Parametric Prediction of Heap Memory Requirements

Víctor Braberman Federico Fernández
Diego Garbervetsky
Departamento de Computación, FCEyN, UBA, Argentina
{vbraber, ffernandez, diegog}@dc.uba.ar

Sergio Yovine
CNRS-VERIMAG, France
Sergio.Yovine@imag.fr

Abstract

This work presents a technique to compute symbolic polynomial approximations of the amount of dynamic memory *required* to safely execute a method without running out of memory, for Java-like imperative programs. We consider object allocations and deallocations made by the method and the methods it transitively calls. More precisely, given an initial configuration of the stack and the heap, the *peak* memory consumption is the maximum space occupied by newly created objects in all states along a run from it. We over-approximate the peak memory consumption using a scoped-memory management where objects are organized in regions associated with the lifetime of methods. We model the problem of computing the maximum memory occupied by any region configuration as a parametric polynomial optimization problem over a polyhedral domain and resort to Bernstein basis to solve it. We apply the developed tool to several benchmarks.

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

General Terms Languages, Theory, Verification, Reliability

Keywords Heap Space Analysis, Heap Consumption, Memory Regions, Java

1. Introduction

Automatic dynamic memory management is a very powerful and useful mechanism which does not come for free. Indeed, it is well known that garbage collection makes execution and response times extremely difficult to predict, mainly because of unbounded pause times. Several solutions have been proposed, either by building GCs with real-time performance, e.g. [4] (see [15] for a survey), or by using a scope-based programming paradigm, e.g. [18, 6, 9]. However, there is still the problem of predicting *how much memory* a program will need to run without crashing with an out-of-memory exception. This question is inherently hard [19].

In [7] we presented a technique for computing a parametric upper-bound of the amount of memory dynamically *requested* by Java-like imperative programs. The idea consists in quantifying dynamic allocations done by a method. Given a method m with parameters P_m it shows an algorithm that computes a polynomial over P_m which over-approximates the amount of memory allocated during the execution of m. This bound is a symbolic over-approximation of the total amount of memory the application *requests* to a virtual machine via `new` statements, but not the *actual* amount of memory really consumed by the application. This is because memory freed by the GC is *not* taken into account. Roughly speaking, the technique identifies allocation sites (`new` statements) reachable from the method under analysis and counts the number of times those statements are visited. To do that, linear invariants describing possible valuations of variables at each allocation site are generated and the number of integer solutions of those invariants are counted. Finally the result is adapted to consider the type of each allocated object. For region-based memory management [18, 6, 9] where objects are placed in regions associated with computation units, the same technique allows obtaining polynomial bounds of the size of every memory region, assuming regions are synthesized at compile-time (e.g.,[27, 9, 25]).

Here we propose a new technique to over-approximate the amount of memory *required* to run a method (or a program). Given a method m with parameters P_m we obtain a parametric upper-bound of the amount of memory necessary to *safely* execute m and all methods it calls, without running out of memory. This expression can be seen as a *pre-condition* stating that m requires that much free memory to be available before executing. To compute this estimation we consider memory deallocation that may occur during the execution of the method. Basically, we adopt a region-based memory management where region lifetime is associated with method lifetime. Then, we model all the potential configurations of regions stacks at run-time in order to obtain an expression of the peak consumption of the method under analysis. This modeling leads to a set of polynomial (and parametric) maximization problems. We solve this problem using a technique based on Bernstein basis technique [13] which yields a parametric solution.

An important feature of the approach is that the obtained expressions are parametric are *easy-to-evaluate*. In fact, the resulting expressions are evaluation trees composed of a small (and known at compile-time) number of polynomials. Among other applications, those expressions can be evaluated at run-time just before execution of the method under analysis -using its actual parameters- in order to obtain an over-approximation which predicts the worst-case amount of memory that may be occupied by the execution of the method.

It is worth mentioning that focusing on scoped-memory also provides a solution to a more general problem since the amount of memory required to run a method in a region-based memory management can be used as an over approximation of the amount required to run a method in the context of a GC that frees objects as soon as they become dead or is triggered by a threshold.

In a few words, the main contributions of this work are:

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

- A formal problem statement of peak consumption that serves as paramount and source of a rigorously-derived model to approximate peak memory consumption using region-based GC.
- An effective solution based on using program invariants and region sizes to produce polynomial maximization problems.
- A parametric (and compile-time) solution that builds run-time evaluation trees by solving off-line maximization problems using Bernstein basis.
- A practical validation of the method consisting of an implementation of this technique into a tool to automatically obtain dynamic memory specifications, together with some benchmarks.

To our knowledge, the technique presented here to infer polynomial bounds of dynamic memory peaks under a region-based manager and its effective computation using Bernstein basis is a novel approach to predict memory requirements.

Outline Sec. 2 informally explains the problem and suggests a solution through an illustrative example. Sec. 3 introduces the basic definitions and assumptions. Sec. 4 presents an effective definition for scoped-based memory management. Sec. 5 proposes a computational procedure to solve the problem. Sec. 6 presents a prototype and shows experimental results on several benchmarks. Sec. 7 discusses related work. Sec. 8 presents conclusions and future work.

2. Informal presentation

Consider the program in Fig. 1. Its call graph, whose nodes are labeled with the set of `new` statements (*allocation sites*) in the corresponding method, is shown in Fig. 2. Let us informally study the behavior of this program with respect to memory occupancy.

```
class Test {
   void m0(H h) {
1:  h.m1();
2:  h.m2(3 * h.mc);
   }
   void  m1() {
3:   B[] dummyArr
        = new B[this.mc][];
4:   for (int i = 1;
         i <= this.mc; i++) {
5:     dummyArr[i-1]= m3(i);
    }
}
   void  m2(int k2) {
6:  B[] m3Arr=m3(k2);
   }

   B[] m3(int n) {
7:   B[] arrB = new B[n];
8:   N l  = new N();
9:   for (int j=1;j <= n;j++) {
10:     arrB[j-1] = l.m4(j);
     }
11:  return arrB;
   }
}
class N {
  N next;  B value;

   B m4(int v) {
12:  N c = new N();
13:  c.value = new B(v);
14:  c.next = this.next;
15:  this.next = c;
16:  return c.value;
   }
}
class H {
  int mc;
}
```

Figure 1. Running example

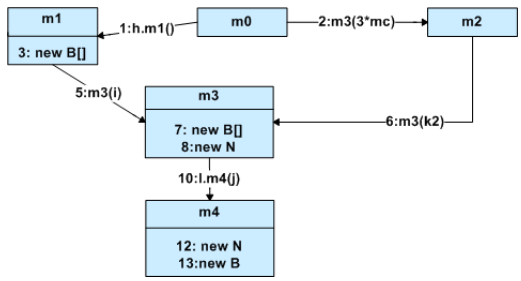

Figure 2. Call graph

Fig. 3 shows the curves of memory occupancy in two runs where $m0$ is invoked with h such that $h.mc = 3$ and $h.mc = 7$, respectively. The abscissa represents program "time" in terms of successive call chains (i.e., paths in the call graph) executed along the run. We write call chains by alternating methods and line numbers, for example, $m0.1.m1.5.m3$. The ordinate represents occupied memory in bytes.

Light gray curves trace the *accumulated* amount of memory allocated by $m0$. Each light gray curve has a maximum value corresponding to the *total* amount of allocated memory. The actual value of this *total allocation* depends on the calling context, that is, the call chain and the actual parameters of the called methods. The total allocation is the *maximal* memory budget for executing $m0$ in a given context. That is, if such amount is available when calling $m0$, it is ensured to execute without ever exhausting the memory, even if no garbage collection is performed during its execution.

Computing total allocation is theoretically difficult [19]. In [7] we proposed a static analysis to estimate it in a conservative manner as a polynomial function of the parameters, regardless of call chains. For instance, the polynomial bounding from above the total allocation of $m0$ is:

$$\texttt{totAlloc}_{m0}(h.mc) \quad = \quad \mathsf{size}(B[])\left(\frac{1}{2}h.mc^2+\frac{9}{2}h.mc\right)$$
$$+\mathsf{size}(B)\left(\frac{1}{2}h.mc^2+\frac{7}{2}h.mc\right)+\mathsf{size}(N)\left(\frac{1}{2}h.mc^2+\frac{9}{2}h.mc+1\right)$$

where $\mathsf{size}(C)$ is the memory occupied by an object of class C. Let us assume for simplicity that $\mathsf{size}(C) = 1$. Then, $\texttt{totAlloc}_{m0}(3) = 52$ and $\texttt{totAlloc}_{m0}(7) = 162$. Dotted lines in Fig. 3 plot these values for the corresponding runs.

Black curves trace memory occupancy under an "ideal" GC that reclaims memory as soon as objects are no longer referenced. Each black curve has a maximum value corresponding to the maximum amount of memory occupied by *live* objects along the run.

This *peak consumption* defines the memory *requirement* of $m0$ in a calling context. That is, $m0$ is guaranteed to exhaust the memory, unless there is *at least* such amount when it is called. For instance, if the free memory when $m0$ is called with $h.mc = 3$ is smaller than 20, $m2$ will end up throwing an exception. Since no other GC can do better than the ideal one, the peak consumption under the latter is indeed the *minimal* memory requirement of the program. That is, it is *definitely* unsafe to run it with a smaller amount of free memory, for whatever GC.

The value and place of occurrence in the program body of the peak heavily depend on the calling context. Fig. 3 shows that the peak reaches two quite different values (near 30 in the first run, and near 70 in the second one), and it is attained by two different call chains ($m0.2.m2\ldots$ and $m0.1.m1\ldots$, respectively).

