0.013)	0.465 (0.013)
em3d/BiGraph.create	0.449	0.458 (0.042)	0.477 (0.051)	0.476 (0.049)
em3d/Em3d.compute	0.420	0.424 (0.020)	0.496 (0.029)	0.496 (0.028)
em3d/BiGraph.toString	0.287	0.287 (0.018)	0.341 (0.023)	0.342 (0.024)
hth/Village.createVill	0.697	0.704 (0.082)	1.040 (0.111)	1.043 (0.110)
hth/Village.simulate	-	-	-	-
hth/Village.getResults	1.551	1.627 (0.035)	4.506 (0.037)	4.517 (0.037)
mst/Graph.<init>	1.604	1.615 (0.071)	1.897 (0.098)	1.931 (0.100)
mst/MST.computeMST	3.557	3.564 (0.072)	9.195 (0.118)	9.198 (0.115)
per/QTN.createTree	2.439	2.448 (0.095)	5.803 (0.141)	5.727 (0.141)
pow/Root.<init>	0.755	0.757 (0.139)	0.848 (0.204)	0.840 (0.200)
pow/Root.toString	1.962	2.049 (0.115)	2.479 (0.154)	2.530 (0.158)
pow/Lateral.compute	0.170	0.176 (0.009)	0.175 (0.012)	0.177 (0.012)
tadd/TreeNode.<init>	0.132	0.133 (0.021)	0.139 (0.019)	0.138 (0.019)
tsp/Tree.buildTree	10.586	10.590 (0.083)	15.186 (0.101)	15.142 (0.103)
tsp/Tree.printVO	1.473	1.477 (0.045)	2.404 (0.040)	2.411 (0.039)
vor/Vertex.createPs	0.363	0.364 (0.047)	0.588 (0.055)	0.587 (0.055)
vor/Vertex.buildDelTr	-	-	-	-
vor/Edge.outputVorDiag	-	-	-	-
vor/Vertex.print	0.461	0.464 (0.014)	1.630 (0.017)	1.629 (0.017)
parseCmdLine	0.665	0.667 (0.025)	0.928 (0.033)	0.879 (0.033)

limitations are not related to the analysis presented in this paper but to external components which can be independently improved.

In order to assess the precision of our analysis we have calculated manually closed-form UBs of the worst-case cost for all methods (except for those in which COSTA fails) for the different GC strategies. The main conclusion is that the results obtained by our analysis are very accurate. Out of a total of 88 configurations (4 configurations for each of the 22 methods), COSTA is fully precise, i.e., it infers exactly the same UB manually, in 83 configurations. In the other 5 cases, the source of imprecision is not related to our analysis but rather to external components (like the *CRs* solver used to obtain a closed-form UB from our equations, or the path-length abstraction) as we explain below:

- Total allocation for method `pow/Lateral.compute`. The UB obtained manually is $\Sigma_{i=1}^{t-3}$`i*nlv*8`, whereas COSTA obtains the UB `8*nat(t-3)^2*nat(nlv) + 16*nat(t-3) * nat(nlv) + 8*nat(nlv)`. The imprecision is due to the path-length abstraction [32] used in the underlying size analysis of COSTA.
- Method `tsp/Tree.printVO`. For both total allocation and heap consumption w.r.t. $\mathcal{G}_s$, the manually obtained UB is `2*nat(t-1) + 2`, whereas COSTA obtains the UB `2*nat(t-1) + 4`. The imprecision is negligible, it is due to way the *CRs* solver handles base-cases.
- Total allocation for method `tadd/TreeNode.<init>`. The manually obtained UB is $6*2^{\texttt{nat(levels-1)}}$, whereas COSTA obtains $10*2^{\texttt{nat(levels-1)}} - 6$. The imprecision is again due to the *CRs* solver, i.e., the generated equations precisely capture the memory consumption, however, the *CRs* solver fails to obtain the exact UB.
- Total allocation for method `vor/Vertex.print`. The UB obtained manually is $2*2^{\texttt{nat(t-1)}}$, whereas COSTA obtains $4*2^{\texttt{nat(t-1)}} - 2$. The imprecision is again due to the *CRs* solver (as in the above case).

