Parametric Inference of Memory Requirements for Garbage Collected Languages

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

Samir Genaim
Complutense University of Madrid
samir.genaim@fdi.ucm.es

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

Abstract

The accurate prediction of program's memory requirements is a critical component in software development. 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. We present a novel approach to inferring upper bounds on memory requirements of Java-like programs which is *parametric* on the notion of *object lifetime*, i.e., on when objects become collectible. If objects lifetimes are inferred by a *reachability* analysis, then our analysis infers accurate upper bounds on the memory consumption for a reachability-based garbage collector. Interestingly, if objects lifetimes are inferred by a *heap liveness* analysis, then we approximate the program minimal memory requirement, i.e., the peak memory usage when using an optimal garbage collector which frees objects as soon as they become dead. The key idea is to integrate information on objects lifetimes into the process of generating the *recurrence equations* which capture the memory usage at the different program states. If the heap size limit is set to the memory requirement inferred by our analysis, it is ensured that execution will not exceed the memory limit with the only assumption that garbage collection works when the limit is reached. Experiments on Java bytecode programs provide evidence of the feasibility and accuracy of our analysis.

Categories and Subject Descriptors F3.2 [*Semantics of Programming Languages*]: Program Analysis; F2.0 [*Analysis of Algorithms and Problem Complexity*]: General; D3.0 [*Programming Languages*]: General

General Terms Languages, Theory, Verification, Reliability

Keywords Live Heap Space Analysis, Peak Memory Consumption, Low-level Languages, Java Bytecode, Garbage Collection

1. Introduction

Accurately estimating the *memory requirement* of programs is crucial in contexts in which memory remains a scarce commodity and also when a system failure due to insufficient memory might have severe consequences. Due in part to the difficulty of predicting the space usage of programs that use dynamic memory allocation, real-time and embedded software typically uses only statically allocated data, which is known to have disadvantages [22]. Existing *heap space analyses* aim at predicting the memory usage of programs and can be classified into the following categories:

(1) do not take garbage collection (GC) into account (e.g., [2, 7]),

(2) assume specific models of GC (e.g., scope-based GC [3, 6, 22]),

(3) are restricted to simple complexity bounds (e.g., linear [11]),

(4) are not fully automatic (e.g., [16]).

EXAMPLE 1.1. *Let us motivate our work on a contrived example which is depicted in Fig. 1 (to the left). Because it has simple (constant) memory requirements, it is useful to describe intuitively the differences among the different heap space analyses and, later, to explain the main technical parts of the paper. In Fig. 1 (to the right) we also provide several approximations of the memory requirement of executing method* m_1 *by using different approaches, where the notation* $\mathrm{s}(\mathrm{X})$ *means the memory required for an instance of class* X.

A first safe approximation is to infer the total memory allocation T *[2, 7] which accumulates the sizes of* all *objects created along the execution. By assuming a specific behavior of GC, recent approaches try to approximate the peak amount of* reachable *data on the heap, i.e., data to which some variable in the program environment points. Obviously, this approximation is tight and sound only when executing the program using the assumed GC. In* scope-based *GC, deallocation of unreachable objects takes place on method's return and only those objects created during the method's execution can be freed. By assuming this GC, [3, 6] 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* S *is obtained. The important point is that* $\mathrm{s}(\mathrm{A})$ *and* $\mathrm{s}(\mathrm{B})$ *are always accumulated, plus the largest of the consumption of* m_2 *(i.e.,* $\mathrm{s}(\mathrm{C}) + \mathrm{s}(\mathrm{E})$*) and the memory* escaped *from* m_2 *(i.e.,* $\mathrm{s}(\mathrm{E})$*) plus the continuation (i.e.,* $\mathrm{s}(\mathrm{D})$*). The scope assumption is motivated by the notion of stack reference liveness [21] in Java-like languages, according to which some objects which the local variables (and operand stack elements) point to, become unreachable upon exit from methods, i.e., when the corresponding call stack frames are removed.*

A recognized difficulty of inferring the memory requirement of a program is that the behavior of GC is unpredictable, i.e., an object which becomes eligible for GC will usually be cleaned up eventually, but there is no guarantee when (or even if) that will happen. Therefore, making any assumption during analysis (e.g., scope-based) on when objects are garbage collected might not correspond to the actual memory usage and, even worse, might not be sound if GC works less often. Different garbage collectors use different techniques for deciding on: (1) *when* GC is performed; and (2) *what* can be collected, i.e., the *lifetime* of objects. Instead of adopting a

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

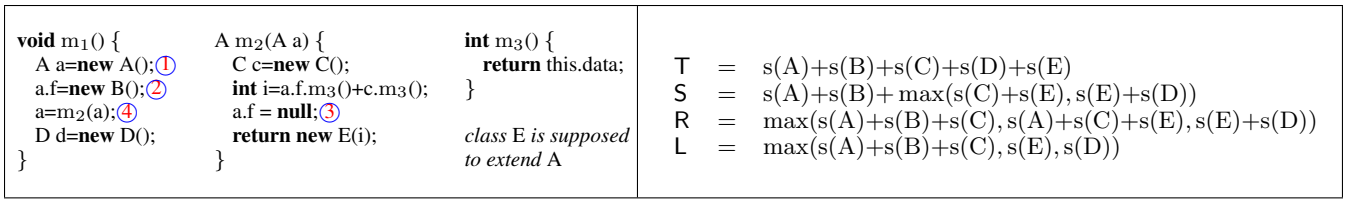

Figure 1. A Java Program and its memory requirements: T=total-allocation; S=scope-based; R=reachability-based; L=liveness-based.

particular garbage collector, we present a novel approach to accurately estimating the memory requirements of object-oriented imperative programs which is parametric w.r.t. (2), i.e., when objects are eligible for GC, and sound for the following scenarios of (1):

(i) GC collects objects as soon as they become eligible for collection or, at most, before the next heap allocation instruction.