Total allocation and peak consumption for the ideal GC give "absolute" upper and lower bounds of memory for executing a program, respectively. The actual memory occupancy under a particular GC lies somewhere in between. A first attempt to conservatively approximating its peak consumption would be to resort to total allocation. However, as Fig. 3 illustrates, this results in an overly pessimistic prediction. Obtaining tighter estimations requires knowing *when*, *what*, and *how many* objects are collected.

Therefore, computing peak consumption precisely is not an easy task. Indeed, it is *undecidable*. In practice, as this example shows, the main difficulty relies on the fact that we need to take into account not only all possible calling contexts, as for computing total allocation, to characterize how many objects can be collected, but also the behavior of the underlying GC, to determine when and what objects will actually be collected.

To pursue the analysis of memory occupancy of our example, let us assume the GC behaves as follows: (1) collections take place on method returns, and (2) only dead objects allocated by a method's

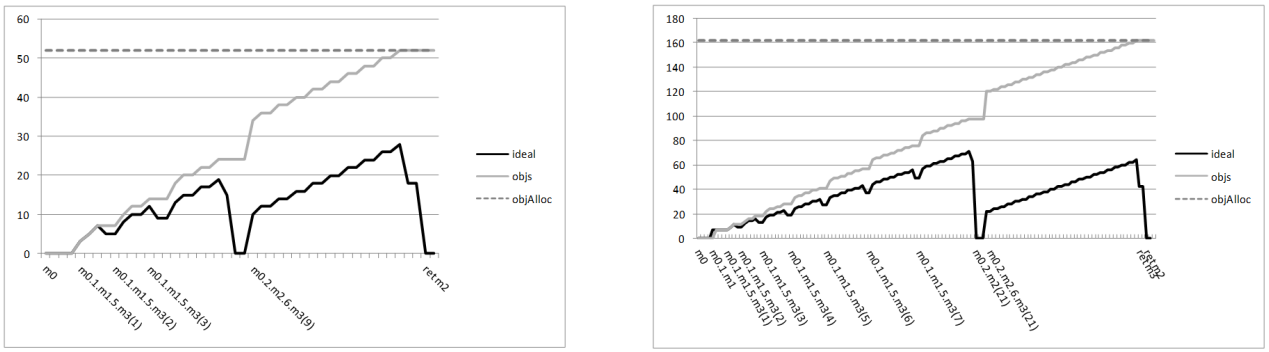

Figure 3. Two traces: $h.mc = 3$ (left), $h.mc = 7$ (right).

transitive callees are collected. This way, the heap is organized as a partition where each "region" has an associated method in the call stack whose return will cause its collection [25].

In our example, objects created in $m4$ by the allocation site at line 12 (denoted $m4.12$) will be in $m3$'s region and collected when $m3$ returns (they cannot be collected when $m4$ returns because they are pointed-to by objects in its calling context, i.e., they *escape* the scope of $m4$). Those from $m4.13$ will belong to the regions of $m2$ or $m1$, depending on the call chain leading to $m4$: they will be associated with $m2$ if the call chain is $m0.2.m2.6.m3.10.m4$, and with $m1$ if it is $m0.1.m1.5.m3.10.m4$. A similar reasoning can be applied for $m3.7$, $m3.8$ and $m1.3$. Objects allocated at $m3.7$ are collected when $m2$ or $m1$ return, depending on the call chain, while those from $m3.8$ are in $m3$'s region, and those from $m1.3$ are in $m1$'s.

Once objects (identified by their allocation sites) collection time have been fixed, that is, the regions identified, their size is bounded by the total allocation restricted to the set of allocation sites of each region. This size can be conservatively approximated using the techniques of [7].

From $m0$, $m0.1.m1.5.m3.10.m4$ and $m0.2.m2.6.m3.10.m4$ are the two (maximal) call chains. Since no object allocated by $m1$ and its transitive callees will live longer than $m1$, and similarly for objects allocated in the call chain from $m2$, the peak consumption for $m0$ is clearly the maximum between the peak along each call chain, which are the sum of the *maximum* sizes of the regions.

Take for instance $m3$. It is called $h.mc$ times from $m1$ (line 5). Each time, it allocates in $m1$'s region (because it returns it to its caller) an array `B[]` of n elements, allocates an object of type `N` in its own region, and calls n times $m4$ which allocates an object of type `N` in $m3$'s region and one of type `B` in $m1$'s (becaus it is returned and assigned to `b3[]` in $m3$). Its region size is therefore $\mathsf{size}(N)(1+n)$, where n goes from 1 to $h.mc$. Clearly, its peak is achieved when n is equal to $h.mc$, at the maximum region size of $\mathsf{size}(N)(1+h.mc)$, that is, the maximum value of the expression $\mathsf{size}(N)(1+n)$ over the interval $1 \leq n \leq h.mc$.

We can apply a similar reasoning to the other methods. Without getting into more details (see [7] and following sections), the polynomials $P_1(h.mc)$ and $P_2(h.mc)$ characterizing the peaks for call chains $m0.1.m1.5.m3.10.m4$ and $m0.2.m2.6.m3.10.m4$ are:

$$\begin{aligned} P_1 &= (\mathsf{size}(B[]) + \mathsf{size}(B))\frac{1}{2}\left(h.mc + h.mc^2\right) + \mathsf{size}(B[])h.mc \\ &\quad + \mathsf{size}(N)(1 + h.mc) \\ P_2 &= (\mathsf{size}(B[]) + \mathsf{size}(B))3h.mc + \mathsf{size}(N)(1 + 3h.mc) \end{aligned}$$

Then, again assuming $\mathsf{size}(C) = 1$ for all C:
$P_1(3) = \max(19, 28) = 28$ and $P_2(7) = \max(71, 64) = 71$, which is consistent with the executions shown in Fig. 3. Moreover, in this case, we can obtain a max-free expression of the peak:

$$\begin{aligned} \max(P_1, P_2) &= \max\left(h.mc^2 + 3h.mc + 1, 9h.mc + 1\right) \\ &= 1 + 3h.mc + h.mc \begin{cases} 6 & \text{if } 0 \leq h.mc \leq 6 \\ h.mc & \text{else} \end{cases} \end{aligned}$$

This example suggests an approach for computing a conservative approximation of the ideal peak consumption, yet tighter than the total allocation. Roughly speaking, the main idea consists in chosing a region-based GC whose peak consumption is computed as the *maximum* over all call chains of the sums of the *maximum* region sizes along the call chain. Since we expect to obtain an expression on terms of input parameters, not a single value, this requires *symbolically* maximizing an expression (in general non-linear) over a domain constraining the values of the parameters (given by a program invariant, e.g., $1 \leq n \leq h.mc$). The rest of the paper is devoted to fixing the mathematical framework and developing a concrete and systematic solution to the problem.

3. Problem statement

3.1 Notation

Let $Prog$ be a sequential Java-like program. Each method $m \in \mathcal{M}$ is characterized by its formal parameters P_m, local variables V_m, and body (list of statements) $B_m \in \mathcal{S}^*$, where $\mathcal{S}$ is the set of statements (`new`, `call`, `ret`, ...). B_m^j is the j-th statement. Wlog, we assume there is a distinguished method under analysis `mua`. Let $Var = \bigcup_{m\in\mathcal{M}} V_m$, $Par = \bigcup_{m\in\mathcal{M}} P_m$, and $\mathcal{V}$ be a set of possible values.

We assume that a *complete* call graph, $\mathsf{CG} = (\mathcal{M}, \mathcal{E})$, $\mathcal{E} \subseteq \mathcal{M} \times \mathbb{N} \times \mathcal{M}$, can be computed at compile-time (*closed-world* assumption)[1]. CG_m is the subgraph of CG rooted at $m \in \mathcal{M}$. We will heavily manipulate the control part of stacks in the rest of the paper (control stacks, thereafter). The possible configurations of control stacks the program may reach are statically over-approximated by the set of paths in the call graph, which we call call chains. That is, $\Pi \subseteq (\mathcal{M} \times \mathbb{N})^* \times \mathcal{M}$ is the set of *call chains*: $\mathsf{cch} \in \mathsf{CCH}$ iff cch is a path in CG. $|\mathsf{cch}|$ is the *length* of cch. Actually, control stacks are much like call chains but, since we want them to represent integrally control information of a program state, we also include program counter as the last element. That is, $\mathsf{CST} \subseteq (\mathcal{M} \times \mathbb{N})^*$ is the set of *control stacks*: $\mathsf{cst} = \mathsf{cch}.n \in \mathsf{CST}$ iff $\mathsf{cch} \in \mathsf{CCH}$ and $n \in [1, |B_{\mathsf{cch}_{|\mathsf{cch}|}}|]$. We define $|\mathsf{cst}|$ to be $|\mathsf{cch}|$. For all $x, x' \in \mathsf{CST} \cup \mathsf{CCH}$, $k \geq 1$, $x[1..k]$ is the prefix (call chain) $m_1 \dots m_k \in \mathsf{CCH}$ of x, x_k is the k-th method, $x \equiv_k x'$ iff $x[1..k] = x'[1..k]$, and $x \equiv x'$ iff $|x| = |x'|$ and $x \equiv_{|x|} x'$. $\mathsf{CCH}_m \triangleq \{\mathsf{cch} \in \mathsf{CCH} \mid \mathsf{cch}_1 = m\}$.

[1] In general, because of polymorphism, CG is a *conservative* approximation of all possible concrete method calls.

3.2 Semantics

Firstly, semantics will be defined from a functional perspective which means that no dead object collection is performed. Semantics is given by means of traces of a state transition system. In this setting, a dynamic memory manager behavior will be modeled as a trace transformer, in the sense that they specify how they would get rid of dead objects of the heap.