Note that, the fact that in many cases there are no variations between the total and the peak consumption is because in those cases there are no temporary objects. This is the case of most methods whose purpose is to initialize the data structures that are later used by the benchmarks, namely all constructors (named `<init>`) and all methods whose name includes the word "create" or "build".

A final interesting point to observe is that the majority of `main` methods start by invoking method `parseCmdLine`. Let E be the UB of the memory consumed by the rest of the calls. Then, the total memory consumption of method `main` is "`nat(args) + 4 + E`". Since the objects created inside `parseCmdLine` do not escape from their scope, the peak consumption inferred by COSTA w.r.t. $\mathcal{G}_s$ is "`max(nat(args) + 4, E)`". Interestingly, if we consider the $\mathcal{G}_l$ strategy the peak consumption is simply "E", i.e., it does not depend anymore on parameter *args*.

Table 2 shows the total runtimes of COSTA for inferring the UBs of Table 1. The experiments have been performed on a MacBook Air with 1.6 GHz dual-core Intel Core i5 processor and 4 GB memory. All runtimes that appear in the table are in *seconds*, and they are calculated as the average runtime of 5 executions of the corresponding benchmark. Columns $\mathbf{T}_T$,

$\mathbf{T}_S$, $\mathbf{T}_R$ and $\mathbf{T}_L$ show, respectively, the total runtime taken by COSTA for inferring the total allocation UB, and the peak heap consumption UB using the GC strategies $\mathcal{G}_s$, $\mathcal{G}_r$, and $\mathcal{G}_l$. Columns $\mathbf{T}_S$, $\mathbf{T}_R$ and $\mathbf{T}_L$ include also in parenthesis the runtime for inferring the corresponding collectible types, which is done by using escape, reachability and liveness analysis respectively. Clearly, the later runtimes are negligible when compared to the total runtime. Note that total runtime in each column includes the runtime of several static analyses and program transformations [4] applied by COSTA, e.g., transforming the bytecode to the intermediate representation, size analysis, etc.

There are 4 benchmarks in which the runtimes in columns $\mathbf{T}_S$, $\mathbf{T}_R$ and $\mathbf{T}_L$ are significantly higher than the one in column $\mathbf{U}_T$. This is because in these cases the number of *peak heap space equations* is much larger than the number of *total heap space equations*. Recall that according to Definition 18 every total heap space equation induces several peak heap space equations, depending on the number of calls and memory allocation instructions in the corresponding rule. For the case of $\mathcal{G}_s$, Definition 18 is optimized in the implementation to consider only program points that correspond memory can be reclaimed only upon exit from methods. This way we generate less equations than in the case of $\mathcal{G}_r$ and $\mathcal{G}_l$. This is why the $\mathbf{T}_S$ is smaller that $\mathbf{T}_R$ and $\mathbf{T}_L$.

8. Related work

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [39]). Analysis of live heap space is different because it involves explicit analysis of all program states. Most of the work of memory estimation has been studied for functional languages. The work in [20] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager.

The techniques proposed by Unnikrishnan et al. [35,34] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form UBs, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations. Moreover, the problem of inferring the peak heap space is different from memory prediction in functional languages since, due to the absence of mutable data structures, GC can be modeled in a scope-based fashion where the scopes are determined by the corresponding function definitions.

For imperative object-oriented languages, related techniques have been recently proposed. For an assembly language, [14] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided, which is similar to our reliance on types. For a Java-like language, the approach of [10] infers UBs of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach is not tied to a particular GC model and computes accurate bounds for exponential, logarithmic, etc. complexities (unlike [10,14]) and it is fully automatic (unlike [21]). Other approaches to resource usage analysis are developed to measure other types of resources, namely [18] predicts number of instructions, [3] is generic in the definition of cost model but neither of them supports peak memory consumption, since the underlying techniques are developed to measure accumulative resources, while memory usage is a resource that increases and decreases along the execution. Other work, such as [13], provides a framework for checking that the memory usage conforms to user-supplied specifications. User-supplied specifications may be hard to provide and are likely to be impractical for bytecode programs.