(ii) The heap size limit has been fixed to the obtained upper bound and GC is activated when new memory is to be allocated and the limit is reached.

In any of the above scenarios, it is guaranteed that execution will not exceed the upper bounds that we infer. While scenario (i) is of theoretical interest, (ii) is practical and realistic, since the least that one can expect from GC (i.e., the less restrictive assumption) is that it frees memory when no more memory is available. Similarly to [6, 3, 22], the basic techniques we use are based on the generation of *recurrence relations* which are then solved into closed-form upper bounds (i.e., expressions without recurrences). The main challenge is to integrate into them objects lifetime information where heap data might be garbage collected at any program state. This is non-trivial since we need to generate recurrence relations which capture the memory requirements at a *program point* level, rather than at a method level as all previous approaches do. As our main contribution, we propose a novel combination of the information gathered by a previous analysis on *objects lifetime* together with the generation of recurrence relations and the use of the technique of *partial evaluation* which, as our experimental results on Java bytecode programs show, allows us to accurately infer the memory requirements at the different program states.

We develop two interesting instances of our method. In the first instance, objects lifetimes are inferred by a *reachability* analysis and without the restriction of being scope-based. Then, our method is able to obtain the upper bound 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, then 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 another instance, we consider a garbage collector which reclaims objects when they become dead (i.e, will not be used in the future). Then, we obtain the upper bound 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 the max similarly to what we have seen for R. Note that, in theory, L is indeed the minimal memory requirement for executing the method. This paper makes the following important contributions:

(1) Our work is the first one that can be used 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.

(2) Also, if we base our analysis on the same liveness information, our approach is the first one to obtain upper bounds on the memory requirement for:

(2.1) GC schemes (for Java-like languages) that take advantage of liveness information inferred at compile time [21].

(2.2) Languages with region-based memory management in which programs are instrumented with explicit region (de)allocation annotations by relying on a liveness analysis [9].

(3) Provide information for understanding/debugging the memory usage of programs, which can be a critical resource during software development.

2. Memory Requirements in Simple Imperative Bytecode

To formalize our analysis, we consider a simple *rule-based* imperative language (in the style of any of [2, 23, 18]). It has been shown that Java bytecode (and hence Java) can be compiled into this intermediate language [1]. Moreover, the translation preserves the *heap* memory requirement of the original program. 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:

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

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, field manipulation, and type comparison through the instruction $\mathsf{type}(x, \mathrm{C})$, which succeeds if the runtime class of x is exactly C. 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. The key features of this 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 and (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. The translation from (Java) bytecode to the rule-based form is per-

formed in two steps [1]. 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 2.1. *Fig. 2 shows the Java source (to the left) of another example that we will use in the paper (besides the one in Fig. 1) because it has interesting memory requirements, namely exponential and polynomial bounds. The source code is shown only for clarity as the analyzer generates the rule-based representation (to the right) 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* $\mathrm{s_i}$ *indicate that they originate from stack positions. For instance, the "*new Tree^1*" instruction creates an object of type* Tree *(the superscript* 1 *is the unique identifier for this allocation site) and assigns the corresponding reference to variable* $\mathrm{s_0}$ *(which 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* $\mathrm{s_2}$ *and a recursive call is done. A similar recursive invocation follows. In Java bytecode, constructor methods are named* init*. In both rules, the return value is* r *which in the first takes the object reference and in the second takes* null*. In the rule-based representation for* f*, the important observation is that loops are extracted in separate procedures which are treated by the analysis as methods, namely,* $\mathrm{f_1}$ *and* $\mathrm{f_2}$ *are intermediate procedures which 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.*

2.1 Semantics

The execution of bytecode in rule-based form is exactly like standard bytecode; a thorough explanation is outside the scope of this paper (see [20]). 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 and $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$. 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 which perform arithmetic and assignment operations. The evaluation $eval(exp, tv)$ returns the evaluation 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 $\mathsf{newobject}(\mathrm{C^i})$ creates a new object of class C and initializes its fields to either 0 or null, depending on their types. 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* of the form $\langle \bot, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ 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.

2.2 The Notion of Memory Requirement

We use $\mathrm{s(C)}$ to denote the amount of memory required to hold an instance object of class C, $\mathrm{s}(o)$ denotes the amount of memory occupied by an object o, and $\mathrm{s}(h)$ denotes the amount of memory occupied by all objects in the heap h, namely $\Sigma_{r\in \mathrm{dom}(h)}\mathrm{s}(h(r))$. 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 $total(t) = \mathrm{s}(h_\mathrm{n}) - \mathrm{s}(h_0)$. If the derivation is infinite, then $total(t) = \max(\{\mathrm{s}(h_i) \mid S_i = A_i; h_i \in t\}) - \mathrm{s}(h_0)$.

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 memory requirement 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 *memory requirement* (or peak memory usage) of t in the presence of GC is defined as $peak(t) = \max(\{\mathrm{s}(h_i) \mid S_i = A_i; h_i \in t\}) - \mathrm{s}(h_0)$. This is the notion that our analysis aims at approximating.

3. Parametric Inference of Memory Requirements

In practice, when executing a program, there is a maximum heap size limit, and when it is exceeded the program terminates in an *out of heap memory* error state. Building over previous work on heap space analysis [2, 3], we present a novel approach to inferring accurate memory requirements which can be used in combination with reachability-based or liveness-based memory managers. From a practical point of view, our analysis results are sound under only the assumption that GC will reclaim memory if we are about to exceed the heap size limit and new memory has to be allocated. This is to say that, when executing the program by fixing the inferred memory requirement as heap size limit, it is ensured that the program will not run into an *out of heap memory* state.[1] The analysis consists of four steps which will be described in the following four sections:

1. Generation of upper bounds on the total memory consumption ([2, 4]).
2. Inference of information on lifetime of objects.
3. Generation of memory requirement recurrence relations.
4. A partial-evaluation based transformation of the recurrence equations.

