Verimag

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Symbolic Prediction of Dynamic-Memory Requirements

V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine

Verimag Research Report n^o TR-2007-11

November 2, 2007

Reports are downloadable at the following address
http://www-verimag.imag.fr

UNIVERSITE JOSEPH FOURIER

CNRS

INPG

Symbolic Prediction of Dynamic-Memory Requirements

V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine

November 2, 2007

Abstract

This work presents a technique to compute symbolic non-linear approximations of the amount of dynamic memory *required* to safely run a method in (Java-like) imperative programs. We do that for scoped-memory management where objects are organized in regions associated with the lifetime of methods. Our approach resorts to a symbolic non-linear optimization problem which is solved using Bernstein basis.

Keywords: Dynamic memory, Scoped-memory management, Memory requirements.

Reviewers:

Notes: V. Braberman, F. Fernández, and D. Garbervetsky are affiliated with Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina. E-mail: {vbraber,diegog}@dc.uba.ar, federico.fernandez@gmail.com. S. Yovine is affiliated with Verimag. E-mail: Sergio.Yovine@imag.fr. This work has been partially supported by ECOS project A06E02, ANCyT grant PICT 32440, UBACyTX020 and IBM Innovations Grants.

How to cite this report:

@techreport { ,
title = { Symbolic Prediction of Dynamic-Memory Requirements},
authors = { V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine},
institution = { Verimag Research Report },
number = {TR-2007-11},
year = { },
note = { }
}

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 unbounded pause times. Several solutions have been proposed, either by building garbage collectors with real-time performance, e.g. [2] (see [14] for a survey), or by using a scope-based programming paradigm, e.g. [16, 5, 15, 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 [17].

In a previous work we presented a technique for computing a parametric upper-bound of the amount of memory dynamically *requested* by Java-like imperative programs [7]. The idea consists in quantifying dynamic allocations done by a method. Given a method m with parameters $p_1, \ldots, p_k$ we exhibit an algorithm that computes a parametric non-linear expression over $p_1, \ldots, p_k$ 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 garbage collector is *not* taken into account. We also showed that assuming a region-based memory management [16, 5, 15, 9] where objects are organized in regions associated with computation units, the same technique allows to obtain non-linear parametric bounds of the size of every memory region.

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 $p_1, \ldots, p_k$ we obtain a polynomial upper-bound of the amount of memory necessary to *safely* execute the method and all methods it calls, without running out of memory. This polynomial can be seen as a *pre-condition* stating that the method requires that much free memory to be available before executing, and also as a *certificate* engaging the method is not going to use more memory than the specified. To compute this estimation we consider memory deallocation that may occur during the execution of the method. Basically, assuming a region-based memory management we model all the potential configurations of regions stacks at run-time. Since region sizes are expressed as polynomials, this model leads to a symbolic non-linear optimization problem. This problem can be solved using a technique using Bernstein basis [12].

To our knowledge, the technique used to infer non-linear dynamic memory requirements under a region-based memory manager and its effective computation using Bernstein basis is a novel approach to memory requirements calculus.

Applications of this set of techniques are manifold, from improvements in memory management to the generation of parametric memory-allocation certificates. These specifications would enable application loaders and schedulers (e.g., [21]) to make decisions based on available memory resources and the memory-consumption estimates.

Outline

In section 2 we present a definition of the problem we want to solve and some assumptions that we are making. In section 3 propose an effective definition of a function that predict memory requirements for a scoped-based memory management. In section 4 propose an approach to compute the memory requirements function. In section 3 we show some experiments. In section 6 we discuss some aspects of the technique that we would like to improve. In section 7 we discuss some related work. In section 8 we present our conclusions and future work.

2 Problem statement

Let Θ be a *non-recursive, sequential* Java-like program, given as a set $\mathcal{M}$ of methods. W.l.o.g., we assume there is a distinguished method under analysis $\mathsf{mua} \in \mathcal{M}$, where the program starts. Each $m \in \mathcal{M}$ is characterized by its formal parameters P_m, its local variables V_m, and its body (list of statements) $\mathsf{stm}_m \in \mathcal{S}^*$, where $\mathcal{S}$ is the set of statements (`new`, `call`, `ret`, ...). For the sake of simplicity, we assume that parameters are of integer type.

A program state $\sigma \in \Sigma$ consists of a *control stack* $\mathsf{cst}(\sigma) \in (\mathcal{M} \times \mathbb{N})^*$, the *data stack* $\mathsf{val}(\sigma) \in [(V \cup P \mapsto \mathcal{V})^*]$ of valuations of variables and parameters (with $V = \bigcup_{m \in \mathcal{M}} V_m$ and $P = \bigcup_{m \in \mathcal{M}} P_m$),

and the *heap* $\mathsf{heap}(\sigma) \subseteq \mathcal{O}^2$, $\mathcal{O} \subseteq \mathcal{V}$, modelled as a directed graph of *objects*. We write $\mathsf{m}_k(\sigma)$, $\mathsf{pc}_k(\sigma)$, and $\mathsf{val}_k(\sigma)$ to denote the k-th method, control location and data valuations, respectively.