There are a number of improvements and extensions that can be incorporated in order to increase precision our heap space analysis, but which do not require any modification to the formal framework. An important extension is to consider a field-sensitive analysis. When data is stored in the heap, such as in object fields (numeric or references), keeping track of their value during static analysis becomes rather complex and computationally expensive. Analyses which keep track (resp. do not keep track) of object fields are referred to as *field-sensitive* (resp. *field-insensitive*). A resource analysis can be made field-sensitive by applying techniques as those developed in [17,28,26,2].

Another source is imprecision is when programs traverse *cyclic data structures*. The problem is mainly due to the difficulty in bounding the number of loop iterations. Consider the loop `while(x.data != e) x = x.next;` and assume that `x` points to a cyclic linked list. In order to bound the number of iterations, one needs to (1) verify that there is an element equal to e in `x`; (2) verify that the loop will eventually visit all the elements; and (3) bound the number of elements in the data structure. The difficulty lies in verifying (1) and (2), since they require under-approximations. Also, there is imprecision due to the over-approximation applied by the analyses which infer sharing, acyclicity, and constancy information (e.g., the analysis can infer that a variable might point to a cyclic data-structure while in practice it does not). One can develop more precise analyses for inferring such properties and overcome precision problems at the price of performance.

9. Conclusions and future work

We have presented a general approach to automatic and accurate live heap space analysis for garbage-collected languages. As a first contribution, we propose how to obtain accurate bounds on the active memory at a program point, by combining the total allocation performed up to this point together with information inferred on the object *lifetimes*, i.e., an approximation of the set of objects which can be garbage collected at that point. Then, we introduce a novel form of *peak consumption cost relation* which uses the computed active memory bounds and precisely captures the actual heap consumption of programs' execution for garbage-collected languages. Such cost relations can be converted into closed-form UBs by relying on existing UB solvers [1]. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection.

Finally, the memory consumption for executing a program typically include both the heap and the frame stack usage. This paper focuses on the heap space because estimating the maximal height of the frames stack from our heap analysis is straightforward. In particular, given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \leftarrow g, b_1, \ldots, b_n$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b_{i_j}} = \mathrm{q_{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \max(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_1}(\bar{x}_{i_k})) \;\; \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replaces their corresponding 1's by 0's. This is a subject for future work.

Acknowledgments

We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped us to improve this article. Preliminary versions of this work appeared in the Proceedings of ISMM'07 [6], ISMM'09 [7] and ISMM'10 [8].

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

Appendix. Proofs of theorems

We start by defining an operational semantics for the *CRs*. This semantics is later used to: (1) formally define the meaning of an upper bound for a *CR*; and (2) base the proofs on relating program traces to *CRs* traces.

A *CRs* state is a pair $\langle b_1 \cdots b_n, e \rangle$. The first component consists of the expressions $b_1 \cdots b_n$ that have to be evaluated. Each b_i is either a call of the form $p(\bar{v})$ where $\bar{v}$ are integer values, or a cost expression that does not involve free variables (arithmetic expression over terms of the form s(C)). The second component e is a cost expression that refers to the cost accumulated so far. Note that e does not have variables, i.e., $vars(e) = \emptyset$. The operational semantics consists of the following two rules:

$$(1)\ \frac{p(\bar{x}) = b_1 + \cdots + b_n, \varphi \in P, \;\; \sigma \models \bar{x} = \bar{v} \wedge \varphi}{\langle p(\bar{v}) \cdot bb, e \rangle \rightarrow \langle b_1\sigma \cdots b_n\sigma \cdot bb, e \rangle} \qquad (2)\ \frac{e' \text{ is a cost expression}}{\langle e' \cdot bb, e \rangle \rightarrow \langle bb, e + e' \rangle}$$

Rule (1) is for the case of evaluating a call to a *CR*. It first looks for a matching equation and a satisfying assignment σ for the constraint $\bar{x} = \bar{v} \wedge \varphi$ (i.e, choosing values for all variables in the constraint such it is satisfied), and then adds the ground instances of all $b_1, \ldots, b_n$ to the sequence of expressions to be evaluated (the ground instance of b_i is $b_i\sigma$, which replaces the variables in b_i by their values from σ). Rule (2) is for the case of accumulating a (ground) cost expression, which simply adds it to the total cost. An execution in this setting starts from an initial state $\langle p(\bar{v}), 0 \rangle$, and, if it terminates, it ends in a final state $\langle \epsilon, e \rangle$. We use $\rightarrow^*$ for the transitive relation of $\rightarrow$. We refer to such execution as *CR* traces. Note that in the first rule, choosing an equation and a satisfying assignment are non-deterministic choices. Therefore, for an initial state $\langle p(\bar{v}), 0 \rangle$ we might have many *CR* traces (possibly infinite).