3.1 Total Memory Allocation Upper Bounds

Any heap space analysis aims at approximating the memory usage of the program as a function of the input *data size*s. As customary, the *size* of data is determined by its variable type [1]: 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., if variable l in $\mathrm{f_1}$ is a reference to a

[1] In Java, the option "`-Xmxn`" can be used to specify the memory limit.

```
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.
```

Figure 2. Java code of running example and rule-based representation of m and f

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

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

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

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

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

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

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

list, then l represents its length. Note that the size measure of data structures is unrelated to function $\mathrm{s(C)}$ as defined in the previous section to measure the actual space occupancy. The size measure is mainly used for estimating the number of iterations of recursive procedures. When we need to compute the sizes $\bar{v}$ of a tuple of variables $\bar{\mathrm{x}}$, we use the notation $\bar{v} = \alpha(\bar{\mathrm{x}}, tv, h)$, which means that the integer value v_i is the size of the variable $\mathrm{x_i}$ in the context of the variables table tv and the heap h. For instance, 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 [12]. For instance, associated with the recursive rule $\mathrm{f_1}$, we infer the size relation $l > l'$ which indicates that the length of l decreases when calling $\mathrm{f_1}$ recursively where l' refers to the size of l at the program point where $\mathrm{f_1}$ 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 and refer to [12, 1] 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 [14]. 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 3.1 (total memory allocation equations [2]). *Let* r *be a rule of the form* $\mathrm{p}(\langle \bar{\mathrm{x}}\rangle, \langle \bar{\mathrm{y}}\rangle) \leftarrow \mathrm{g}, \mathrm{b_1}, \ldots, \mathrm{b_n}$ *and* φ_r *its corresponding size relations. Then, its* total memory allocation equation *is defined as* $p(\bar{x}) = \Sigma_{i=1}^{n} \mathcal{M}(\mathrm{b_i}), \varphi_r$ *where* $\mathcal{M}(\mathrm{x := new\ C^i}) = \mathrm{s(C^i)}$, $\mathcal{M}(\mathrm{q_i}(\langle \bar{\mathrm{x}}_\mathrm{i}\rangle, \langle \bar{\mathrm{y}}_\mathrm{i}\rangle)) = q_i(\bar{x}_i)$*; otherwise* $\mathcal{M}(\mathrm{b_i}) = 0$.

Note that each call in the rule $\mathrm{q_i}(\langle \bar{\mathrm{x}}_\mathrm{i}\rangle, \langle \bar{\mathrm{y}}_\mathrm{i}\rangle)$ has a corresponding abstract version $q_i(\bar{x}_i)$ where $\bar{x}_i$ are the size abstractions of $\bar{\mathrm{x}}_\mathrm{i}$ 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 was not required by previous memory analyses [3, 2],

but it will be crucial when developing our analysis in order to make RR capture the heap space usage at a program point level.

EXAMPLE 3.2. *The total allocation equations for the rules* m *and* f *in Fig. 2 are:*

$$\begin{array}{ll} m(n) = \mathrm{s}(\mathrm{Tree}^1)+\mathrm{g}(\mathrm{n})+\mathrm{f}(\mathrm{n},\mathrm{s}_1)+\mathrm{m}(\mathrm{s}_2)+\mathrm{m}(\mathrm{s}_3) & \\ \qquad\qquad \{n{>}0, s_1{=}n, s_2{=}n{-}1, s_3{=}n{-}1\} & \\ m(n) = 0 \qquad\qquad \{n{=}0\} & \\ f(n,l){=}f_1(l,r) + f_2(i,r') \qquad \{r{=}0, i{=}n\} & \end{array}$$

The first equation corresponds to the first rule of procedure m. *It states that the total memory consumption when executing* m *with an input value* n *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. *Note that the attached constraints describe the size relations between the local variables and the input variable* n. *The total allocation for* f *is the sum of total allocations of the loops* f_1 *and* f_2. □

Once the RR are generated, a worst-case cost analyzer uses a solver in order to obtain closed-form upper bounds, i.e., cost expressions without recurrences. Given a RR $p(\bar{x})$, we denote by $p^{ub}(\bar{x})$ its upper bound. The upper bounds that [4] can infer from the above RR are *cost expressions* of the following form:

$$e \equiv n|\mathrm{s}(\mathrm{C})|\mathsf{nat}(\mathrm{l})|\log(\mathsf{nat}(\mathrm{l})+1)|\mathrm{e}*\mathrm{e}|\mathrm{e}_1+\mathrm{e}_1|2^{\mathsf{nat}(\mathrm{l})}| \max(\{e_1,\ldots,e_k\})$$

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. 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 [4] to obtain upper bounds for our examples, sometimes simplified by removing constants to facilitate understanding.