An object $o \in \mathcal{O}$ is said to be *live* in $\mathsf{heap}(\sigma)$ iff it is *reachable* from a variable defined in the state, that is, there exists k such that $\mathsf{val}_k(\sigma)(v)$ has a path to o in the heap. The set of live objects of σ is denoted $\mathsf{live}(\sigma)$. We define $\mathsf{dead}(\sigma)$ to be $\mathsf{heap}(\sigma) \setminus \mathsf{live}(\sigma)$. Let $\mathtt{size}(o) \geq 1$ be the amount of memory occupied by o. $\mathsf{memUsed}(\sigma) \triangleq \sum_{o \in \mathsf{heap}(\sigma)} \mathtt{size}(o)$, is the amount of memory *occupied* in σ.

The semantics is given by a deterministic transition system $\langle \Sigma, \Sigma^{\mathsf{mua}}, \rightarrow \rangle$, where $\Sigma^{\mathsf{mua}} \subseteq \Sigma$, is the set of initial states, such that for all $\sigma \in \Sigma^{\mathsf{mua}}$, $\mathsf{cst}(\sigma) = (\mathsf{mua}, 0)$ and $\mathsf{heap}(\sigma) = \emptyset$. The transition relation $\rightarrow$ is defined by the operational semantics *without* deleting dead objects from the heap. For any $\sigma \in \Sigma$, we write $\mathsf{succ}(\sigma)$ to denote the state σ' such that $\sigma \rightarrow \sigma'$. This relation is assumed to be such that for any two states σ and $\bar{\sigma}$ that are equal except for the set of dead objects, $\mathsf{succ}(\sigma)$ is also equal to $\mathsf{succ}(\bar{\sigma})$, except for the set of dead objects. Moreover, $\mathsf{dead}(\sigma) \subseteq \mathsf{dead}(\mathsf{succ}(\sigma))$, that is, it is not possible to *forge* pointers.

A (finite) *run* is a sequence $\rho = \sigma_0, \sigma_1, \dots \in \Sigma^*$, 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. A run is *uniquely* determined by the values assigned to the parameters of mua. We write ρ^{ϑ} to denote the run determined by the parameter valuation ϑ.

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 $i \geq 0$, $\mathsf{dead}(\Gamma(\rho_i)) \subseteq \mathsf{dead}(\rho_i)$, while keeping the rest of the state unchanged. We call $\mathtt{ideal}$ manager the one such that $\mathsf{dead}(\mathtt{ideal}(\rho_i)) = \emptyset$. From now on, we will indistinctly denote $\Gamma(\rho_i)$ as $\Gamma_i(\rho)$.

2.1 Peak consumption

The *peak* amount of memory consumed by a run ρ^{ϑ}, is defined as follows:

$$\mathsf{peak}_{\Gamma}(\vartheta) \quad \triangleq \quad \max_i \mathsf{memUsed}(\Gamma_i(\rho^{\vartheta})) \tag{2.1}$$

Clearly, we have that for all ϑ, $\mathsf{peak}_{\mathtt{ideal}}(\vartheta) \leq \mathsf{peak}_{\Gamma}(\vartheta)$, for all Γ.