Definition 27 (*Upper Bound*)**.** A closed-form function $p^{ub}(\bar{x})$ is an upper bound for a *CR* p, if for any $\langle p(\bar{v}), 0 \rangle \rightarrow^* \langle bb, e \rangle$ it holds that $p^{ub}(\bar{v}) \geq e$.

Proof of Theorem 6. The first theorem is an immediate consequence of the correctness of the cost analysis framework of [3], for the memory consumption cost model $\mathcal{M}$ of Definition 3. In what follows we describe what the correctness of [3] guarantees. These guarantees are also used later in the proofs of the other theorems. As notation, given a heap h, we denote by $|h|$ the cost expression that describes the contents of the heap h, excluding those that were already in the initial configuration's heap, e.g., an expression of the form $10 * \mathrm{s}(\mathrm{C}^1) + 16 * \mathrm{s}(\mathrm{D}^2)$ where C^1 and D^2 are types (a class annotated with allocation site). Recall that we have assumed that objects in the initial heap are not collectible during the execution. □

Lemma 28 (*Guarantees of [3]*). *Let P be a program with an entry procedure p, and $t \equiv S_0 \rightsquigarrow^* S_n$ a (possibly incomplete) trace such that $S_0 \equiv \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$. Let $0 < i_1 < \cdots < i_k \leq n$ be k indexes of all states in t in which the next instruction to be executed is a memory allocation instruction, i.e., $S_{i_j} \equiv \langle q_{i_j}, \mathsf{new}\ C_{i_j} \cdot bc_{i_j}, tv_{i_j} \rangle \cdot A_{i_j}; h_{i_j}$. Then, using the total memory allocation CR of P, as defined in Definition 3, we can construct a CR trace $t^\alpha \equiv R_0 \rightarrow^* R_{n'}$ where $k \leq n' \leq n$ such that:*

1. *$R_0 \equiv \langle p(\bar{v}), 0 \rangle$ such that $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$;*
2. *There exists exactly k CR states of the form $\langle \mathrm{s(C)} \cdot bb, e \rangle$ in t^α with indexes $0 < r_1 < \cdots < r_k \leq n'$ such that: $R_{r_j} \equiv \langle \mathrm{s(C_{r_j})} \cdot bb, e_{r_j} \rangle$, $C_{r_j} \equiv C_{i_j}$ where C_{i_j} is the class in "$\mathsf{new}\ C_{i_j}$" from state S_{i_j}, and $e_{r_j} = |h_{i_j}|$. Moreover, the program point associated with $\mathrm{s(C_{r_j})}$ in the CR is the program point that corresponds to the instructions $\mathsf{new}\ C_{i_j}$.*

Intuitively, given a program trace t with an initial state $S_0 \equiv \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$, then it is guaranteed that its total memory consumption behavior can be simulated using the corresponding total memory allocation *CRs* starting from $R_0 \equiv \langle p(\bar{v}), 0 \rangle$ with $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$. This clearly implies that an upper bound, as defined in Definition 27, for the *CR* p is also an upper bound on the total memory consumption of the program when starting from S_0.

Proof of Theorem 20. In what follows we assume that we have a sound set of collectible types for all program points and methods of P. We use existing techniques to infer them before starting our analysis, and therefore proving their soundness is out of the scope of this article. We start with the proof of the theorem for condition (i), namely, when "*objects are collected as soon as they become collectible*". This is equivalent to applying $\mathcal{G}$ in each execution step, i.e, if we are in a state S_i, we apply $\mathcal{G}$ on the heap of S_i, and then we execute the instruction that leads to S_{i+1}. The proof is done in several steps as follows:

1. We define a $\mathcal{G}$-aware *CR* semantics such that for a given program trace t, where $\mathcal{G}$ is applied in every execution step, $peak(t)$ is approximated by a $\mathcal{G}$-aware *CR* trace using only the *total allocation* equations; and
2. We show that the $\mathcal{G}$-aware *CR* traces, using the *total allocation* equations, are isomorphic to those of the *CR* traces, using the *peak* equations.