A program state $\sigma \in \Sigma$ consists of a *control stack* $\mathsf{cst}(\sigma) \in \mathsf{CST}$, a *data stack* $\mathsf{v}(\sigma) \in [(\mathit{Var} \cup \mathit{Par} \mapsto \mathcal{V})^*]$ of valuations of variables and parameters, and a *heap* $\mathsf{h}(\sigma) \subseteq \mathcal{O} \times \mathcal{F} \times \mathcal{O}, \mathcal{O} \subseteq \mathcal{V}$, modeled as a directed graph of *objects* where edges represent field (or array) dereferences. $\mathsf{m}_k(\sigma)$, $\mathsf{pc}_k(\sigma)$, and $\mathsf{v}_k(\sigma)$ denote the k-th method, control location and data valuation, respectively[2].

An object $o \in \mathcal{O}$ is *live* in $\mathsf{h}(\sigma)$ iff it is *reachable* from a variable defined in the state, that is, there exists k such that $\mathsf{v}_k(\sigma)(v)$ has a path to o in $\mathsf{h}(\sigma)$. The set of live objects of σ is denoted $\mathsf{live}(\sigma)$. We define $\mathsf{dead}(\sigma)$ to be $\mathsf{h}(\sigma) \setminus \mathsf{live}(\sigma)$.

Let $\mathsf{size}(o) \geq 1$ be the amount of memory occupied by o. Given heaps $\mathcal{H}, \mathcal{H}'$, $\mathsf{used}^{\mathcal{H}}(\mathcal{H}') \triangleq \sum_{o \in \mathcal{H}' - \mathcal{H}} \mathsf{size}(o)$. That is the amount of memory occupied by objects in $\mathcal{H}'$ which are not in $\mathcal{H}$: We write $\mathsf{used}^{\mathcal{H}}(\sigma)$ as a shorthand for $\mathsf{used}^{\mathcal{H}}(\mathsf{h}(\sigma))$.

The semantics of $Prog$ is given by a deterministic transition system $\langle \Sigma, \rightarrow \rangle$, where for all $\sigma \in \Sigma$, $\mathsf{cst}(\sigma) = \mathsf{cst}.(\mathsf{mua}, n).\mathsf{cst}'$, with $\mathsf{cst}, \mathsf{cst}' \in \mathsf{CST}$, $\mathsf{mua} \notin \mathsf{cst}$. $\Sigma_{\mathsf{mua}} \subseteq \Sigma$ is the set of *initial* states: $\bar{\sigma} \in \Sigma_{\mathsf{mua}}$ iff $\mathsf{cst}(\bar{\sigma}) = \mathsf{cst}.(\mathsf{mua}, 0)$. That is, this set encompasses all possible invocations of mua[3].

The transition relation $\rightarrow$ is defined by the operational semantics *without* deleting dead objects from the heap. For $\sigma \in \Sigma$, $\mathsf{succ}(\sigma)$ is such that $\sigma \rightarrow \mathsf{succ}(\sigma)$. For all $\sigma, \sigma' \in \Sigma$ which are equal except for the set of dead objects, $\mathsf{succ}(\sigma)$ is equal to $\mathsf{succ}(\sigma')$, except for the set of dead objects. Also, it is not possible to *forge* pointers: $\mathsf{dead}(\sigma) \subseteq \mathsf{dead}(\mathsf{succ}(\sigma))$.

A *run* is a sequence $\rho = \sigma_0, \sigma_1 \ldots \in \Sigma^*$, with $\sigma_i \rightarrow \sigma_{i+1}$, and $\sigma_0 \in \Sigma_{\mathsf{mua}}$. We denote ρ_i the state corresponding to the i-th element of ρ. Since the transition system is deterministic, a run is *uniquely* determined by its initial state. Then we use $\bar{\sigma}_i$ to denote the i-element of the run starting at $\bar{\sigma}$.

A *dynamic memory manager* is a function $\Gamma : \Sigma^* \mapsto \Sigma^*$ that removes from the heap (a subset of) dead objects, that is, for all runs ρ, $\Gamma(\rho)$ is such that for all $i \geq 0$, $\mathsf{dead}(\Gamma_i(\rho)) \subseteq \mathsf{dead}(\rho_i)$ (where $\Gamma_i(\rho)$ is the i-element of $\Gamma(\rho)$) while keeping the rest of the state unchanged. We write $\bar{\sigma}_i^{\Gamma}$ to denote the state corresponding to the i-element of the run starting at $\bar{\sigma}$ and collected using Γ.

The goal of defining the semantics without dead-object collection[4] is to be able to compare dynamic memory managers with respect to their freeing power: we say that Γ *is more efficient* than $\hat{\Gamma}$ iff $\mathsf{dead}(\Gamma_i(\rho)) \subseteq \mathsf{dead}(\hat{\Gamma}_i(\rho))$. This implies: $\mathsf{used}^{\mathcal{H}}(\Gamma_i(\rho)) \leq \mathsf{used}^{\mathcal{H}}(\hat{\Gamma}_i(\rho))$, for all $\rho, i, \mathcal{H}$.

Let `ideal` denote the dynamic memory manager such that $\mathsf{dead}(\mathtt{ideal}_i(\rho)) = \emptyset$. Thus, `ideal` is the *most efficient* one, i.e., $\mathsf{used}^{\mathcal{H}}(\mathtt{ideal}_i(\rho)) \leq \mathsf{used}^{\mathcal{H}}(\Gamma_i(\rho))$, for all $\Gamma, \rho, i, \mathcal{H}$.

3.3 Peak consumption

Given Γ and $\bar{\sigma} \in \Sigma_{\mathsf{mua}}$, we are interested in knowing the *maximum* space occupied by all objects *created* (i.e., not in $\mathsf{h}(\bar{\sigma})$) along the run starting at $\bar{\sigma}$. We call this maximum the *peak*:

$$\mathsf{peak}_{\Gamma}(\bar{\sigma}) \quad \triangleq \quad \max_i \mathsf{used}^{\mathsf{h}(\bar{\sigma})}(\bar{\sigma}_i^{\Gamma}) \tag{1}$$

[2] To homogeneously model a state, the current program location (method and statement) is the top of the stack.

[3] The first one in the case of mua being recursive.

[4] We use "to free" or "to collect" interchangeably.

Note that, if Γ is more efficient than $\hat{\Gamma}$, then $\mathsf{peak}_{\Gamma}(\bar{\sigma}) \leq \mathsf{peak}_{\hat{\Gamma}}(\bar{\sigma})$, for all $\bar{\sigma} \in \Sigma_{\mathsf{mua}}$, for all Γ.

3.4 Peak in scoped-memory management

We assume there is a *scoped-memory* manager that reclaims memory when methods return. It is only allowed to collect dead objects created *during* the execution of the method (and the methods it transitively calls). Objects created in an outer scope cannot be collected by the current method, but only reclaimed by its ancestors. Objects are placed in regions associated with methods. Objects in a region can point to objects in the same region or a parent one corresponding to an ancestor method in the call stack. This scoping rule can be satisfied by inferring regions at compile-time (e.g.,[26, 25]).

Let $r \subseteq \mathcal{O}$ be a region. We assume every object is assigned to a region and there is one (possibly empty) region per method occurrence in the call stack. For any $\sigma \in \Sigma$, $\mathsf{rst}(\sigma)$ is the region stack of σ. There is a one to one correspondence between call stack and regions stack. That is, one (possibly empty) region per method occurrence in the call stack. The set of nodes of the heap is a *disjoint* union of regions. A region-based memory manager Γ eliminates the region on top of the region stack on method return. To ensure that only dead objects are removed, the assignment of new objects into regions is done in such a way that on the return of the method on the top of the stack, all objects in the top region are dead. This holds because inter-region references comply with scoping rules. From Eq. (1) and the above, it follows that:

$$\mathsf{peak}_{\Gamma}(\bar{\sigma}) \; = \; \max_i \sum_{1 \leq k \leq |\mathsf{cst}(\bar{\sigma}_i)|} \mathsf{used}^{\mathsf{h}(\bar{\sigma})}(\mathsf{rst}_k(\bar{\sigma}_i^{\Gamma}))$$

where k is a position in the call stack and rst_k its associated region.

Notice that under this memory model that new objects are either allocated in regions created after the observed mua invocation or they are allocated in regions already stacked when that mua invocation occurs. Furthermore, new regions consist integrally of new objects and are the only regions that may be popped in the period of calculation of peak. Old regions may only grow and they do that by incorporating new objects which lifetimes exceed mua invocation lifetime-time. Thus, we are interested in distinguishing control stacks and regions preceding the execution of mua from the ones involved in its execution since they satisfy different properties. Recall that for all $\sigma \in \Sigma$, $\mathsf{cst}(\sigma) = \mathsf{cst}.(\mathsf{mua}, n).\mathsf{cst}'$, $\mathsf{mua} \notin \mathsf{cst}$. Hereinafter, $\mathsf{cst}{\downarrow}(\sigma)$ denotes cst, $\mathsf{cst}{\uparrow}(\sigma)$ denotes $(\mathsf{mua}, n).\mathsf{cst}'$, $\mathsf{rst}{\downarrow}(\sigma)$ (resp., $\mathsf{v}{\downarrow}(\sigma)$) and $\mathsf{rst}{\uparrow}(\sigma)$ (resp., $\mathsf{v}{\uparrow}(\sigma)$) denote the corresponding region (resp., data-value) stacks. Observe that, for all i, j, $\mathsf{cst}{\downarrow}(\bar{\sigma}_i) = \mathsf{cst}{\downarrow}(\bar{\sigma}_j)$.