EXAMPLE 3.3. *The total memory upper bounds obtained from the RR of Ex. 3.2 are:*

$$\begin{array}{l} f^{ub}(n,l){=}\mathsf{nat}(l)*\mathrm{s}(\mathrm{Long}^3)+\mathsf{nat}(\mathrm{n})*\mathrm{s}(\mathrm{Integer}^4) \\ m^{ub}(n) = (2^{\mathsf{nat}(n)}{-}1){*}(\mathrm{s}(\mathrm{Tree}^1)+\mathsf{nat}(\mathrm{n})*(\mathrm{s}(\mathrm{List}^2)+ \\ \qquad\qquad \mathrm{s}(\mathrm{Long}^3)+\mathrm{s}(\mathrm{Integer}^4))) \end{array}$$

Intuitively, for method f, *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 upper bounds capture exactly this intuition.*

3.2 Inference of Objects Lifetime

A *GC strategy* classifies objects in the heap into two categories: those which are collectible and those which are not. We mean by *applying a GC strategy on a heap* the process of removing all collectible objects from such heap. 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. The static inference of both such collectible information is undecidable and therefore it is usually approximated. A common approach to approximate it is to provide information for each program point on the *types* (i.e., the class name with allocation site) of collectible objects instead of on the actual objects. This approximation is typically done w.r.t. an *entry* procedure (such as `main` in Java). In order to define these notions, we first make all program points unique. The k-th program rule $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle) \leftarrow g, b_1^k,\ldots,b_n^k$ has $n{+}1$ program points. The first one, $[k,0]$, after the execution of the guard g and before the execution of b_1 until $[k,n]$ after the execution of b_n.

DEFINITION 3.4 (collectible classes). *Let* P *be a program with an entry procedure* $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$, C^{i} *a type,* $[k,j]$ *a program point, and* $\mathcal{G}$ *a GC strategy. We say that* 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* $S_n{=}\langle p^k, b_j^k \cdot bc, tv_n\rangle{\cdot}A; h_n$ *in* t*, applying* $\mathcal{G}$ *on* h_n *results in a heap which does not have any object of type* C^{i} *(i.e. any object created at statement "*new C^{i}*"). The set of all collectible classes at* $[k,j]$ *w.r.t.* $\mathcal{G}$ *is denoted* $\mathcal{C}(\mathcal{G},[k,j])$.

The set of all collectible classes at a program point $[k,j]$ w.r.t. a reachability-based GC strategy $\mathcal{G}_r$, called *unreachable classes*, is denoted $\mathcal{C}(\mathcal{G}_r,[k,j]))$. This set can be approximated using points-to analysis [24]. For simplicity, we assume that *all* unreachable objects are collected whenever $\mathcal{G}_r$ is activated. In practice, this can be adjusted to the actual underlying GC strategy.

EXAMPLE 3.5. *The following reachability information is obtained for the program points in Fig. 2. At* ⓐ, ⓑ ,ⓓ *and* ⓔ*, the set of reachable classes is* $\{\mathrm{Tree}^1\}$*; and at* ⓒ, ⓕ, ⓖ, ⓗ, ⓘ, ⓙ, *and* ⓚ *the set of reachable classes is* $\{\mathrm{Tree}^1,\mathrm{List}^2\}$. *Likewise, the reachability information for the program points in Fig. 1 is that at* ① *the set of reachable classes is* $\{\mathrm{A}\}$*, at* ② *is* $\{\mathrm{A},\mathrm{B}\}$*, at* ③ *is* $\{\mathrm{A},\mathrm{C}\}$*, and at* ④ *is* $\{\mathrm{E}\}$. *The complementary sets are the unreachable classes.*

As another instance, we consider a liveness-based GC strategy. This is interesting because it allows us to approximate the memory requirement for region-based memory managers and garbage collectors that make use of compile-time liveness information. Besides, it allows obtaining upper-bounds on the (theoretical) minimal memory requirement for executing a program, which provides useful information for understanding/debugging the memory usage of programs. The set of all collectible classes at a program point $[k,j]$ w.r.t. a liveness-based GC strategy $\mathcal{G}_l$, called *dead classes*, is denoted by $\mathcal{C}(\mathcal{G}_l,[k,j]))$. This set can be approximated by using points-to analysis and backwards inference similar to [9].

EXAMPLE 3.6. *The following liveness information is obtained for the program points in Fig. 2: at* ⓐ, ⓑ ,ⓓ ,ⓔ, ⓖ, ⓙ, *and* ⓚ *the set of live classes is* $\{\mathrm{Tree}^1\}$*; and at* ⓒ, ⓕ, ⓗ *and* ⓘ *it is* $\{\mathrm{Tree}^1,\mathrm{List}^2\}$. *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 classes is* $\{\mathrm{A}\}$ *and at* ② *is* $\{\mathrm{A},\mathrm{B}\}$*, and at* ③ *and* ④ *is* $\emptyset$. *The complementary sets form the dead classes.*

Note that usually context-insensitive points-to analysis is enough in order to precisely approximate unreachable and dead classes sets. However, when advanced object oriented patterns are used, such as factory, context-sensitive points-to analysis might be required.

3.3 Parametric Recurrence Relations

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 upper bound for $n_1 - n_2$, one can obtain an upper bound for n_1 and a lower bound for n_2. In our context, we have a symbolic

upper bound e_1 for n_1, but the information about collectible classes does not provide any lower bound on n_2, since it does not include information on how many objects of each type we have at that program point. But, since the collectible classes sets in Sec. 3.2, provide the information that *all* instances of specific classes at that program point can be freed, and since e_1 is a symbolic upper bound, we can obtain a sound upper bound for $n_1 - n_2$ by replacing in e_1 all occurrences of $\mathrm{s(C)}$ by 0, for each C that is in the corresponding set of collectible classes. By relying on this basic idea, we will estimate the memory requirement at each program point and then choose the maximum among them.

In what follows, given a cost expression $e{=}e_1 + \ldots + e_n$, we denote by $e[e_i \mapsto f]$ the replacement of the sub-expression e_i by f.