Determining the peak consumption at compile time would enable to know "a priori" the amount of memory required to *safely* execute the program (from mua) without running out of memory. However, predicting this peak is hard for different reasons, as the following example tries to illustrate.

```
void m0(int mc) {
1:  m1(mc);
2:  m2(3 * mc);
}
void  m1(int k) {
3:   B[][] dummyArr = new B[k][];
4:   for (int i = 1; i <= k; 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] = m4(l,j);
   }
11: return arrB;
}

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

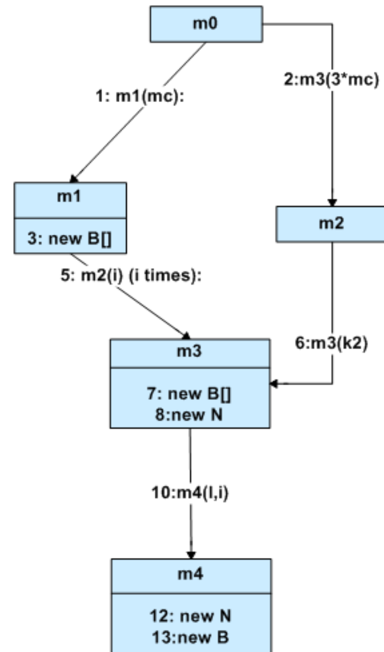

Figure 1: A sample program with its detailed call graph

Example. Consider the program in Fig. 1. The objects allocated in method $m4$ at locations 12 and 13 (denoted as $m4.12$ and $m4.13$) cannot be collected when $m4$ finishes its execution because they are referenced from outside. $m4.12$ can be collected when $m3$ finishes its execution. $m4.14$ can be collected just at the end of $m2$ or $m1$ when the last reference is removed. And so on. Fig. 2 depicts the effects of allocations and deallocations on memory occupancy for different executions ($m0$ invoked with $mc = 3$ and $mc = 7$, respectively) using the `ideal` memory manager. The figure only shows memory occupation generated by explicit requests at allocation statements, that is, there is no allocation overhead introduced by the memory manager. In the first run the peak is reached after $m0$ calls $m2$, and in the second one, the peak occurs after $m0$ calls $m1$. □

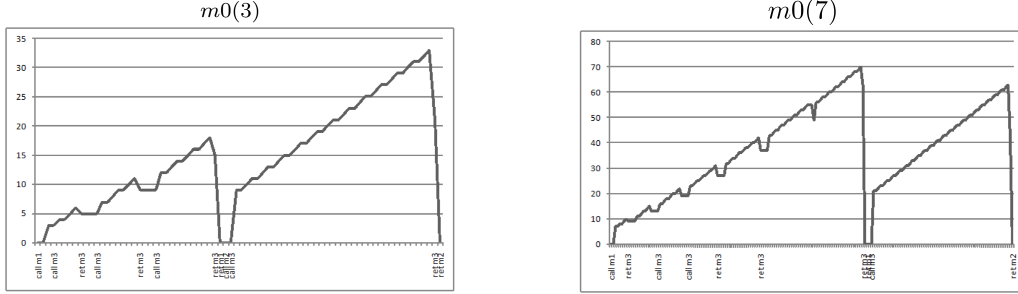

Figure 2: Two traces: $m0(3)$ (above) and $m0(7)$ (below).

This example pinpoints the issues that need to be considered for estimating the peak: it is necessary (1) to estimate the amount of memory *requested* by a method, and (2) to determine *when* objects will be *collected*, both in terms of *the parameters of* mua, as different peak consumptions could occur at *different* places (control locations, method instances, ...) depending on initial parameters.

2.2 Peak consumption for scoped-memory management

The `ideal` memory manager is optimal in terms of memory consumption. This collector is used in works that *verify* memory usage certificates such as [11, 3]. However, it is not well understood how to *predict* memory consumption for it.

In this work we follow a different strategy: we assume the presence of a *scoped-memory* manager that reclaims memory only *at the end* of the execution of every method. The collector is only allowed to claim for dead objects created *during* the execution of the method (and the method it transitively calls). Objects created in an outer scope cannot be collected by the current method and may be reclaimed by some of the methods in the call stack. In particular, we will choose a scoped-based memory management where objects are organized in regions and each method has an associated region (denoted as an m-region) whose lifetime corresponds with its associated method's lifetime [15]. To be safe, objects in a region can point to objects in the same region or a parent region (corresponding to a method that is in the call stack). This scoping restriction can be satisfied inferring regions at compile-time by performing escape analysis [15, 23, 22].

Let $r \subseteq \mathcal{O}$ be a region. We assume there is one (possibly empty) region per method occurrence in the call stack: for any $\sigma \in \Sigma$, the *region stack* is $\mathsf{rst}(\sigma) = r_1 \ldots r_t$ where $\mathsf{m}(\sigma) = m_1 \ldots m_t$. The set of nodes of the heap is a *disjoint* union of regions. A region-based memory manager Γ only eliminates r_t. For all ρ, $\Gamma(\rho)$ is as follows:

- $\Gamma_0(\rho) = \rho_0$,
- and for all $i \geq 0$, $\Gamma_{i+1}(\rho)$ is:
  - if $\mathsf{stm}(\mathsf{pc}_t(\rho_i)) = \mathtt{ret}$, then
    - $\mathsf{heap}(\Gamma_{i+1}(\rho)) = \mathsf{heap}(\Gamma_i(\rho)) \setminus r_t$, and
    - $\mathsf{rst}(\Gamma_{i+1}(\rho)) = \mathsf{rst}(\Gamma_i(\rho)) \setminus r_t$,
  - otherwise $\Gamma_{i+1}(\rho) = \mathsf{succ}(\Gamma_i(\rho))$ and $\mathsf{rst}(\Gamma_{i+1}(\rho))$ is:

* if the statement is a `call`, a new (empty) region is attached to the method pushed on top of the stack,
* if an object o is created (`new`), it is equal to $\mathsf{rst}(\Gamma_i(\rho))$, except for the region to which o is assigned to,
* otherwise it remains unchanged.

Furthermore, to ensure that *only* dead objects are removed, the assignment of new objects into regions is supposed to be 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 from the fact that inter-region references comply with scoping rules.