For all i, Γ ensures $\mathsf{h}(\bar{\sigma}) \cap \mathsf{rst}{\uparrow}(\bar{\sigma}_i^{\Gamma}) = \emptyset$ and $\mathsf{h}(\bar{\sigma}) \subseteq \mathsf{rst}{\downarrow}(\bar{\sigma}_i^{\Gamma})$. Therefore, we can split the above equation to distinguish the regions created by the execution of the mua from the regions already created when mua is invoked.

$$\mathsf{peak}_{\Gamma}(\bar{\sigma}) \quad = \quad \mathsf{peak}{\downarrow}_{\Gamma}(\bar{\sigma}) + \mathsf{peak}{\uparrow}_{\Gamma}(\bar{\sigma}) \tag{2}$$

where

$$\mathsf{peak}{\downarrow}_{\Gamma}(\bar{\sigma}) \; = \; \max_i \sum_{1 \leq k \leq |\mathsf{cst}{\downarrow}(\bar{\sigma}_i)|} \mathsf{used}^{\mathsf{h}(\bar{\sigma})}(\mathsf{rst}{\downarrow}_k(\bar{\sigma}_i^{\Gamma})) \tag{3}$$

$$\mathsf{peak}{\uparrow}_{\Gamma}(\bar{\sigma}) \; = \; \max_i \sum_{1 \leq k \leq |\mathsf{cst}{\uparrow}(\bar{\sigma}_i)|} \mathsf{size}(\mathsf{rst}{\uparrow}_k(\bar{\sigma}_i^{\Gamma}))) \tag{4}$$

with $\mathsf{size}(r) = \sum_{o \in r} \mathsf{size}(o)$. Notice in that in Eq. 4 we replace used by the sum of the sizes of all objects in the region. This is because regions created since mua *only* contain newly created objects (i.e. $\mathsf{h}(\bar{\sigma}) \cap \mathsf{rst}{\uparrow}(\bar{\sigma}_i^{\Gamma}) = \emptyset$). In contrast, regions existing prior to (the first call to) mua contain objects created since mua and objects created before (i.e. $\mathsf{h}(\bar{\sigma}) \subseteq \mathsf{rst}{\downarrow}(\bar{\sigma}_i^{\Gamma})$).

The rest of this paper is devoted to develop a computational technique to approximate $\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma)$ and $\mathsf{peak}\!\downarrow_\Gamma(\bar\sigma)$.

4. Computing peak

In this section we sketch our technique to obtain a (static) parametric prediction of the peak consumption of a method under out scoped-memory management. We will start from the right-hand side of Eq. 4 which depends on program states and apply a series of approximation steps to obtain an inequality which only depends on the parameters P_{mua} and the subgraph $\mathsf{CG}_{\mathsf{mua}}$ of the call graph rooted at the method under analysis mua.

Σ can be partitioned according to call chains in CCH: $\sigma, \sigma' \in \Sigma$, are in the same class iff $\mathsf{cst}(\sigma) \equiv \mathsf{cst}(\sigma')$. Indeed, a class is determined by a call chain cch, that is, σ is in the class of cch iff $\mathsf{cst}(\sigma) \equiv \mathsf{cch}$. Besides, $\mathsf{cst}\!\uparrow(\sigma) \equiv \mathsf{cch}$ for some $\mathsf{cch} \in \mathsf{CCH}_{\mathsf{mua}}$.

Then, from Eq. 4 it follows that:

$$\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma) \;=\; \max_{\mathsf{cch}\in\mathsf{CCH}_{\mathsf{mua}}} \max_{\substack{i \text{ st.}\\ \mathsf{cch}\equiv\mathsf{cst}\uparrow(\bar\sigma_i)}} \sum_{1\le k\le|\mathsf{cch}|} \mathsf{size}(\mathsf{rst}\!\uparrow_k(\bar\sigma_i^\Gamma)))$$

The right-hand side of this equation can be over-approximated by summing up the maximum region sizes along $\rho^{\bar\sigma}$

$$\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma) \;\le\; \max_{\mathsf{cch}\in\mathsf{CCH}_{\mathsf{mua}}} \sum_{1\le k\le|\mathsf{cch}|} \max_{\substack{i \text{ st.}\\ \mathsf{cch}\equiv\mathsf{cst}\uparrow(\bar\sigma_i)}} \mathsf{size}(\mathsf{rst}\!\uparrow_k(\bar\sigma_i^\Gamma)))$$

and then, by considering a partition for each stack-depth: $\sigma, \sigma' \in \Sigma$ are in the same class for depth k, iff $\mathsf{cst}\!\uparrow(\sigma) \equiv_k \mathsf{cst}\!\uparrow(\sigma')$. Thus:

$$\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma) \;\le\; \max_{\mathsf{cch}\in\mathsf{CCH}_{\mathsf{mua}}} \sum_{1\le k\le|\mathsf{cch}|} \max_{\substack{i \text{ st.}\\ \mathsf{cch}\equiv_k\mathsf{cst}\uparrow(\bar\sigma_i))}} \mathsf{size}(\mathsf{rst}\!\uparrow_k(\bar\sigma_i^\Gamma))))$$

It follows that computing an over-approximation of the peak reduces to computing *maximum* region sizes.

4.1 Approximating region sizes

Given a method $m \in \mathcal{M}$, let $\mathsf{rsize}_m(P_m)$ be a function that *over-approximates* the size of *any* region of m: for all σ, k, such that $\mathsf{m}_k(\sigma) = m$, $\mathsf{size}(\mathsf{rst}_k(\sigma)) \le \mathsf{rsize}_m(\mathsf{v}_k(\sigma)(P_m))$. Hence, we can over-approximate the previous inequality by replacing $\mathsf{size}(\mathsf{rst}\!\uparrow_k(\bar\sigma_i^\Gamma))$ (which depends on states) by $\mathsf{rsize}_m(P_m)$ (which only depends on m's parameters):

$$\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma) \;\le \max_{\mathsf{cch}\in\mathsf{CCH}_{\mathsf{mua}}} \sum_{1\le k\le|\mathsf{cch}|} \max_{\substack{i \text{ st.}\\ \mathsf{cch}\equiv_k\mathsf{cst}\uparrow(\bar\sigma_i)}} \mathsf{rsize}_m(\mathsf{v}\!\uparrow_k(\bar\sigma_i)(P_m)) \quad (5)$$

where $m = \mathsf{cch}_k$. Since $\mathsf{v}\!\uparrow_k(\bar\sigma_i^\Gamma) = \mathsf{v}\!\uparrow_k(\bar\sigma_i)$ this inequality does not depend on Γ.

$\mathsf{rsize}_m(P_m)$ can be computed, in may cases, using [7] or it may be specified by the developer. For $m \in \mathcal{M}$, let $\mathcal{A}$ be the set of new statements reachable from m along a call chain in CCH_m. Any region associated with m is composed of objects created by a subset $\mathcal{R}_m \subseteq \mathcal{A}$[5]. [7] constructs a polynomial over-approximating (in terms of P_m) the number of times each $\alpha \in \mathcal{R}_m$ could be executed in a run from m.

Example In our running example:

$$\begin{aligned}
\mathcal{R}_{m0}(this,h) &= \quad \mathcal{R}_{m4}(this,v) = \{\}\\
\mathcal{R}_{m1}(this) &= \{m1.3, m1.5.m3.7, m1.5.m3.10.m4.13\}\\
\mathcal{R}_{m2}(this,k2) &= \{m2.6.m3.7, m1.6.m3.10.m4.13\}\\
\mathcal{R}_{m3}(this,n) &= \{m3.8, m3.10.m4.12\}
\end{aligned}$$

[5] $\mathcal{R}_m$ can be computed using region synthesis techniques like [26].

and [7] gives:

$$\begin{aligned}
\mathsf{rsize}_{m0}(this,h) &= \mathsf{rsize}_{m4}(this,v) = 0\\
\mathsf{rsize}_{m1}(this) &= \mathsf{size}(B[])this.mc\\
&\quad +(\mathsf{size}(B[])+\mathsf{size}(B))(\frac{1}{2}this.mc^2+\frac{1}{2}this.mc)\\
\mathsf{rsize}_{m2}(this,k2) &= (\mathsf{size}(B[])+\mathsf{size}(B))k2\\
\mathsf{rsize}_{m3}(this,n) &= \mathsf{size}(N)+\mathsf{size}(N)n
\end{aligned}$$

where $\mathsf{size}(C)$ is the size of objects of class C. □

4.2 Approximating maximum region sizes

Eq. 5 still relies on $\bar\sigma_i^\Gamma$ to determine a valuation of P_m when evaluating $\mathsf{rsize}_m(P_m)$. Observe that in Eq. 5 regions are classified by call-stack prefixes and each element in the sum corresponds to the maximum size of a region over all states matching the call-stack. Data-values of any $\sigma \in \Sigma$, with $\mathsf{cst}(\sigma) \equiv \mathsf{cch}$, can be over-approximated by an invariant $\mathcal{I}^{\mathsf{cch}}_{\mathsf{mua}}$ binding P_{mua} with the parameters of *all* methods in cch. In particular, such a *binding invariant* can be used to constrain the *calling context* of m, that is, the values of P_m in terms of P_{mua}. Binding invariants can be obtained, for instance, from local program invariants as in [7].