Point (2) implies that an upper bound for the entry procedure p, when considering the peak equations, is also an upper bound for any $\mathcal{G}$-aware *CR* trace. Then, using point (1), we conclude that it is also an upper bound for $peak(t)$, which is exactly what want to prove.

The results of Lemma 28, which guarantees that the total memory consumption of t can be simulated using the total allocation equations, implies that the peak consumption can also be simulated at the level of the corresponding t^α (of Lemma 28). Simulating $\mathcal{G}$ at the level of t^α means: at some program points of interest, we remove collectible types (which are sound) from the cost expression that is accumulated so far, and then we move to the next state. In what follows we provide a possible way of simulating $\mathcal{G}$ at the level of the total equation.

Given a program P with an entry p, its total allocation equations $\mathcal{S}$, and a closed-form upper bound function $q^{ub}(\bar{x})$ for each *CR* $q \in \mathcal{S}$, we define the $\mathcal{G}$-aware *CR* semantics by means of the following rules:

$$(\mathcal{G}{-}1)\ \frac{q(\bar{x}) = b_1 + \cdots + b_n, \varphi \in P,\ \ \sigma \models \bar{x} = \bar{v} \wedge \varphi,\ \ 1 \leq j \leq n, e' = \mathcal{A}(b_1 + \cdots + b_{j-1}, [k, j-1], \mathcal{G})\sigma}{\langle q(\bar{v}), e \rangle \rightarrow_{\mathcal{G}} \langle e' \cdot b_j\sigma, e \rangle}$$

$$(\mathcal{G}{-}2)\ \frac{e' \text{ is a cost expression}}{\langle e' \cdot bb, e \rangle \rightarrow \langle bb, e + e' \rangle}.$$

First observe that in order to compute the peak consumption of t, it is enough to consider only the size of the heaps in the states of t in which the memory increases, i.e., those states that allocate memory. The $\mathcal{G}$-aware *CR* semantics aims at simulating all execution paths up to those states. The idea of the $\mathcal{G}$-aware *CR* semantics is that when executing a call $q(\bar{v})$ we have to consider all execution paths that lead to a cost expression $s(C)$ that corresponds to a "new C" instruction. Rule $(\mathcal{G}{-}1)$ considers all such possibilities as follows: for each $1 \leq j \leq n$, it assumes that $b_1, \ldots, b_{j-1}$ has been executed and therefore it accumulates its cost e' and continues with b_j. Note that $b_1, \ldots, b_{j-1}$ are not executed, but rather we use the pre-computed (sound) closed-form upper bounds on their total memory allocation, which is an over approximation of their real execution. Moreover, instead of accumulating the total cost of $b_1, \ldots, b_{j-1}$, it accumulates only the memory that has been created during their execution and is still active at that program point, i.e., $\mathcal{A}(b_1 + \cdots + b_{j-1}, \textcircled{j}, \mathcal{G})$. Note that $\mathcal{A}(b_1 + \cdots + b_{j-1}, \textcircled{j}, \mathcal{G})$ is sound due to the soundness of the total allocation upper bounds and that of the collectible sets. Intuitively, any prefix of t that ends in a state S_i, has a corresponding $\mathcal{G}$-aware *CR* trace $\langle p(\bar{v}), 0 \rangle \rightarrow^*_{\mathcal{G}} \langle s(C_i), e_i \rangle$ such that $e_i \geq |h_i|$ where h_i is the heap of state S_i and $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$. Then, $max(\{s(C_i) + e_i \mid \langle p(\bar{v}), 0 \rangle \rightarrow^*_{\mathcal{G}} \langle s(C_i), e_i \rangle\})$ is an upper bound for $peak(t)$. □