The memory peak of a run collected by a region-based memory manager Γ is a function of the size of the regions. Let $\texttt{size}(r) = \sum_{o \in r} \texttt{size}(o)$. Then:

$$\mathsf{peak}_\Gamma(\vartheta) = \max_i \sum_{1 \le k \le |\mathsf{cst}(\rho_i^\vartheta)|} \texttt{size}(\mathsf{rst}_k(\Gamma_i(\rho^\vartheta))) \tag{2.2}$$

The rest of this paper is devoted to develop a computational technique to approximate the right-hand side of Eq. 2.2.

3 Computing peak consumption for scoped-memory

Let $\Pi_{\mathsf{mua}} \subseteq (\mathcal{M} \times \mathbb{N})^*$ be the set of call stacks whose first element is $(\mathsf{mua}, 0)$. For $\pi \in \Pi_{\mathsf{mua}}$, we write $\pi[1..k]$ to denote the prefix of length k, and π_k to denote the k-th method. Σ can be partioned according to call stacks: $\sigma, \sigma' \in \Sigma$ are in the same class iff $\mathsf{cst}(\sigma) = \mathsf{cst}(\sigma')$. Then, from Eq. 2.2 it follows that:

$$\mathsf{peak}_\Gamma(\vartheta) = \max_{\pi \in \Pi_{\mathsf{mua}}} \max_{i \text{ st.} \pi = \mathsf{cst}(\rho_i^\vartheta)} \sum_{1 \le k \le |\pi|} \texttt{size}(\mathsf{rst}_k(\Gamma_i(\rho^\vartheta))) \tag{3.1}$$

The right-hand side of Eq. 3.1 can be over-approximated by summing up the maximum region sizes along the run:

$$\mathsf{peak}_\Gamma(\vartheta) \le \max_{\pi \in \Pi_{\mathsf{mua}}} \sum_{1 \le k \le |\pi|} \max_{i \text{ st.} \pi = \mathsf{cst}(\rho_i^\vartheta)} \texttt{size}(\mathsf{rst}_k(\Gamma_i(\rho^\vartheta))) \tag{3.2}$$

We can further over-approximate the right-hand side of Eq. 3.2 by considering a partition for each stack-depth k, that is, $\sigma, \sigma' \in \Sigma$ are in the same class for depth k, iff $\mathsf{cst}(\sigma)[1..k] = \mathsf{cst}(\sigma')[1..k]$. Thus:

$$\mathsf{peak}_\Gamma(\vartheta) \le \max_{\pi \in \Pi_{\mathsf{mua}}} \sum_{1 \le k \le |\pi|} \max_{i \text{ st.} \pi[1..k] = \mathsf{cst}(\rho_i^\vartheta)[1..k]} \texttt{size}(\mathsf{rst}_k(\Gamma_i(\rho^\vartheta))) \tag{3.3}$$

Since Π_{mua} is finite and all $\pi \in \Pi_{\mathsf{mua}}$ are of finite length, it follows that computing an over-approximation of the peak reduces to computing maximum region sizes.

3.1 Approximating region sizes

Given a method $m \in \mathcal{M}$, let $\mathsf{rsize}_m(P_m)$ be a function of m's parameters that yields an *over-approximation* of the size of *any* region associated to m, that is, for every state σ that contains a region r_k associated with method m at position k in the stack, $\texttt{size}(r_k) \le \mathsf{rsize}_m(\mathsf{val}_k(\sigma)(P_m))$. Thus, it follows that:

$$\mathsf{peak}_\Gamma(\vartheta) \le \max_{\pi \in \Pi_{\mathsf{mua}}} \sum_{1 \le k \le |\pi|} \max_{i \text{ st.} \pi[1..k] = \mathsf{cst}(\rho_i^\vartheta)[1..k]} \mathsf{rsize}_{\pi_k}(\mathsf{val}_k(\rho_i^\vartheta)(P_{\pi_k})) \tag{3.4}$$

Eq. 3.4 does not depend on Γ because $\mathsf{val}_k(\rho_i^\vartheta)(P_{\pi_k}) = \mathsf{val}_k(\Gamma_i(\rho^\vartheta))(P_{\pi_k})$.

The rsize functions can be computed using the techniques presented in [7]. Here, we illustrate the approach with an example.

Example. For instance, rsize functions for our motivating example are:

$$\begin{aligned}
\mathsf{rsize}_{m0} &= 0\\
\mathsf{rsize}_{m1} &= \mathtt{size}(B[])k + (\mathtt{size}(B[]) + \mathtt{size}(B))(\frac{1}{2}k^2 + \frac{1}{2}k)\\
\mathsf{rsize}_{m2} &= (\mathtt{size}(B[]) + \mathtt{size}(B))k2\\
\mathsf{rsize}_{m3} &= \mathtt{size}(N) + \mathtt{size}(N)n\\
\mathsf{rsize}_{m4} &= 0
\end{aligned}$$

where $\mathtt{size}(B)$ and $\mathtt{size}(B[])$ are the sizes of objects of type B and $B[]$. □

3.2 Approximating maximum region sizes

Equation 3.4 still depends on the run determined by the valuation ϑ of P_{mua} to obtain the values of the parameters to evaluate rsize. These values can be approximated by an invariant binding P_{mua} with the parameters of *all* methods in given call chain π. This *binding invariant* constrains the possible valuations of variables stored in stack frames of methods invoked in π. Such an invariant, denoted $\mathcal{I}_{\pi}^{\mathsf{mua}}$, can be obtained, for instance, from local invariants as in [7].

Example. For instance a valid binding invariant $\mathcal{I}_{\mathsf{mua}}^{m0.1.m1.5.m3}$ in our example is $\{k = mc, 1 \leq i \leq k, n = i\}$. □