Example A binding invariant $\mathcal{I}^{m0.1.m1.5.m3}_{\mathsf{mua}}$ in our example is $\{this_{m1} = h, 1 \le i \le this_{m1}.mc, n = i\}$ which can be simplified as $\{1 \le n \le h.mc\}$ □

For $\mathsf{cch} = \mathsf{cst}.m \in \mathsf{CCH}_{\mathsf{mua}}$, the maximum value of $\mathsf{rsize}_m(P_m)$ in *all* σ with $\mathsf{cst}.m \equiv_k \mathsf{cst}\!\uparrow(\sigma)$ for some k, can be over-approximated by its maximum value over a binding invariant $\mathcal{I}^{\mathsf{cst}.m}_{\mathsf{mua}}$. Now, let:

$$\begin{aligned}
\mathsf{maxrsize}^{\mathsf{cst}.m}_{\mathsf{mua}}(P_{\mathsf{mua}}) \;\triangleq\; & \textit{Maximize } \mathsf{rsize}_m(P_m) \qquad (6)\\
& \textit{sbj.to } \mathcal{I}^{\mathsf{cst}.m}_{\mathsf{mua}}(P_{\mathsf{mua}}, P_m, W)
\end{aligned}$$

where W are local variables appearing in methods in cst. Since a binding invariant is a conservative approximation of the set of potential states, it follows that

$$\max_{\substack{i \text{ st.}\\ \mathsf{cch}\equiv_k\mathsf{cst}(\bar\sigma_i)}} \mathsf{rsize}_m(\mathsf{v}\!\uparrow_k(\bar\sigma_i))(P_m)) \le \mathsf{maxrsize}^{\mathsf{cch}[1..k]}_{\mathsf{mua}}(\mathsf{v}\!\uparrow_1(\bar\sigma)(P_{\mathsf{mua}}))$$

We define:

$$\mathsf{mem}\!\uparrow_{\mathsf{mua}}(P_{\mathsf{mua}}) \;\triangleq \max_{\mathsf{cch}\in\mathsf{CCH}_{\mathsf{mua}}} \sum_{1\le k\le|\mathsf{cch}|} \mathsf{maxrsize}^{\mathsf{cch}[1..k]}_{\mathsf{mua}}(P_{\mathsf{mua}}) \quad (7)$$

Thus, we can prove by construction that:

Lemma 1. For all Γ and $\bar\sigma \in \Sigma_{\mathsf{mua}}$,

$$\mathsf{peak}\!\uparrow_\Gamma(\bar\sigma) \le \mathsf{mem}\!\uparrow_{\mathsf{mua}}(\mathsf{v}\!\uparrow_1(\bar\sigma)(P_{\mathsf{mua}})).$$

Sec.5 develops an effective procedure to compute $\mathsf{maxrsize}$.

Example 1 Table 1 shows $\mathsf{maxrsize}^{\mathsf{cch}}_{m0}$ for the example of Fig. 1. Since that every call chain is a prefix of $m0.1.m1.5.m3.10.m4$ or $m0.2.m2.6.m3.10.m4$ it suffices to consider the sum only for these two call chains. Therefore, we have that:

$$\begin{aligned}
\mathsf{mem}\!\uparrow_{m0}(this,h) \;=& \max\{(\mathsf{size}(B[])+\mathsf{size}(B))(\frac{1}{2}h.mc^2+\frac{1}{2}h.mc)\\
&+\mathsf{size}(B[])h.mc+\mathsf{size}(N)(1+h.mc),\\
&(\mathsf{size}(B[])+\mathsf{size}(B))3h.mc+\mathsf{size}(N)(1+3h.mc)\}
\end{aligned}$$

Assume, for simplicity, that $\mathsf{size}(C) = 1$ for all C. Then:

$$\begin{aligned}
&\mathsf{mem}\!\uparrow_{m0}(this,h) \quad =\\
&= \max\{h.mc^2+2h.mc+1+h.mc, 6h.mc+1+3h.mc\}\\
&= 1+3h.mc+\max\{h.mc^2, 6h.mc\}\\
&= 1+3h.mc+\begin{cases} h.mc^2 & \text{if } h.mc<0 \vee h.mc>6\\ 6h.mc & \text{if } 0\le h.mc\le 6\end{cases}
\end{aligned}$$

Table 1. maxrsize for the running example.

$\mathsf{cst}.m$	$\mathcal{I}_{m0}^{\mathsf{cst}.m}$	$\mathsf{maxrsize}_{m0}^{\mathsf{cst}.m}(this, h)$
$m0$	true	0
$m0.1.m1$	$\{this_{m1} = h\}$	$(\mathsf{size}(B[]) + \mathsf{size}(B)).(\frac{1}{2}h.mc^2 + \frac{1}{2}h.mc) + \mathsf{size}(B[])h.mc$
$m0.1.m1.5.m3$	$\{h.mc \geq 1, this_{m1} = h, 1 \leq i \leq this_{m1}.mc, n = i\}$	$\mathsf{size}(N) + \mathsf{size}(N)h.mc$
$m0.1.m1.5.m3.10.m4$	$\{h.mc \geq 1, this_{m1} = h, 1 \leq i \leq this_{m1}.mc, n = i, 1 \leq j \leq n, v = j\}$	0
$m0.2.m2$	$\{k_2 = 3h.mc\}$	$(\mathsf{size}(B[]) + \mathsf{size}(B))3h.mc$
$m0.2.m2.6.m3$	$\{k_2 = 3h.mc, n = k_2\}$	$\mathsf{size}(N) + \mathsf{size}(N)3h.mc$
$m0.2.m2.6.m3.10.m4$	$\{h.mc \geq 1, k_2 = 2h.mc, n = k_2, 1 \leq j \leq n, this_{m4} = l, v = j\}$	0

Fig. 4 shows that $\mathsf{mem}\uparrow$ (dotted light gray line) is an upper-bound of the actual memory requirements of the example of Fig. 1 (dark solid line). It also shows how regions are created when methods are invoked and released when they return. Light gray bars depict rsize_{m1}, dark gray bars rsize_{m3}, and the others rsize_{m2}. Notice that, in fact, $\mathsf{mem}\uparrow$ is in this case a precise approximation of the requirements using a region-based memory manager. It also shows that once maxrsize is computed (dotted red rectangles on the left figure) the number of expressions to compare (maximums operations) is finite and related to the number of paths in CG. □

Example of dynamic binding The approach deals with polymorphic method calls because they are captured by a conservative call graph. Fig. 5 shows a small fragment of code that performs a poly-

```
class Test{
  int n;
  void m(A a){
1: for(i=0;i<n<i++){
2:   a.f(i);
3:   b = new B();
   }
  }
}
```

```
class A1 extends A {
  void f(int p){
  1: a = new C();
  }
}
class A2 extends A{
  void f(int p){
1:a = new D();
  }
}
```

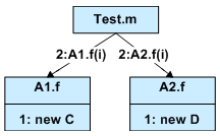

Figure 5. Example of inheritance

morphic call. A call graph for this program includes call chains $test.m.2.A1.f.1$ and $test.m.2.A2.f.1$. The technique "merges" these branches by applying a maximum operation between them. Thus:

$$\begin{aligned}\mathsf{mem}\uparrow_m(this, a) &= \mathsf{rsize}_m(this.n) + \max\{\mathsf{maxrsize}_m^{test.m.2.A1.f.1}, \\ &\qquad \mathsf{maxrsize}_m^{test.m.2.A2.f.1}\} \\ &= this.n * \mathsf{size}(B) + \max\{\mathsf{size}(C), \mathsf{size}(D)\}\end{aligned}$$

where $\{0 \leq i < this.n\}$ is the binding invariant of both call chains. □

Example 3 Eq. 7 introduces an additional source of over approximation by adding the maximum of each m-region along a call chain. However, it can be the case that two regions cannot both reach the maximum size at the same time. Consider the example in Fig. 6 assuming that $N \leq 10$. The call chains in this example rooted at $m1$ are $m1.2.m2$ and $m1.2.m2.5.m3$.

```
void m1(int N){
1:for(h=1;h<=N;h++)
2:  m2(h);
}
```

```
void m2(int k){
3:B[] b = new B[11-k];
4:for(j=1;j<=k;j++)
5:  m3(j);
}
```

```
void m3(int c){
6:for(i=1;i<=2*c;
      i++)
7: A a=new A();
}
```

Figure 6. Over-approximation due to Eq. 7.

$\mathsf{maxrsize}_{m1}^{m1} = 0$ because $m1$ does not capture any object. $m2$-region size depends on the expression $11 - k$ (line 3). It means that its maximum size is reached when $k = 1$. Thus, the maximum $m2$-region when constrained by $m1.2.m2$ is $\mathsf{maxrsize}_{m1}^{m1.2.m2}(N) = 10$ obtained by assigning $h = 1$. On the other hand, $m3$-region size is proportional to the value of c. However, the variable c reaches its maximum value when j does. This is exactly when $j = k$ being the maximum value of k reached when $h = n$. Thus, $\mathsf{maxrsize}_{m1}^{m1.2.m2.5.m3}(N) = 2N$ and is obtained assigning $h \mapsto N, k \mapsto h, j \mapsto k, c \mapsto j$. However, as we have seen, the assignment $h \mapsto N$ does not maximize the size of $m2$-regions. Thus, both situations cannot happen at the same time and adding the resulting maxrsize expressions leads to an over-approximation. This problem is shown graphically in Fig. 7. □

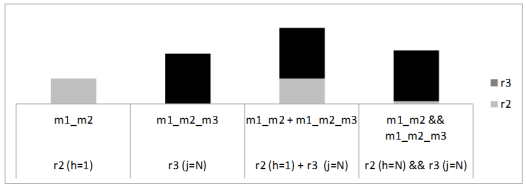

Figure 7. Over-approximation due to Eq. 7

4.3 Computing peak↓

So far, we have presented a technique to over-approximate $\mathsf{peak}\uparrow$ which corresponds to the maximum amount of memory occupied by regions created since mua. However there may be objects that need to be allocated in regions that exist before the invocation of mua. This part of the equation is modeled by $\mathsf{peak}\downarrow$. We know the size of a region monotonically increases until the region is removed. In this case, pre-mua regions are not collected during the observed executions. Thus, we just need to know how many objects will be associated to pre-mua regions during mua execution. To deal with this situation we introduce a new function denoted $\mathsf{mem}\downarrow_m$ which yields an over-approximation, in terms of P_m, of the amount of dynamic memory allocated by objects created during the execution of m that cannot be released when m terminates, meaning that they have to be allocated in outer scopes. $\mathsf{mem}\downarrow_m$ provides useful information to callers of m as they must consider that the call to m will require some additional space of their own regions. A technique to automatically infer the amount of memory that escapes a method can be found in [7]. In our example no object escapes $m0$ meaning that $\mathsf{mem}\downarrow_{m0}(this, h) = 0$.