DEFINITION 3.7 (total memory). *Given an expression $e = e_1 + \ldots + e_n$, where each e_i is either of the form $\mathrm{s(C^i)}$ or a call $q(\bar{x})$, we define its total memory, denoted $total(e)$, to the cost expression obtained as $e[e_i \mapsto q^{ub}(\bar{x})]$, for each e_i which is a call to a procedure of the form $q(\bar{x})$.*

EXAMPLE 3.8. *Assume that $m_2^{ub} = \mathrm{s(C)} + \mathrm{s(E)}$. Given the expression $e = \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{m_2} + \mathrm{s(D)}$, we have $total(e) = \mathrm{s(A)} + \mathrm{s(B)} + \mathrm{s(C)} + \mathrm{s(E)} + \mathrm{s(D)}$.*

The interest of having the total memory is that we can approximate the *active memory* (i.e., the memory that cannot be collected by the GC) at a given program point from it as follows.

DEFINITION 3.9 (active memory). *Given an expression e, we define the* active memory *at a program point $[k, j]$ w.r.t. a GC strategy $\mathcal{G}$, denoted $\mathcal{A}(e, [k, j], \mathcal{G})$, as the cost expression obtained as $total(e)[\mathrm{s(C^i)} \mapsto 0]$ for all $\mathrm{C^i} \in \mathcal{C}(\mathcal{G}, [k, j])$.*

EXAMPLE 3.10. *From the expression e in Ex. 3.8, the reachability information of Ex. 3.5, and the liveness information of Ex. 3.6, for the program point ④ in Fig. 1, we obtain $\mathcal{A}(e, ④, \mathcal{G}_r) = \mathrm{s(E)}$ and $\mathcal{A}(e, ④, \mathcal{G}_l) = 0$.*

The main idea behind our memory requirements analysis is to produce disjunctive equations which capture the active memory at the program points where the memory usage can increase, i.e., at the memory allocation instructions. Since the RR generated in Def. 3.1 allow us to identify exactly these points by means of their associated *size* constants, we generate the parametric memory requirements equations from them.[2]

DEFINITION 3.11 (memory requirements equations). *Given a total memory allocation equation "$p(\bar{x}) = e_1 + \cdots + e_n,\ \varphi$" and a GC strategy $\mathcal{G}$, the corresponding* memory requirement equation *is defined as: "$\hat{p}(\bar{x}) = max(f_1, \ldots, f_n),\ \varphi$", such that $f_i = \mathcal{A}(e_1 + \cdots + e_{i-1}, [k, j], \mathcal{G}) + \hat{e}_i$ where $[k, j]$ is the program point that corresponds to e_i and $\hat{\mathrm{s}}(\mathrm{C^i}) = \mathrm{s(C^i)}$.*

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 total memory upper bounds 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 memory requirements equations when applying Def. 3.11 to the equations defining e_i. When e_i is a constant, function $\hat{\mathrm{s}}$ is just s. As mentioned in Sec. 3.1, 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.

[2] In previous work [3, 2] the equations are generated from the program rules instead.

EXAMPLE 3.12. *The total memory equations for the methods in Fig. 1 are (for simplicity we ignore $\mathrm{m_3}$ since it does not affect the memory consumption):*

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

According to Def. 3.11, the following memory requirements equations are obtained for a generic GC strategy $\mathcal{G}$:

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

The disjunctive information is handled in the solver by replacing the max operator by non-deterministic equations and finding an upper bound using [4]. Below, we show to the left (resp. right) the non-deterministic equations for a reachability-based strategy before (resp. after) the elimination of the unreachable classes computed in Ex. 3.5. The notation ~~s(C)~~ indicates setting $\mathrm{s(C)}$ to zero once collectible classes are eliminated:

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

The upper bounds obtained from these equations are

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

This upper bound improves over the scope-based upper bound S in Fig. 1 but we still have not succeeded in inferring R. This is because we need the partial evaluation transformation explained in the next section.

As another instance, the equations for a liveness-based strategy $\mathcal{G}_l$ (i.e., w.r.t. the dead classes of Ex. 3.6) differ from the above ones only in the fourth and sixth equations which are as follows:

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

The upper bounds obtained from these equations are

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

Again, the partial evaluation transformation is needed to obtain L of Fig. 1. As mentioned in Sec. 1, liveness-based upper bounds safely and accurately describe the peak memory usage for a memory manager based on liveness. For example, this is the case of region-based manager, for the program in Fig. 2 if objects of the same type are allocated in the same region, and for the program in Fig. 1 if A, B and C are allocated in one region which is deallocated at program point ③.

THEOREM 3.13 (soundness). *Let P be a program, $\mathcal{G}$ be the GC strategy, $\mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle)$ an entry procedure, and $\hat{p}^{ub}(\bar{x})$ an upper bound for the corresponding memory requirement equations generated in Def. 3.11. Assuming that we start the execution from an initial state $S_0 = \langle \bot, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle), tv_0 \rangle; h_0$ 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;

(ii) the heap size is fixed to $\hat{p}^{ub}(\bar{v})$ and $\mathcal{G}$ is applied when we reach this limit.

Informally, the 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 Sec. 1.

3.4 A Partial Evaluation Transformation of Recurrence Relations

The technique in Sec. 3.3 obtains precise upper bounds 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 become visible in the same rule, which might result in a loss of precision. This happens in the program of Fig. 1, the object to which the variable "a" refers becomes dead in m_2 at program point ③, and the object to which "a.f" refers becomes both dead and unreachable at program point ③. However, the equations that we generate for m_1 in Ex. 3.12 (both for $\mathcal{G}_l$ and $\mathcal{G}_r$) do not take advantage of this information, but rather from 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 upper bounds R and L in Fig. 1. The well-known technique of partial evaluation [17] (PE for short) gives us a leeway. 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.