For $m \in \mathcal{M}$ and $\pi.m \in \Pi_{\mathsf{mua}}$, the maximum value of rsize_m in *all* states σ where $\pi.m$ is a prefix of $\mathsf{cst}(\sigma)$, can be over-approximated by its maximum value over a binding invariant $\mathcal{I}_{\mathsf{mua}}^{\pi.m}$. Now, let:

$$\mathsf{maxrsize}_{\mathsf{mua}}^{\pi.m}(P_{\mathsf{mua}}) \triangleq \textit{Maximize}\ \mathsf{rsize}_m(P_m)\ \textit{sbj.to}\ \mathcal{I}_{\mathsf{mua}}^{\pi.m}(P_{\mathsf{mua}}, P_m, W) \tag{3.5}$$

where W are local variables appearing in the methods in π. Clearly,

$$\max_{i\ \mathrm{st.}\pi[1..k]=\mathsf{cst}(\rho_i^{\vartheta})[1..k]} \mathsf{rsize}_{\pi_k}(\mathsf{val}_k(\rho_i^{\vartheta})(P_{\pi_k})) \leq \mathsf{maxrsize}_{\mathsf{mua}}^{\pi[1..k]}(\vartheta) \tag{3.6}$$

Example. $\mathsf{rsize}_{m3}(n) = \mathtt{size}(N) + \mathtt{size}(N)n$ and $\mathcal{I}_{\mathsf{mua}}^{m0.1.m1.5.m3} = \{k = mc, 1 \leq i \leq k, n = i\}$, imply $\mathsf{maxrsize}_{m0.1.m1.5.m3}^{m0}(mc) = \mathit{size}(N) + \mathit{size}(N)mc$. □
Let

$$\mathsf{memRq}(\vartheta) \triangleq \max_{\pi \in \Pi_{\mathsf{mua}}} \sum_{1 \leq k \leq |\pi|} \mathsf{maxrsize}_{\mathsf{mua}}^{\pi[1..k]}(\vartheta) \tag{3.7}$$

It follows that: For all ϑ, $\mathsf{peak}_{\Gamma}(\vartheta) \leq \mathsf{memRq}(\vartheta)$.

Example. Table 1 shows $\mathsf{maxrsize}_{m0}^{\pi.m}$ for the example of Fig. 1. Call chain for this example are prefixes of $m0.1.m1.5.m3.10.m4$ and $m0.2.m2.6.m3.10.m4$. Since we compute a sum over all regions for a given call chain, the sum over any prefix of these call chains will give a value which is lower or equal. Thus, it is enough to apply the maximum to the result of these chains. Using the resulting maxrsize expressions we can reduce memRq_{m0} to:

$$\begin{aligned}
&\mathsf{memRq}_{m0}(mc) = \max\{\\
&\quad (\mathtt{size}(B[]) + \mathtt{size}(B))(\frac{1}{2}mc^2 + \frac{1}{2}mc) + \mathtt{size}(B[])mc + \mathtt{size}(N)(1 + mc),\\
&\quad (\mathtt{size}(B[]) + \mathtt{size}(B))3mc + \mathtt{size}(N)(1 + 3mc)\}
\end{aligned}$$

Here, for simplicity, we will assume that $\mathtt{size}(T) = 1$ for all T. Then:

$$\begin{aligned}
\mathsf{memRq}_{m0}(mc) &= \max\{mc^2 + 2mc + 1 + mc, 6mc + 1 + 3mc\}\\
&= 1 + \max\{mc^2 + 3mc, 9mc\} = 1 + 3mc + \max\{mc^2, 6mc\}\\
&= 1 + 3mc + \begin{cases} mc^2 & \text{if } mc < 0 \vee mc > 6\\ 6mc & \text{if } 0 \leq mc \leq 6 \end{cases}
\end{aligned}$$

Table 1: Expression for function $\mathsf{maxrsize}$ for the example

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

Fig. 3 shows that memRq_{m0} is an upper-bound of the actual memory requirements of the example of Fig. 1. 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}. □

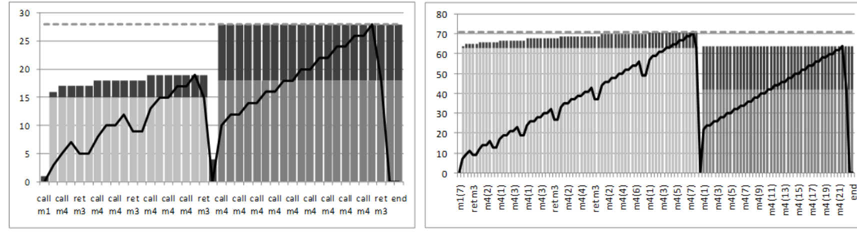

Figure 3: Consumption, rsize, and memRq_{m0} for $m0(3)$ and $m0(7)$.

4 Computing maxrsize and memRq

The formulation of $\mathsf{maxrsize}$ characterizes a non-linear maximization problem, where the polynomial rsize represents the input and the binding invariant for the control stack represents the restriction, whose solution 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 stated and solved symbolically. As a consequence, it is not possible to resort to non-linear numerical optimization.

4.1 Computing maxrsize

To compute $\mathsf{maxrsize}$, we resort to [12] which proposes an extension of Bernstein expansion [4] 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 Berstein basis give minimum and maximum bounds on the polynomial values. Here, we do not go into the details, which can be found in [13]. 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 $pol \in \mathbb{Q}[\vec{x}]$, 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 $\mathbf{p} \in \mathbb{Q}^n$:

$$\max\{pol(\mathbf{x}) \mid I(\mathbf{p}, \mathbf{x})\} \leq \begin{cases} \max\{q(\mathbf{p}) \in \mathcal{C}_1\} & \text{if } D_1(\mathbf{p}) \\ \ldots & \\ \max\{q(\mathbf{p}) \in \mathcal{C}_l\} & \text{if } D_l(\mathbf{p}) \end{cases} \quad (4.1)$$

We compute maxrsize by applying the Bernstein expansion to rsize, constrained by a linear binding invariant, with parameters P_{mua}. D_i. That is:

$$\mathsf{maxrsize}^{\pi.m}_{\mathsf{mua}} \quad = \quad \mathsf{Ber}(\mathsf{rsize}_m, \mathcal{I}^{\pi.m}_{\mathsf{mua}}) \tag{4.2}$$

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

Table 2: Computing the function maxrsize using Bernstein basis

$\mathsf{maxrsize}^{m0.1.m1.5.m3}_{m0} =$	$\mathsf{Ber}(\mathcal{I}^{m0}_{m0.1.m1.5.m3}, \mathsf{rsize}_{m3})$
Domain: $\{mc \geq 1\}$	Candidates: $\{mc + 1\}$
Domain: $\{mc < 1\}$	Candidates: $\{0\}$
$\mathsf{maxrsize}^{m0.2.m2.6.m3}_{m0} =$	$\mathsf{Ber}(\mathcal{I}^{m0}_{m0.2.m2.6.m3}, \mathsf{rsize}_{m3})$