Lemma 29. *Let P be a program with an entry procedure p, $\mathcal{G}$ a GC strategy, and $t \equiv S_0 \rightsquigarrow^* S_n$ a (possibly incomplete) trace such that $S_0 \equiv \langle \bot, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle), tv_0 \rangle; h_0$, where $\mathcal{G}$ is applied in every state. Then, for any $S_i = \langle q_i, \mathsf{new}\ C_i \cdot bc_i, tv_i \rangle \cdot A; h_i$, there exists a $\mathcal{G}$-aware CR trace $\langle p(\bar{v}, 0) \rangle \rightarrow^*_{\mathcal{G}} \langle s(C_i), e \rangle$ such that $e \geq |h_i|$ where $\bar{v} = \alpha(\bar{x}, tv_0, h_0)$.*

Now we move to step 2, which relates the $\mathcal{G}$-aware *CR* traces that use the total allocation equations, to those *CR* traces that use the peak consumption equations.

Lemma 30. *Let $\mathcal{S}_1$ be a total allocation CRs of a program P, and $\mathcal{S}_2$ its corresponding peak CRs. Then, $R_0 \rightarrow^*_{\mathcal{G}} R_n$ is a $\mathcal{G}$-aware CR trace using $\mathcal{S}_1$, iff $\hat{R}_0 \rightarrow^* \hat{R}_n$ is a CR trace using $\mathcal{S}_2$, where $\hat{R}_i$ is obtained from R_i by replacing each call $q(\bar{v})$ by $\hat{q}(\bar{v})$.*

In order to prove this lemma, we show that from two isomorphic $\mathcal{G}$-aware *CR* state $\langle p(\bar{v}), e\rangle$ and *CR* state $\langle \hat{p}(\bar{v}), e\rangle$, we can move to isomorphic states $\langle e' \cdot q(\bar{v}'), e\rangle$ and $\langle e' \cdot \hat{q}(\bar{v}'), e\rangle$ using the corresponding semantics. Consider one $\mathcal{G}$-aware *CR* execution step starting from $\langle p(\bar{v}), e\rangle$. Rule ($\mathcal{G}$−1) first chooses an equation "$p(\bar{x}) = b_1 + \cdots + b_n, \varphi$" (which has a unique id k) and an assignment $\sigma \models \bar{x} = \bar{v} \wedge \varphi$, and then non-deterministically (on the value of j) moves to one of the following states:

$$\begin{array}{l}\langle b_1\sigma, e\rangle\\ \langle \mathcal{A}(b_1, [k, 1], \mathcal{G})\sigma \cdot b_2\sigma, e\rangle\\ \qquad\qquad\vdots\\ \langle \mathcal{A}(b_1 + \cdots + b_{n-1}, [k, n-1], \mathcal{G})\sigma \cdot b_n\sigma, e\rangle.\end{array}$$

Now we consider one *CR* execution step of $\langle \hat{p}(\bar{v}), e\rangle$. First recall that, when generating the peak equations, a total allocation equation "$p(\bar{x}) = b_1 + \cdots + b_n, \varphi$" (with id k), is translated to the following set of peak equations (after unfolding the max):

$$\begin{array}{l}\hat{p}(\bar{x}) = \hat{b}_1\ , \varphi\\ \hat{p}(\bar{x}) = \mathcal{A}(b_1, [k, 1], \mathcal{G}) + \hat{b}_2\ , \varphi\\ \qquad\qquad\vdots\\ \hat{p}(\bar{x}) = \mathcal{A}(b_1 + \cdots + b_{n-1}, [k, n-1], \mathcal{G}) + \hat{b}_n\ , \varphi.\end{array}$$

In order to execute $\langle \hat{p}(\bar{v}), e\rangle$, Rule (1) can match $\hat{p}(\bar{v})$ with any of the above rules using the same assignment σ as above, because we use the same constraint $\bar{x} = \bar{v} \wedge \varphi$, and then moves to one of the following *CR* states:

$$\begin{array}{l}\langle \hat{b}_1\sigma, e\rangle\\ \langle \mathcal{A}(b_1, [k, 1], \mathcal{G})\sigma \cdot \hat{b}_2\sigma, e\rangle\\ \qquad\qquad\vdots\\ \langle \mathcal{A}(b_1 + \cdots + b_{n-1}, [k, n-1], \mathcal{G})\sigma \cdot \hat{b}_n\sigma, e\rangle\end{array}$$

which are isomorphic to those of $\langle p(\bar{v}), e\rangle$. Note that Rules ($\mathcal{G}$−2) and (2) are identical, and therefore we do not need to show anything for them. This completes the proof of the theorem for condition (i).