Observe that, to approximate $\mathsf{peak}\downarrow(\bar{\sigma})$, we only need to take into account the amount of memory escaping mua as escape information is absorbent. By absorbent we mean that any object escaping the scope of a method m', transitively called by m, is eventually captured by some method in the call stack $m \ldots m'$.
Therefore, $\mathsf{mem}\downarrow$ has (by definition) the following property:

Lemma 2. *For all* Γ *and* $\bar{\sigma} \in \Sigma_{mua}$,

$$\mathit{peak}\downarrow_\Gamma(\bar{\sigma}) \leq \mathit{mem}\downarrow_{mua}(\mathit{vl}_1(\bar{\sigma})(P_{mua})).$$

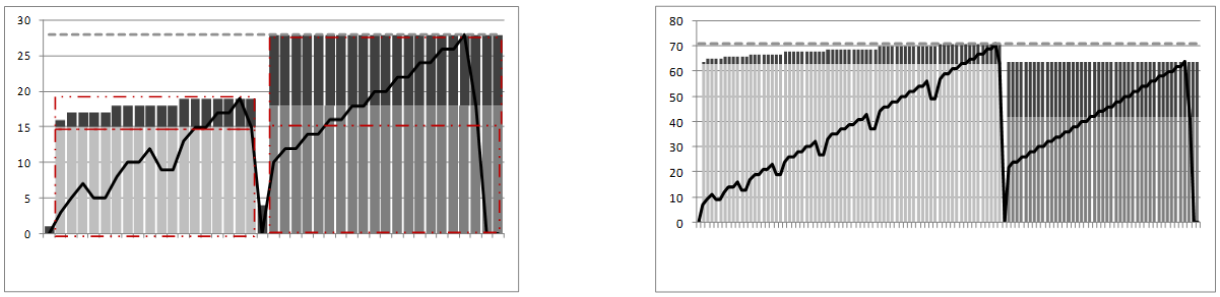

Figure 4. Ideal, rsize, and mem↑ for $m0(this, h)$ ($h.mc = 3$,left) and $m0(this, h)$ ($h.mc = 7$, right).

Finally, we can prove using lemmas 1 and 2:

Theorem 1. *For all* Γ *and* $\bar{\sigma} \in \Sigma_{mua}$,

$$peak_{\Gamma}(\bar{\sigma}) \leq [mem\uparrow_{mua} + mem\downarrow_{mua}](\mathcal{V}\uparrow_1(\bar{\sigma})(P_{mua})).$$

5. Computing maxrsize and mem↑

The definition of maxrsize (Eq. 6) formulates a non-linear maximization problem, where rsize represents the input and the binding invariant for the control stack represents the restriction. The solution of Eq. 6 is an expression in terms of P_{mua}. Since our goal is to avoid expensive runtime computations, we need to perform off-line reduction as much as possible at compile time. Off-line calculation also means that the problem must be solved symbolically. As a consequence, it is not possible to resort to non-linear numerical optimization.

5.1 Computing maxrsize

Recall that our original definition of rsize_m admits parameters of any type (in particular objects). To apply the technique that follows, we assume that rsize_m is actually a polynomial evaluated on integer values, together with a mapping from path expressions rooted at P_m to the actual variables in the polynomial. For instance: $\mathsf{rsize}_m(p1, p2) = p1.a.length + p1.b.val * p1.a.length + p2.size()$ would be replaced by $\mathsf{rsize}'_m(p1_1, p1_2, p2_1) = p1_1 + p1_2 * p1_1 + p2_1$ where $p1_1 = p1.a.length$, $p1_2 = p1.b.val$ and $p2_1 = p2.size()$. Something similar is done for linear invariants which also require variables of integer type.

Thus, if rsize_m is expressed as a polynomial (which is the case if it is computed using [7]) and binding invariants are expressed using linear constraints we can resort to [13] which proposes an extension of Bernstein expansion for symbolically bounding the range of a polynomial over a linear domain. The idea is as follows: Bernstein polynomials form a basis for the space of polynomials, such that the coefficients of a polynomial in Bernstein basis give minimum and maximum bounds on the polynomial values. Here, we do not go into the details, which can be found in [14]. Let $\vec{x} = (x_1, \ldots, x_k)$ be a vector of variables, and $\vec{p} = (p_1, \ldots, p_n)$ a vector of parameters. Given a polynomial $\phi \in \mathbb{Q}[\vec{x}]$, and a convex polytope $I \in \mathbb{Q}^{|\vec{x}\times\vec{p}|}$, there is a function

$$\mathsf{Ber} : \mathbb{Q}[\vec{x}] \times \mathbb{Q}^{|\vec{x}\times\vec{p}|} \mapsto \mathbf{2}^{\mathbb{Q}^{|\vec{p}|}\times\mathbf{2}^{\mathbb{Q}[\vec{p}]}}$$

that yields a set $\{(D_i, \mathcal{C}_i)\}_{i\in[1,l]}$ where $D_i \in \mathbb{Q}^{|\vec{p}|}$ is a linear domain and $\mathcal{C}_i \subseteq \mathbb{Q}[\vec{p}]$ is a set of "candidate" polynomials such that, for all $\mathbf{p} \in \mathbb{Q}^{|\vec{p}|}$:

$$\max\{\phi(\mathbf{x}) \mid I(\mathbf{p}, \mathbf{x})\} \leq \begin{cases} \max\{q(\mathbf{p}) \in \mathcal{C}_1\} & \text{if } D_1(\mathbf{p}) \\ \ldots & \\ \max\{q(\mathbf{p}) \in \mathcal{C}_l\} & \text{if } D_l(\mathbf{p}) \end{cases}$$

Example Applying Bernstein to $\phi(n) = n^2 - 1$ for the polytope $I(P_1, P_2, n, i) = \{1 \leq i \leq P_1 + P_2, i \leq 3P_2, n = i\}$ yields the following result:

$$D_1\colon \{2P_2 \geq P_1\} \quad \mathcal{C}_1\colon \{(P_1 + P_2)^2 - 1, P_2 + P_1 - 1\}$$
$$D_2\colon \{2P_2 \leq P_1\} \quad \mathcal{C}_2\colon \{9P_2^2 - 1\}$$ □

Thus, we compute $\mathsf{maxrsize}_{\mathsf{mua}}$ by applying the Bernstein expansion to $\mathsf{rsize}^{\mathsf{cst}.m}_{\mathsf{mua}}$, constrained by a linear binding invariant $\mathcal{I}^{\mathsf{cst}.m}_{\mathsf{mua}}$, with parameters P_{mua}. That is:

$$\mathsf{maxrsize}^{\mathsf{cst}.m}_{\mathsf{mua}}(P_{\mathsf{mua}}) = \begin{cases} \max\{q(P_{\mathsf{mua}}) \in \mathcal{C}_1\} & \text{if } D_1(P_{\mathsf{mua}}) \\ \ldots & \\ \max\{q(P_{\mathsf{mua}}) \in \mathcal{C}_l\} & \text{if } D_l(P_{\mathsf{mua}}) \end{cases}$$

where $\{\mathcal{C}_i, D_i\}_{i=1..l} = \mathsf{Ber}(\mathsf{rsize}_m, \mathcal{I}^{\mathsf{cst}.m}_{\mathsf{mua}})$.