Domain: true	Candidates: $\{3mc\}$

$\mathsf{maxrsize}^{m0.1.m1}_{m0} =$	$\mathsf{Ber}(\mathcal{I}^{m0}_{m0.1.m1}, \mathsf{rsize}_{m1})$
Domain: true	Candidates: $\{mc^2 + 2mc\}$
$\mathsf{maxrsize}^{m0.2.m2}_{m0} =$	$\mathsf{Ber}(\mathcal{I}^{m0}_{m0.2.m2}, \mathsf{rsize}_{m2})$
Domain: true	Candidates: $\{6mc\}$

4.2 Evaluating memRq

Recall that $\mathsf{memRq}(\vartheta)$ (Eq. 3.7) is basically a comparison between values to choose the largest one. Here we present an alternative definition of memRq where instead of comparing all potential call chains we recursively generate an evaluation tree that gets the same results: $\mathsf{memRq}_{\mathsf{mua}} = \mathsf{memRq}^{\mathsf{mua}}_{\mathsf{mua}}$ where

$$\mathsf{memRq}^{\pi.m}_{\mathsf{mua}} \quad = \quad \mathsf{maxrsize}^{\pi.m}_{\mathsf{mua}} + \max_{(m,l,m_i)\in\mathcal{E}_{\mathsf{mua}}} \mathsf{memRq}^{\pi.m.l.m_i}_{\mathsf{mua}} \tag{4.3}$$

where $\mathcal{E}_{\mathsf{mua}}$ is the set of edges of the call graph rooted at mua.

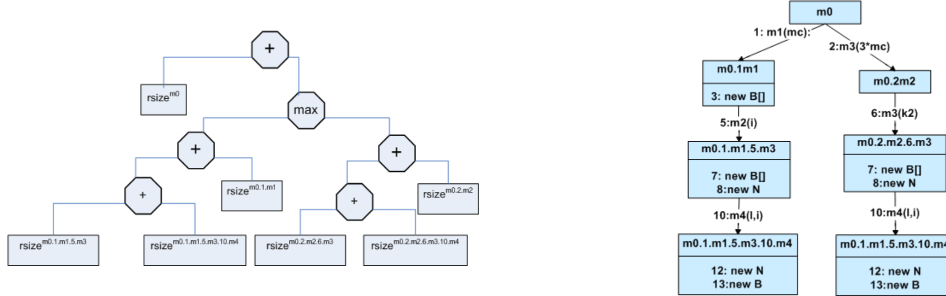

Figure 4: Evaluation tree of the computation of the amount of memory required to run $m0$ and its correlation with the application (unfolded) call graph

Evaluation trees enable symbolic compile-time manipulation in order to reduce the runtime effort of evaluation of the peak consumption prediction. An evaluation tree for our example is presented in Fig. 4. The tree has a direct relation with the application call graph: *max* nodes 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. 4) 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 [6].

Example Fig. 5 shows the evolution of the evaluation tree for the example Fig. 1. The first tree is the evaluation tree after applying Ber for solving the maximization problem for maxrsize. The next two trees are successive simplifications. To go from the first tree to the second one, we start by removing the `Case` node by taking directly the case $m+1$ by using the fact that the binding invariant forces $mc \geq 1$. Then, we sum the nodes in the left part of the `max` node getting the expression $1 + 3mc + mc^2$. Since $3mc$ appears

also in the right side ($6mc = 3mc + 3mc$) we can factor that node. Then, to move from the second tree to the third we convert the `max` node to a `Case` node after finding the interval where one polynomial mc^2 is above $6mc$ and vice versa. □

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 many of the comparisons can be solved off-line.

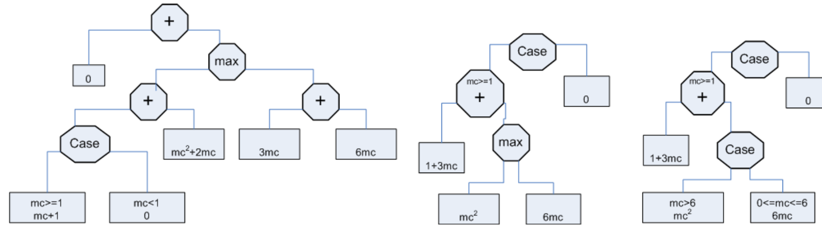

Figure 5: Evaluation tree after computing maxrsize and two successive reductions.

5 Preliminary experiments

The initial set of experiments were carried out on a subset of programs from JOlden [8] benchmarks. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications the target application classes we had in mind when we devised the technique. In order to make the result more readable, the tool computes the number of object instances created when running the selected method, rather than the actual memory allocated by the execution of the method. Table 3 shows the computed peak expressions, and the comparison between real executions and estimations obtained by evaluating the polynomials. The last column shows the relative error ((#Objs - Estimation)/Estimation).

Table 3: Experimental results

Example	memRq	Param.	#Objs	Estimation	Err%
`MST`(nv)	$1 + \frac{9}{4}nv^2 + 3nv + 5 + \max\{nv-1, 2\}$	10	253	270	6%
		20	943	985	4%
		100	22703	22905	1%
		1000	2252003	2254005	0%
`Em3d`(nN, nD)	$6nN.nD + 2nN + 14 + \max\{6, 2nN\}$	(10,5)	344	354	3%
		(20,6)	804	814	1%
		(100,7)	4604	4614	0%
		(1000,8)	52004	52014	0%
`BiSort(n)`	$6 + n$	10	13	16	19%
		20	21	26	19%
		200	69	135	45%
		64	69	70	1%
		128	133	134	1%
`Power()`	32656	-	32420	32656	1%

These experiments show that the technique produced quite accurate results, actually yielding almost exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of allocations associated with exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the bisort example, the reason of the over-approximation is that the actual number of instances is always bounded by $2^i - 1$, with $i = \lfloor log_2 n \rfloor$. Indeed, the estimation was exact for arguments power of 2.

6 Discussion

Peak consumption for an arbitrary method:

Our technique assumes that mua is the program's starting method. If we want to use our technique for an arbitrary method m we need to consider the fact that m (or some method it may transitively call) may allocate some object whose lifetime exceeds its m's lifetime. In this case, objects should be allocated in some region of a caller of m. To deal with this situation we introduce a new function denoted $\mathtt{memEscapes}_m$ 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, meaning that they have to be allocated in other callers' regions. `memEscapes` 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. As we do for rsize in [7] we propose technique to automatically infer `memEscapes`.

To cope with this extension, we only need to take into account the amount of memory escaping m 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 $m, \ldots, m'$. Thus, memRq is redefined as follows:

$$\mathsf{memRq}_{\mathsf{mua}} \quad \triangleq \quad \mathtt{memEscapes}_{\mathsf{mua}} + \mathsf{memRq}_{\mathsf{mua}}^{\mathsf{mua}} \tag{6.1}$$

About the parameterization of memRq:

We defined peak in terms of P_{mua} and we assumed P_{mua} are of integer type. Nevertheless, a method may be invoke with parameter of more complex data types and consumption may be more directly related with other expressions derivable from the parameters. For instance, suppose that we want to know the amount of memory required to run a method `clone(`c`: Collection)` that returns a fresh copy of a collection c. Clearly, the size of c is relevant for computing the memory requirements. To cope with this, we can use a new variable $size$ for the peak calculation and relate it with c using a predicate $size = c.size()$.

For this, we devise an alternative definition of peak that allows introducing new variables and a uses a predicate relating these variables with the method's formal parameters and the objects reachable from the parameters. In practice, this definition is supported by relating these new variables with the formal parameters using the binding invariant.

Another related issue is the fact that the method under analysis may start with a non-empty heap, holding an input data structure (such as a collection).Nevertheless, we can easily deal with non-empty initial heaps, by modifying the peak definition as:

$$\mathsf{peak}_{\Gamma}(\vartheta) \quad \triangleq \quad \max_i \mathsf{memUsed}(\Gamma_i(\rho^{\vartheta})) - \mathsf{memUsed}(\rho_0^{\vartheta}) \tag{6.2}$$

The rest of the derivations follow similarly.

Dealing with recursion and complex data structures:

Two major shortcomings of our technique are the restriction about recursion and support for more complex data structures. We do not allow recursion because our technique relies on having a finite evaluation tree. Although we believe that this restriction is acceptable for embedded systems, we are working in alternative solutions. For instance, it is possible to provide and use peak memory-requirements specification for whole mutually recursive set of methods considering them as being only one method. Besides, some particular cases, such as tail-recursion can be handled by working with a compacted call graph, provided binding invariants are available. This is the way the `bisort` case study has been analyzed in Section 3.

In [7] we present some solutions to deal with some typical iteration patterns in collections. We are also studying the possibility of combining our technique with approaches like [10, 11] that seems to be suitable for the verification of Presburger expressions accounting for memory consumption annotations for class methods. We believe that it is possible to devise a technique integrating our analysis together with those mentioned type-checking based ones. The approach would be as follows. While methods for data container classes (like the ones provided by standard libraries) are annotated and verified by type-checking techniques, loop-intensive applications built on top of those verified libraries may be analyzed using our

approach. Benefits are twofold: first, work done by our technique would be reduced since we would have to deal with significatively smaller call graphs, and second, our ability to synthesize non-linear consumption expressions would entail an increase of expressive power of type-checking based techniques.

7 Related Work