Now we move to the proof of the theorem for condition (ii). We start by the following observation which together with the proof for condition (i) implies the correctness of the theorem for condition (ii).

Observation 1. *The collectible property is monotonic, i.e., once an object becomes collectible it will remain collectible in any future state (until it gets collected).*

This observation is due to the fact, for example, that it is not possible to make unreachable (resp. dead) objects reachable (resp. alive). The formal definition of $\mathcal{G}$ (i.e., the collectible objects or types) should be carefully stated to satisfy this condition. Those defined in Section 4 clearly satisfy this condition since they are based on reachability and liveness. The proof of the Theorem for condition (ii) follows from its proof for condition (i) and Observation 1 as follows:

1. The proof for condition (i) guarantees that we can execute the program within the limit of $\hat{p}^{ub}(\bar{v})$ memory if we apply $\mathcal{G}$ at each state (i.e., as soon as objects become collectible).
2. Observation 1 guarantees that we can delay the application of $\mathcal{G}$ to a later point without the risk of not collecting something that we would have been collected if we had applied $\mathcal{G}$ earlier.
3. The above two points imply that delaying $\mathcal{G}$ until we are about to exceed $\hat{p}^{ub}(\bar{v})$ is safe. If not, then either the Observation 1 is not correct, or it was not possible to execute the program within the limit of $\hat{p}^{ub}(\bar{v})$ memory.

Note that we can exceed $\hat{p}^{ub}(\bar{v})$ only when allocating new memory.

Proof of Theorem 26. The correctness of this theorem stems from the fact that Lemma 28 holds also if we use the partially evaluated total allocation equations. This is because of the correctness of PE [22,24], which only unfolds calls to equations by their definitions exactly as the *CR* operational semantics does in Rule (1) when calling such equation. Then, the rest of the proof is identical to that of Theorem 20, which is based on the correctness of Lemma 28. □