EXAMPLE 3.14. *Consider the total memory equations of Ex. 3.12. Unfolding the call to m_2 into its calling context results in the following equation:*

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

From it, by applying Def. 3.11, we obtain the following equation for $\mathcal{G}_r$ which is clearly more accurate than the one we have obtained in Ex. 3.12:

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

Solving this equation results in the optimal upper bound R. *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 Ex. 3.12 since it was in a different rule. Similarly, for $\mathcal{G}_l$ we get an equation which has an upper bound that coincides with the upper bound* L *of Fig. 1.*

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 $\mathrm{s(C)}$, calls to other equations and linear constraints. This kind of unfolding is basically the same as that of clauses in constraint logic programming [13] and that of the upper bound solver of [4].

DEFINITION 3.15 (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.

Performing an unfolding step solves the 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 [19]. For instance, in [4], the strategy is to leave one relation per *recursive* strongly connected component (SCC) and unfold the remaining ones. 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 annotation* (BTA) is 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). Our definition of partially evaluated equations can be used with any BTA which ensures termination.

DEFINITION 3.16 (partially evaluated RR). *Given a set of RRs and a BTA, the* partially evaluated RRs *are obtained by iteratively unfolding (Def. 3.15) all calls in the rhs of the equations which do not belong to BTA w.r.t. its defining equations.*

Once the RR have been partially evaluated, we apply Def. 3.11 to them in order to generate the corresponding memory requirement equations.

EXAMPLE 3.17. *For the simple example of Fig. 1, if the BTA includes only m_1, we obtain the RR in Ex. 3.14 and the optimal* R *and* L *upper bounds of Fig. 1.*

THEOREM 3.18 (soundness). *Let P be a program, $\mathcal{G}$ the used GC technique, $\mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle)$ an entry procedure, and $\hat{p}^{ub}(\bar{x})$ an upper bound for the corresponding memory requirement equations generated in Def. 3.11 after PE. Assuming that we start the execution from an initial state $S_0 = \langle \bot, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle), tv_0 \rangle; h_0$ then, for any complete trace t, it holds $\hat{p}^{ub}(\bar{v}) \geq peak(t)$ under the same conditions as in Theorem 3.13 where $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_0, h_0)$.*

The key difference with the PE of [4] is that we apply PE *previously* to the generation of the memory requirement RR, while [4] 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 memory requirements bounds by using PE like [4]. As other differences, we can apply PE with any terminating BTA while [4] requires checking further conditions on the associated graph.

3.5 Comparison to Previous Work

Our work improves over scope-based heap space analyses [3, 6] in both its accuracy and its applicability. Essentially, since we overcome the scope-based restriction, we are able to infer strictly more

precise upper bounds for reachability-based memory managers, as the example below will show. Besides, since our equations capture the memory requirement at a program point level (rather than as a method level as previous approaches do), we can apply it in combination with a liveness-based memory manager which can deallocate objects at any program point. Altogether, our analysis introduces the novel applications pointed out in Sec. 1, which scope-based analyses do not have. In order to clarify the technical differences with the analysis in [3], we apply the scope-based method to our running examples. We obtain the following equation for the program in Fig. 1:

$$\hat{m}_1 = \mathrm{s(A)} + \mathrm{s(B)} + \max(\hat{\mathrm{m}}_2, \check{\mathrm{m}}_2 + \mathrm{s(D)})$$

where $\hat{m}_2$ denotes the peak consumption of m_2 and $\check{m}_2$ its *escaped memory*, i.e., the memory created during the execution of m_2 and still reachable upon return from it. The idea is hence that when there is a method call, the equation contains a disjunctive max between the peak consumption of such call (i.e., $\hat{m}_2$) and the memory that escapes from it plus the peak of the continuation (i.e., $\check{m}_2 + \mathrm{s(D)}$). Apart from the different way in generating the equations, a fundamental difference with our analysis is that, by relying on a escape analysis, only the objects created in the current scope and in scopes reached transitively from it can be deallocated. This is clearly a subset of program point reachability or liveness as we do. As a consequence, the sizes of A and B are always accumulated and, as they become dead (resp. unreachable) in m_2 (resp. upon exit from m_2), this memory state is missed in the RR. This leads to imprecision when inferring memory requirements and illustrates why we need RR which capture the memory usage at a program point level, as discussed in Sec. 1.

The optimization sketched in [3] to approximate the *ideal* GC consists in splitting the rules into smaller scopes in order to apply GC more often. Unfortunately, it suffers from the same scope limitation and hence it does not improve the upper bound neither. Moreover, it is not straightforward, and in many cases it is even impossible, to generate these smaller scopes, since the scope might begin in the middle of one rule and end in the middle of another one. For example, in the program of Fig. 1, a new scope should be created for the first two instructions of m_1 and the part of m_2 up to program point ③. Moreover, the partition which is convenient for one object might not be good for another one. In this example, the accuracy gain is a constant factor. In more complex programs, the gain can be much larger as the next example shows.

EXAMPLE 3.19. *By applying Def. 3.11, the memory requirement equation generated for* m *from the equation in Ex. 3.2, the total upper bounds of Ex. 3.3 and the reachable classes of Ex. 3.5 is:*

$$\begin{aligned}\hat{m}(n)=\\ \max(&\mathrm{s(Tree^1)}, \mathrm{s(Tree^1)}+\hat{\mathrm{g}}(\mathrm{n}), \mathrm{s(Tree^1)}+\mathsf{nat}(\mathrm{n})*\mathrm{s(List^2)}+\\ &\hat{f}(n,s_1), \mathrm{s(Tree^1)}+\cancel{\mathsf{nat}(n)*\mathrm{s(List^2)}}+\mathsf{nat}(s_1)*\cancel{\mathrm{s(Long^3)}}+\\ &\cancel{\mathsf{nat}(n)*\mathrm{s(Integer^4)}}+\hat{m}(s_2), \mathrm{s(Tree^1)}+\cancel{\mathsf{nat}(n)*\mathrm{s(List^2)}}\\ &\cancel{+\mathsf{nat}(s_1)*\mathrm{s(Long^3)}}+\mathsf{nat}(n)*\cancel{\mathrm{s(Integer^4)}}+\\ &(2^{\mathsf{nat}(s_2)}-1)*\mathrm{s(Tree^1)}+\hat{\mathrm{m}}(\mathrm{s_3})\ )\end{aligned}$$

which can be solved (after solving $\hat{f}$) into the following closed form for the case $n>0$:

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

The upper bound obtained by [3] (applying the optimization of ideal GC) is:

$$\begin{aligned}&(2^{\mathsf{nat}(n)}-1)*(\mathrm{s(Tree^1)}+\mathsf{nat}(\mathrm{n})*\mathrm{s(List^2)})+\\ &\mathsf{nat}(n)*\max(\mathrm{s(Long^3)},\mathrm{s(Integer^4)})\end{aligned}$$

There are two fundamental differences:

(1) For executing f*, the approach described in this paper require only* $\max(\mathrm{s(Long^3)},\mathrm{s(Integer^4)})$ *of memory, while [3] requires* $\mathsf{nat}(n)$ *times of* $\max(\mathrm{s(Long^3)},\mathrm{s(Integer^4)})$*. This is because in a scope-based approach the objects created in* $\mathrm{f_1}$ *(resp.* $\mathrm{f_2}$*) are not garbage collected until we exit from* $\mathrm{f_1}$ *(resp.* $\mathrm{f_2}$*), while they become dead/unreachable right after calling* intValue*;*

(2) The memory required by g *is accumulated only once to the memory requirement of* m*, while [3] requires space for allocating* g *an exponential number of times. This is because the* $\mathrm{List^2}$ *objects are created in* g *and become unreachable (resp. dead) in a different scope* m *(resp.* f*).*

The above example not only shows the further accuracy of our approach w.r.t. scope-based ones, but it also 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 [16] which are often restricted to linear upper bounds.

4. Experimental Evaluation

We have implemented our technique within XYZ[3] which can be tried out through its web interface at: XYZ[4] by selecting the cost models for *memory requirement*. Our reachability analysis is based on the points-to analysis of [24], and the heap liveness analysis is similar to the region-based liveness of [9]. The PE transformation leaves one relation per SCC as explained in Sec. 3.4. The experimental evaluation has been performed on the JOlden benchmark suite [8] and on `GCBench`, a typical example from the GC community [5]. For each benchmark, we infer total allocation upper bounds $\mathbf{U}_\mathsf{T}$ and the peak heap usage using the cost models: $\mathbf{U}_\mathsf{S}$ for scope-based GC, $\mathbf{U}_\mathsf{R}$ for reachability-based GC, and $\mathbf{U}_\mathsf{L}$ for liveness-based GC. As regards the `GCBench` benchmark, its `main` method consists of 4 parts that create different data structures. In this case, $\mathbf{U}_\mathsf{T}$ and $\mathbf{U}_\mathsf{S}$ are the sum of the sizes of these data structures since they escape to the scope of the `main` and hence cannot be collected until execution finishes, while $\mathbf{U}_\mathsf{R}$ and $\mathbf{U}_\mathsf{L}$ are the maximum among their sizes. Besides, in the last part of the `main`, there are 3 nested loops and, at each iteration of the inner one, a tree is created and becomes dead and unreachable at the end of it. For this part, $\mathbf{U}_\mathsf{T}$ and $\mathbf{U}_\mathsf{S}$ require a quadratic number of trees while $\mathbf{U}_\mathsf{R}$ and $\mathbf{U}_\mathsf{L}$ require a single tree (the upper bounds are too large to be shown).

As regards the JOlden benchmarks, they all create an initial data structure and then do some post-processing on it. Therefore, the overall memory requirement is dominated by the size of the initial data structure and hence $\mathbf{U}_\mathsf{S}$, $\mathbf{U}_\mathsf{R}$ and $\mathbf{U}_\mathsf{L}$ are almost the same if we start the analysis from `main`. However, during the post-processing, there are some methods in which temporary objects are created and, for them, $\mathbf{U}_\mathsf{S}$ is significantly larger than $\mathbf{U}_\mathsf{R}$ and $\mathbf{U}_\mathsf{L}$. Table 1 summarizes our experiments by focusing on these methods. We substitute the symbolic expressions $\mathrm{s(C)}$ by the number of fields C has, and for arrays we consider their sizes, so that the system can perform mathematical simplifications. As theoretically expected, in all examples $\mathbf{U}_\mathsf{T} \geq \mathbf{U}_\mathsf{S} \geq \mathbf{U}_\mathsf{R} \geq \mathbf{U}_\mathsf{L}$. Let us explain intuitively where the gain in accuracy comes from. In the first two methods, we found loops that create objects which become dead at the end of each iteration, but we could not infer that all of them become unreachable at that point, therefore $\mathbf{U}_\mathsf{L}$ is more precise than $\mathbf{U}_\mathsf{R}$. The `hackGravity` method clones an object of type `MathVector` and immediately in the next instruction it creates an object of type `HG`. Our analysis accurately infers that