Example Table 2 shows the results for the example in Fig. 1. □

Table 2. Computing the function maxrsize using Bernstein basis

$\mathsf{Ber}(\mathsf{rsize}_{m3}, \mathcal{I}^{m0}_{m0.1.m1.5.m3}) = \{(h\_mc \geq 1; h\_mc + 1), (h\_mc < 1; 0)\}$
$\rightarrow \mathsf{maxrsize}^{m0.1.m1.5.m3}_{m0}(h\_mc) = h\_mc$ if ≥ 1, 0 otherwise
$\mathsf{Ber}(\mathsf{rsize}_{m3}, \mathcal{I}^{m0}_{m0.2.m2.6.m3}) = \{(true, 3h\_mc + 1)\}$
$\rightarrow \mathsf{maxrsize}^{m0.2.m2.6.m3}_{m0}(h\_mc) = 3h\_mc + 1$
$\mathsf{Ber}(\mathsf{rsize}_{m1}, \mathcal{I}^{m0}_{m0.1.m1}) = \{(true; h\_mc^2 + 2h\_mc)\}$
$\rightarrow \mathsf{maxrsize}^{m0.1.m1}_{m0}(h\_mc) = h\_mc^2 + 2h\_mc$
$\mathsf{Ber}(\mathsf{rsize}_{m2}, \mathcal{I}^{m0}_{m0.2.m2}) = \{(true; 6h\_mc)\}$
$\rightarrow \mathsf{maxrsize}^{m0.2.m2}_{m0}(h\_mc) = 6h\_mc$

5.2 Evaluating mem↑

Recall that $\mathsf{mem}\uparrow_{\mathsf{mua}}$ (Eq. 7) is basically a comparison between values to choose the largest one. Here we present an alternative definition of mem↑ where instead of comparing all potential call chains we recursively generate an evaluation tree that gets the same results: $\mathsf{mem}\uparrow_{\mathsf{mua}} = \mathsf{mem}\uparrow^{\mathsf{mua}}_{\mathsf{mua}}$ where

$$\mathsf{mem}\uparrow^{\mathsf{cst}.m}_{\mathsf{mua}} = \mathsf{maxrsize}^{\mathsf{cst}.m}_{\mathsf{mua}} + \max_{(m,l,m_i)\in\mathcal{E}_{\mathsf{mua}}} \mathsf{mem}\uparrow^{\mathsf{cst}.m.l.m_i}_{\mathsf{mua}} \quad (8)$$

where $\mathcal{E}_{\mathsf{mua}}$ is the set of edges of the call graph rooted at mua. Note that, if there is no cycle in CG with reachable object allocations then Eq. 8 yields a finite evaluation tree (see next section to see how to deal with some recursion patterns). Evaluation trees are intermediate representations of code that need to be executed at runtime on actual mua parameters to get the actual bound. Evaluation trees will enable symbolic compile-time manipulation in order to reduce the runtime effort for evaluating peak-consumption expressions.

An evaluation tree for our example is presented in Fig. 8. The tree has a direct relation with the application call graph: max nodes

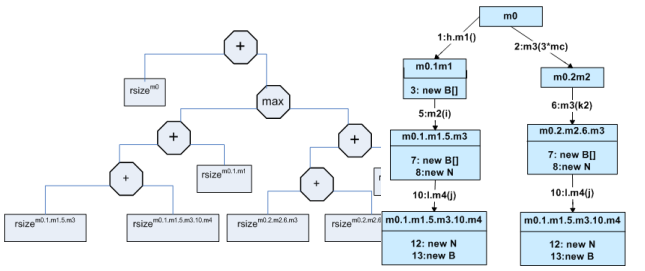

Figure 8. Evaluation tree of $\mathsf{mem}\uparrow_{m0}^{m0}$ and its correlation with the application (unfolded) call graph

are associated with branches in the call graph (i.e. independent regions); sum nodes are related with adjacencies in the call graph (i.e. regions that can live at the same time); leaves are associated with each node of the unfolded call graph (i.e. potential memory regions) by using maxrsize as the operation that yields the largest region size. To model the output of Ber for maxrsize evaluation, trees feature a $case$ node (not shown in Fig. 8) which provides a more general construction and models a set of pairs *(condition, evaluation-tree)*. Evaluation trees can be easily evaluated at runtime. Nevertheless, reductions can be done at compile-time, for instance, by applying powerful symbolic techniques for polynomial manipulation or by assuming some loss of precision of the upper-bounds. Details can be found in [17].

Finally, evaluation trees are translated into Java code to be executed at runtime. Assuming the number of domains and candidates is fixed, in the worst case, the number of evaluations is bounded by the number of branches of the call graph (i.e., the number of edges of a compacted version of the call graph). In practice the size of the evaluation tree is much smaller than the call graph, since only methods with non-null regions are considered and many of the comparisons can be solved off-line.

6. Validation

One possible context for peak consumption analysis is a tool producing time- and memory-predictable Java code out of conventional Java code [17]. In this setting our technique provides specifications of memory requirements necessary to guarantee proper execution. Thus, the technique presented in this paper has been integrated into the tool developed in [7]. That tool is able to automatically analyze sequential Java programs and to synthesize memory regions together with region sizes. Currently [7] does not handle automatically memory-consuming recursion requiring some manual intervention for those cases.

Details about the prototype tool can be found in [17]. One of the most important components is the one that solves the non-linear symbolic maximization problem. We have implemented the Bernstein transformation over multivaluated polyhedra following the approach presented by Clauss in [13]. We use `Daikon` [16] to infer local invariants[6]. We instrument the application in order to guide `Daikon` in the generation of invariants. An important role of the tool is the identification of the variables and path expressions that have to appear in the invariant (length of arrays and strings, size of collections, instance and static fields, etc. [17]). The tool also tries to obtain invariants for common iterations patterns such as iterators. Once we obtain invariants using `Daikon` or by other means (e.g. manually or by static analysis) we generate binding invariants by joining local invariants with the caller–callee binding information about call chains which are generated using the call graph.

6.1 Experimentation

The initial experimentation were carried out using the JOlden [8] benchmarks. As far as we know only Chin et al. [11] used well known benchmarks (JOlden) for their memory usage verification technique and most related techniques have been tested just using ad-hoc examples. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications which were the target application classes we had in mind when we devised the technique.

We automatically transform each application into a region-based one and use [7] to compute region sizes. Then, our new technique comes into play. Table 3 shows the computed peak expressions, the total allocation without a GC (as computed using [7]) and the number of non-empty regions. We also include the actual peak consumption (expressed in #objects) using the region based version of the application compared with the estimations obtained by evaluating the polynomials and the relative error ((#Objs - Estimation)/Estimation). TR is the time in seconds (Pentium 4, 3Ghz, 2GB RAM) corresponding to the region inference and transformation phase:(1) Escape analysis, (2) Region inference, (3) program instrumentation, (4) binary code generation using GCJ. TM is the time for computing region sizes:(1) find creation sites, and compute (2) control-state invariants (it also includes call graph computation), (3) inductive variables, and (4) Ehrhart polynomials [7]. Finally TB is the time to compute the peak memory requirements. In all cases we measure the peak consumption of the `main` method[7].

In these benchmarks the accuracy of the results is more sensitive to the precision of region inference and region sizes than to the modeling of the peak computation itself. This is because only few regions are active at the same time and most of the objects are allocated in the `main` method's region or in methods called only once. This is the reason why the cost of TB was negligible: few regions and simple binding invariants. It also explains why computed peak expressions and expressions obtained without considering GC are similar in `MST`, `Em3d`, `Perimeter` and `TreeAdd`.

`BiSort` is an interesting example. It has a recursive method `inOrder` which prints a tree. We slightly adapt it by creating the method `print` to capture objects created at `inOrder` in a separated region. The region `print` is allocated and deallocated at each iteration making the memory required to run `inOrder` a fixed value. In this way it follows a typical pattern where recursion is used to traverse structures and claimed memory is used and released at each invocation, avoiding accumulating unnecessarily data in the heap. In those cases, no matter how many times the call graph cycle is unrolled in the call stack, there is at most one non-null region and region-size parameters can be bounded using a binding invariant that holds no matter the number of unrolls (in this example the region size is fixed). For such a case an extension of the technique iw used. Basically, it by-passes strongly-connected components of the CG made of methods with null regions and calculates a widened binding invariant for reachable allocating methods. Another alternative to deal with recursive methods consist in replacing the associated CG subgraph with a peak memory-requirement specification for it. For instance, we can specify the peak consumption for the subgraph formed by `inOrder` and `print` as 1, obtaining the same results. Moreover, if we had not created `print`, the peak specification for `inOrder` would have been $(s-1)$ leading to a peak consumption of $2s+3$ (i.e. $peak_{main}(s) = 5+(s-1)+(s-1)$). This expression has to do with a less efficient

[6] Since `Daikon` produces "likely" invariants, we check them to ensure they are correct.

[7] The method `parseCmdLine` parses the command line arguments and stores the actual value of the parameters in static variables of the main class

Table 3. Experimental results

App	$\mathsf{mem}\uparrow_{main} + \mathsf{mem}\downarrow_{main}$	No GC	#Regs	Param.	#Objs	Estimation	Err%	Time (secs)		
								TR	TM	TB
`MST` -v nv	$\frac{9}{4}nv^2+3nv+5+\max\{nv-1,2\}$	$\frac{9}{4}nv^2+4nv+6$	3	10 100 1000	253 22703 2252003	269 22904 2254004	7% 1% 0%	16.03	26.04	0.03
`Em3d` -n n -d d	$6n.d+2n+14+\max\{6,2n\}$	$6n.d+4n+20$	3	(10,5) (100,7) (1000,8)	344 4604 52004	354 4614 52014	3% 0% 0%	17.34	30.37	0.05
`BiSort` -s s -p-m	$s+4$	$2s+5$	4	10 64 128	12 68 132	14 70 134	17% 3% 3%	17.55	3.21	0.03
`TSP` -c c -p-m	$2x+2$(where $x=2^{(\lfloor\log_2 c\rfloor)+1}$)	$4x+2$	5	10 31 63	31 63 127	34 66 130	10% 5% 2%	14.37	4.17	0.08
`Power`-p-m	32424	1552434	3	-	32421	32424	0%	20.72	5.82	0.02
`Health` -l -t	$\frac{1}{9}(21+51x+5(x-1)t)$ (where $x=4^l$)	$\frac{2}{3}(8(x-1)+(5x-2)t)$	6	(4,1) (5,3) (6,4)	1595 7510 34588	1538 7080 32791	4% 6% 5%	27.55	-	0.10
`TreeAdd` -l l -p-m	$x+6$ (where $x=2^l$)	$x+8$	2	8 10 12	262 1030 4102	259 1027 4099	1% 0% 0%	15.32	-	0.00
`BH` -b nb -s s	$13nb^2+246nb+37$	$25s.nb^2+nb(17+74s)+11s+37$	14	10 50 100	2385 11657 156637	3797 44837 23315	59% 285% 563%	25.49	-	0.08
`Perimeter` -l l -p-m	$x+11$ (where $x=4^{(l-4)}$)	$x+11$	2	13 14 17	158042 224090 6305002	262155 1048587 67108875	66% 367% 964%	18.78	-	0.00
`Voronoi`	∞	∞	5					27.76	-	-

use of region-based memory based on method granularity and not with an imprecision of the model or the specification.