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, Closed-form upper bounds in static cost analysis, Journal of Automated Reasoning 46 (2) (2011) 161–203.
[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Ramírez, From object fields to local variables: a practical approach to field-sensitive analysis, in: Static Analysis Symposium, SAS'10, in: LNCS, Springer, 2010, pp. 100–116.
[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of java bytecode, in: Rocco De Nicola (Ed.), 16th European Symposium on Programming, ESOP'07, in: Lecture Notes in Computer Science, vol. 4421, Springer, 2007, pp. 157–172.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, COSTA: design and implementation of a cost and termination analyzer for java bytecode, in: 6th International Symposium on Formal Methods for Components and Objects, FMCO'07, in: Lecture Notes in Computer Science, vol. 5382, Springer, 2008, pp. 113–133.
[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theoretical Computer Science 413 (1) (2012) 142–159.
[6] E. Albert, S. Genaim, M. Gómez-Zamalloa, Heap space analysis for java bytecode, in: Proceedings of the 6th International Symposium on Memory Management, ISMM'07, ACM Press, New York, NY, USA, 2007, pp. 105–116.
[7] E. Albert, S. Genaim, M. Gómez-Zamalloa, Live heap space analysis for languages with garbage collection, in: 8th international symposium on Memory management, ACM Press, New York, NY, USA, 2009, pp. 129–138.
[8] E. Albert, S. Genaim, M. Gómez-Zamalloa, Parametric inference of memory requirements for garbage collected languages, in: 9th International Symposium on Memory Management, ISMM'10, ACM Press, New York, NY, USA, 2010, pp. 121–130.
[9] B. Blanchet, Escape analysis for object oriented languages. application to java(tm), in: 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'99, ACM Press, 1999.
[10] V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine, Parametric prediction of heap memory requirements, in: ISMM, ACM Press, 2008.
[11] B. Cahoon, K. S. McKinley, Data flow analysis for software prefetching linked data structures in java, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, Barcelona, Spain, September 2001.
[12] S. Cherem, R. Rugina, Region analysis and transformation for java programs, in: ISMM, ACM Press, 2004, pp. 85–96.
[13] W.-N. Chin, H. H. Nguyen, S. Qin, M. C. Rinard, Memory usage verification for OO programs, in: Proc. of SAS'05, in: LNCS, vol. 3672, 2005, pp. 70–86.
[14] W-N. Chin, H.H. Nguyen, C. Popeea, S. Qin, Analysing Memory Resource Bounds for Low-Level Programs, in: ISMM, ACM Press, 2008.
[15] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: POPL, ACM Press, 1978.
[16] S. Craig, M. Leuschel, A compiler generator for constraint logic programs, in: Ershov Memorial Conference, 2003, pp. 148–161.
[17] Alain Deutsch, Interprocedural may-alias analysis for pointers: Beyond *k*-limiting, in: PLDI, 1994, pp. 230–241.
[18] S. Gulwani, K. K. Mehra, T. M. Chilimbi, SPEED: precise and efficient static estimation of program computational complexity, in: The 36th Symposium on Principles of Programming Languages, POPL'09, ACM, 2009, pp. 127–139.
[19] J. Hoffmann, M. Hofmann, Amortized Resource Analysis with Polynomial Potential, in: Lecture Notes in Computer Science, vol. 6012, Springer, 2010, pp. 287–306.
[20] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: 30th Symposium on Principles of Programming Languages, POPL'03, ACM Press, 2003.
[21] M. Hofmann, D. Rodriguez, Efficient type-checking for amortised heap-space analysis, in: Proc. of CSL'09, Springer, 2009.
[22] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, New York, 1993.
[23] H. Lehner, P. Müller, Formal translation of bytecode into BoogiePL, in: 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode'07, in: Electronic Notes in Theoretical Computer Science, Elsevier, 2007, pp. 35–50.
[24] M. Leuschel, M. Bruynooghe, Logic program specialisation through partial deduction: control issues, Theory and Practice of Logic Programming 2 (4–5) (2002) 461–515.
[25] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996.
[26] Francesco Logozzo, Cibai: an abstract interpretation-based static analyzer for modular analysis and verification of java classes, in: VMCAI, in: LNCS, vol. 4349, 2007.
[27] Ana Milanova, Atanas Rountev, Barbara G. Ryder, Parameterized object sensitivity for points-to analysis for java, ACM Transactions on Software Engineering and Methodology 14 (2005) 1–41.
[28] A. Miné, Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics, in: Proc. of LCTES'06, ACM, 2006, pp. 54–63.
[29] Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl, Automated termination analysis of java bytecode by term rewriting, in: Christopher Lynch (Ed.), RTA, in: LIPIcs, vol. 6, 2010, pp. 259–276. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[30] Y.G. Park, B. Goldberg, Escape analysis on lists, in: PLDI, 1992, pp. 116–127.
[31] R. Shaham, E. K. Kolodner, S. Sagiv, Estimating the impact of heap liveness information on space consumption in java, in: ISMM, ACM Press, 2002, pp. 171–182.
[32] F. Spoto, F. Mesnard, É. Payet, A termination analyzer for java bytecode based on path-length, ACM Transactions on Programming Languages and Systems 32 (3) (2010).
[33] Mads Tofte, Jean-Pierre Talpin, Region-based memory management, Inf. Comput. 132 (2) (1997) 109–176.
[34] L. Unnikrishnan, S. D. Stoller, Y. A. Liu, Automatic accurate live memory analysis for garbage-collected languages, in: Proc. of LCTES/OM, ACM, 2001, pp. 102–111.
[35] L. Unnikrishnan, S. D. Stoller, Y. A. Liu, Optimized live heap bound analysis, in: Verification, Model Checking, and Abstract Interpretation, VMCAI'03, in: Lecture Notes in Computer Science, vol. 2575, 2003, pp. 70–85.
[36] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, P. Co, Soot-a java optimization framework, in: Conference of the Centre for Advanced Studies on Collaborative Research, CASCON'99, IBM, 1999, pp. 125–135.
[37] B. Wegbreit, Mechanical program analysis, Communications of the ACM 18 (9) (1975).
[38] John Whaley, Monica S. Lam, Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, in: PLDI, ACM, 2004, pp. 131–144.
[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D.B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P.P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem–overview of methods and survey of tools, ACM Transactions on Embedded Computer Systems 7 (3) (2008).