[3] the system name is withheld

[4] the actual link is withheld

Bench	$\mathbf{U}_\mathsf{T}$	$\mathbf{U}_\mathsf{S}$	$\mathbf{U}_\mathsf{R}$	$\mathbf{U}_\mathsf{L}$	$\mathbf{R}_\mathsf{R}$
mst.MST.computeMST	4*nat(A-1)	4*nat(A-1)	4+2*nat(A-1)	2	2
bh.Tree.vp	28*nat(A)+8	28*nat(A)+8	28*nat(A)+8	30	12*A+24
bh.Body.hackGravity	21	21	21	15	12
bh.MathVector.toString	10	10	10	4	2
em3d.BiGraph.toString	4*nat(A-1)+2	4*nat(A-1)+2	4	4	4
voronoi.Vertex.print	$4*2^{\mathtt{nat(A-1)}}$ - 2	2*nat(A-1)+2	2	2	2
bisort.Value.inOrder	$4*2^{\mathtt{nat(A-1)}}$ - 2	2*nat(A-1)+2	2	2	2

Table 1. Total, Scope, Reachability, Liveness Upper Bounds by COSTA on the JOlden

MathVector becomes dead before creating HG. Indeed, this has spotted a possible bug: the cloned MathVector object is not used but rather the original one. Again, the objects are still reachable and hence $\mathbf{U}_\mathsf{L} > \mathbf{U}_\mathsf{R}$. In the last three methods the gain originates from the creation and manipulation of Strings. Usually, several StringBuffer objects are created to hold intermediate strings and become dead and unreachable as soon as data is printed. When such a statement occurs inside a linear loop like in em3d.toString, we are able to infer a constant number of StringBuffer objects, while the scope-based peak analysis obtains one proportional to the number of iterations. Even more, in the last two methods, the total consumption is exponential due to two recursive calls. The scope-based peak is linear since the objects created during the execution do not escape from the method and hence can be garbage collected before the second call. Importantly, we improve this upper bound to a constant in the case of $\mathbf{U}_\mathsf{R}$ and $\mathbf{U}_\mathsf{L}$.

Finally, in the last column, we aim at evaluating the precision of our analysis for Java programs by comparing our upper bounds with the memory consumption of real executions. For this aim, we have implemented a JVMTI agent[5] which, besides tracking object allocations and deallocations, explicitly invokes the (reachability-based) JVM garbage collector before any object or array creation. Column $\mathbf{R}_\mathsf{R}$ shows the obtained consumptions, using the same measure as before (i.e. number of fields and size of arrays) which correspond to the *real* minimal memory requirements. The sizes of the corresponding input are not relevant when the actual memory consumption is constant, which is the case of all benchmarks except bh.Tree.vp. For this one, we have run the program with a set of different inputs, and from the obtained results we have derived the cost expression 12*A+24, which characterizes the actual consumption for all values of A greater than 1. It can be observed that $\mathbf{U}_\mathsf{R}$ is safe in all cases and very close to $\mathbf{R}_\mathsf{R}$ in most of them, except for the first two benchmarks in which the reachability analysis loses precision, mainly due to the context-insensitive nature of the underlying points-to analysis. Note that, in spite of the fact that $\mathbf{U}_\mathsf{L}$ is based on liveness and hence more precise, it is larger than $\mathbf{R}_\mathsf{R}$ in some cases. This is because of the loss of precision in the underlying (static) liveness analysis. All in all, our experiments reveal that the memory requirements inferred by our analysis are accurate and close to the actual consumption.

5. Conclusions

Predicting the memory requirements of a program's execution is a critical component in software development. The memory requirements typically include both the heap and the frames stack usage. This paper focuses on the heap space because estimating the maximal height of the frames stack from our heap analysis is straightforward, as it is done in [3]. Memory usage has been traditionally measured using profiling which is often insufficient since only certain inputs are profiled. Building over recent progress in heap space analysis, our work presents a novel approach to inferring symbolic bounds of the memory requirements for imperative object-oriented languages. It improves over recent work in that it is not tied to a particular GC model (unlike [3, 6]), it computes accurate bounds for exponential, logarithmic, etc. complexities (unlike [6, 11]) and it is fully automatic (unlike [16]). Other approaches to resource usage analysis are developed to measure other types of resources, namely [14] predicts number of instructions, [1] is generic in the definition of cost model but neither of them supports memory requirements, since the underlying techniques are developed to measure accumulative resources, while memory usage is a resource that increases and decreases along the execution. The problem is different from memory prediction in functional languages [22, 15] since, due to the absence of mutable data structures, GC can be modeled in a scoped-based fashion where the scopes are determined by the corresponding function definitions. While in imperative languages, objects can be collected outside the scope in which they have been created, which makes the static prediction more difficult. Other work, such as [10], 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.

[5] See http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/

Acknowledgments

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 *DOVES* project, the TIN2008-05223-E and TIN2008-04473-E (Accion Especial) projects, and HI2008-0153 (Acción Integrada) project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *Proceedings of the 6th International Symposium on Memory Management (ISMM'07)*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *8th international symposium on Memory management*, pages 129–138, New York, NY, USA, June 2009. ACM Press.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain,*

July 15-17, 2008, Proceedings, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.

[5] H. Boehm. An artificial garbage collection benchmark. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html`.

[6] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.

[7] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[8] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *IEEE PACT*, pages 280–291, 2001.

[9] S. Cherem and R. Rugina. Region analysis and transformation for java programs. In *ISMM*, pages 85–96. ACM Press, 2004.

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

[11] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[13] S. Craig and M. Leuschel. A compiler generator for constraint logic programs. In *Ershov Memorial Conference*, pages 148–161, 2003.

[14] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.

[15] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.

[16] M. Hofmann and D. Rodriguez. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Proc. of CSL'09*, LNCS. Springer, 2009.

[17] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[18] H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.

[19] M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

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

[21] R. Shaham, E. K. Kolodner, and S. Sagiv. Estimating the impact of heap liveness information on space consumption in java. In *ISMM*, pages 171–182. ACM Press, 2002.

[22] L. Unnikrishnan and S. Stoller. Parametric heap usage analysis for functional programs. In *Proc. of ISMM'09*. ACM Press, 2009.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[24] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.