`TSP` has also a recursive method `buildTree` that builds a tree which escapes to `main` region. The recursion follows the pattern already explained. `Power` shows a significative difference between the peak consumption and the total allocation. This is because it contains a method, which is invoked several times, where all the objects it creates are allocated in its associated region (nothing escapes). `Voronoi`, `Health`, `TreeAdd`, and `Perimeter` contain not only recursive method calls but also contain regions whose sizes grow exponentially w.r.t. some program parameters. For `Voronoi` the technique is not able yet to tackle some of its featured recursion patterns and it yields infinity as a safe over-approximation. Nevertheless, we were able to treat `Health`, `TreeAdd`, `BH` and `Permiter` by annotating the program with region sizes (this is why we did not complete the TM column for them). For `BH` we perform a minor refactoring to enable a better behavior for the region-based GC.

7. Related Work

The problem of dynamic memory estimation has been studied for functional languages in [20, 22, 29]. The work in [20] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs with an explicit deallocation mechanism. The computed expressions are linear constraints on the sizes of various parts of data. Our technique is meant to work for Java-like programs, is better suited for a region-based memory manager and computes non-linear parametric expressions (polynomials). [22] proposed a variant of ML extended with region constructs [28] together with a type system based on the notion of sized types [23] (linear constrains), such that well typed programs are proven to execute within the given memory bounds given as linear constrains. Although, their work is meant for first-order functional languages, they also rely on regions to control object deallocation. Unlike us they neither synthesize memory consumption nor handle non-linear cases. The technique proposed in [29] consists in, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. Unlike our polynomials, the evaluation of that function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed in [19, 10, 11, 2, 3, 21, 12]. The technique of [19] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper-bound. Since the unknowns may not be program inputs, it is not clear how instances are produced. Also, it seems to be quite overly pessimistic for programs with dynamically created arrays whose size depends on loop variables and it does not consider any memory collection mechanism. [21] presents a type system for heap analysis with explicit deallocations. Their analysis is based on an amortized complexity analysis where a potential is assigned to each datum according to its size and layout. Heap space usage can then be calculated during the type inference based on the annotated potential for each input. They do not present an inference method for heap consumption. The method proposed in [10, 11] relies on a type system and type annotations, similar to [22]. It statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear. Their type system allows aliasing and object deallocation (dispose) annotations. Our technique does not allow such annotations and indeed our memory model is more restricted. But as a counterpart we can synthesize non-linear bounds. In this same conference [12], they propose an extension of this technique which infers upper bounds of heap and stack usage provided they can be expressed as Presburger's formulas. The analysis is path sensitive in the sense that memory consumption expressions may have guards that can capture a more precise symbolic condition for each memory use/bound. For recursive methods or loops they build an abstract definition in constraint form that may be recursive and then solve it using a fix point analysis. In our case we try to avoid a fix point computation by using loop invariants for loops or user provided annotations. Recently Albert et al. [2] propose a technique for parametric cost analysis for sequential Java code. The code is translated to a recursive representation. Then, they infer *size relations* which are similar to our linear in-

variants. Using the size relation, and the recursive program representation they compute *cost relations* which are set of recurrent equation in term of input parameters. Applied to memory consumption their bounds are not limited to polynomials. However, solving recurrence equations is not a trivial task and is not always possible to obtain closed form solutions for a set of recurrence equations. Object deallocation is not considered. In [3] they propose an extension where object deallocation is handled, similarly to ours, by approximating objects lifetime using escape analysis and deallocating at the end of method execution. The paper presents some initial experimentation where object deallocation is not considered.

8. Conclusions and Future work

We presented a novel technique to compute non-linear parametric upper-bounds of the amount of dynamic memory *required* by a method. It was meant for region-based dynamic memory management, when regions are directly associated with methods, but it can be safely used to predict memory requirements for memory management mechanisms that free memory by demand.

The technique is fed by an application call graph, enriched with binding invariant information to constraint calling contexts and parametric specifications of regions sizes. These inputs can be obtained by either the tool developed in [7], using another tool, or by processing user-provided annotations. The output is a parametric specification of the memory required to run a method (or program). The output is given in the form of evaluation trees containing polynomials that can be easily translated to code evaluable in runtime. The size of the evaluation trees are known at compile time and can be reduced either using symbolic mathematical tools or by compromising some accuracy of run-time calculations.

The accuracy of the technique relies on several factors: the precision of its inputs (region sizes and invariants), the program structure that may allow or disallow two active regions get its maximum size at the same time, the precision of the Bernstein approximation and reductions applied to the evaluation tree. More benchmarks would be needed to assess its precision, but we think it is a promising approach.

The technique does not deal in the general case with memory-consuming recursions and, although we believe that this restriction is not an impediment for embedded systems, we are working on covering some other common cases. For more general patterns of recursion one could resort to techniques similar to regular model checking [1] or introducing a fresh variable to model the number of recursive invocations (that variable should be bounded by an expression over mua parameters). In any case, the presented technique could be slightly adapted to leverage on user-provided peak memory-requirements. Thus, a specification could replace recursive methods. Moreover, we also believe that it is possible to devise a technique integrating our analysis together with those mentioned type-checking based ones which are better suited for more complex or recursive data structures [12]. The approach would use type checked annotations for data container classes (like the ones provided by standard libraries) and our inference approach for loop intensive applications on top of those verified libraries.

Acknowledgments

Thanks to the anonymous referees for their valuable comments. This work has been partially funded by ECOS A06E02 “Japiqay”, Rhône-Alpes MADEJA, ANPCyT grants PICT 11738 and PICT 32440, UBACyTX020 and MSR fellowship grant. Sergio Yovine is currently a visiting Professor at DC, FCEyN, UBA.

References

[1] P. Abdulla, B. Jonsson, M. Nilsson, M. Saksena. A survey of regular model checking. *CONCUR'04*, LNCS 3170, 2004.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini. Cost analysis of java bytecode. *ESOP'07*, LNCS 4421, 2007.

[3] E. Albert, S. Genaim, M. Gómez-Zamalloa. Heap space analysis for java bytecode. *ISMM'07*, ACM 2007.

[4] D. Bacon, P. Cheng, D. Grove. Garbage collection for embedded systems. *EMSOFT'04*, ACM 2004.

[5] G. Barthe, M. Pavlova, G. Schneider. Precise analysis of memory consumption using program logics. *SEFM'05*. IEEE 2005.

[6] G. Bollella, J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

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

[8] B. Cahoon, K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. *PACT'01*, IEEE 2001.

[9] S. Cherem, R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, ACM 2004.

[10] W. Chin, S. Khoo, S. Qin, C. Popeea, H. Nguyen. Verifying safety policies with size properties and alias controls. *ICSE'05*, ACM 2005.

[11] W. Chin, H. H. Nguyen, S. Qin, M. Rinard. Memory usage verification for oo programs. *SAS'05*, LNCS 3672, 2005.

[12] W.N. Chin, H.H. Nguyen, C. Popeea, S. Qin Analysing memory resource bounds for low-level programs. In *ISMM 08*, 2008.

[13] Ph. Clauss, I. Tchoupaeva. A symbolic approach to Bernstein expansion for program analysis and optimization. *CC'04*, 2004.

[14] Ph. Clauss, F. Fernández, D. Garbervetsky, S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. TR06-04, Univ. L. Pasteur, 2006.

[15] D. Detlefs. A hard look at hard real-time garbage collection. *ISORC'04*, IEEE 2004.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69(1-3): 35-45, 2007.

[17] D. Garbervetsky. Parametric specification of dymamic memory utilization. *Ph.D Thesis*, DC, FCEyN, UBA, November 2007.

[18] D. Gay, A. Aiken.Language support for regions.*PLDI'01*,ACM 2001.

[19] O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng, MIT, 2002.

[20] M. Hofman, S. Jost. Static prediction of heap usage for first-order functional programs. *POPL'03*, ACM 2003.

[21] M. Hofman, S. Jost. Type-based amortised heap-space analysis. In *ESOP '06*. 2006.

[22] J. Hughes, L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. *ICFP'99*, SIGPLAN Notices 34(9), ACM 1999.

[23] J. Hughes, L. Pareto, A. Sabry. Proving the correctness of reactive systems using sized types. *POPL'96*, ACM 1996.

[24] V. Sundaresan, P. Lam, E. Gagnon, R. Vallée-Rai, L. Hendren, P. Co. Soot: A java optimization framework. *CASCON'99*, IBM 1999.

[25] G. Salagnac, Ch. Rippert, S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. *RTCSA'07*, IEEE 2007.

[26] G. Salagnac, S. Yovine, D. Garbervetsky. Fast escape analysis for region-based memory management. *ENTCS*, 131:99–110, 2005.

[27] A. Salcianu, M. Rinard. Pointer and escape analysis for multithreaded programs. *PPoPP'01*, SIGPLAN Notices 36(7), ACM 2001.

[28] M. Tofte, J.P. Talpin. Region-based memory management. *Inf. Comput.* 132(2):109–176, 1997.

[29] L. Unnikrishnan, S.D. Stoller, Y.A. Liu. Optimized live heap bound analysis. *VMCAI'03*, LNCS 2575, 2003.