The problem of dynamic memory estimation has been studied for functional languages in [18, 19, 25]. The work in [18] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a particular memory mechanism (free list of cells and explicit deallocation in pattern matching). 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 it does compute non-linear parametric expressions. [19] proposed a variant of ML extended with region constructs [24] together with a type system based on the notion of sized types [20] (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 objects deallocation. The technique proposed in [25] 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. The evaluation of the bound function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed in [17, 10, 11, 1]. The technique of [17] 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. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, three points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables and third, it does not consider any memory collection mechanism. The method proposed in [10, 11] relies on a type system and type annotations, similar to [19]. It does not actually synthesize memory bounds, but 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 we can infer non-linear bounds. The reason we do not support individual object deallocation is our current impossibility of computing lower bounds which are required for safely compare the difference between allocations and deallocations. More recently Alter et al. [1] propose a technique for parametric cost analysis for sequential Java code. The code is translated to a recursive representation with a flattened stack. Then, they infer *size relations* which are similar to our linear invariants. 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 the bounds that this technique is able to infer 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. They outline some proposals to approximate solutions. 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. The technique is developed for region-based dynamic memory management, when regions are directly associated with methods, but it can be used safely to predict memory requirements for memory management mechanism that free memory by demand. The inputs of the techniques are the application call graph enriched with binding invariant information to constraint calling contexts, a set of parametric expressions that bounds the size of every region and a mapping from allocation points to regions (we can compute both information using the technique proposed in [7]) and yields a parametric certificate of the memory required to run a method (or program). This certificates are given in the

form of evaluation trees that can be easily translated to code that can be evaluated in runtime to pre-allocate regions. The size of the evaluation trees are known at compile time and can be reduced either using mathematical tools to symbolically solve maximums between polynomials or by compromising some accuracy of run-time calculations.

The precision of the technique relies on several factors: the precision of the inputs (region sizes and invariants), the structure of the program 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 results on JOlden let us think it is a promising approach. We are working to enhance our technique to support recursion and to combine it with others which are better suited for more complex or recursive data structures.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP 07*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007. 7

[2] D. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. *In EMSOFT'04*, 2004. 1

[3] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *SEFM 05*, pages 86–95. IEEE Computer Society, 2005. 2.2

[4] S. Bernstein. *Collected Works*, volume 1-2. USSR Academy of Sciences, 1954. 4.1

[5] G. Bollella and J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000. 1

[6] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Symbolic prediction of dynamic memory requirements using bernstein basis. Tech.Rep, DC, FCEyN. UBA, oct 2007. 4.2

[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. 1, 3.1, 3.2, 6.0.1, 6.0.3, 8

[8] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001. 5

[9] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004. 1

[10] W. Chin, S. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE 2005*, 2005. 6.0.3, 7

[11] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005. 2.2, 6.0.3, 7

[12] Ph. Clauss and I. Tchoupaeva. A symbolic approach to Bernstein expansion for program analysis and optimization. In *CC 04*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004. 1, 4.1

[13] P. Clauss, F. Fernández, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. Tech.Rep.06-04, Université Louis Pasteur, oct 2006. 4.1

[14] D. Detlefs. A hard look at hard real-time garbage collection. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC04)*, page 2332, Vienna, May 2004. 1

[15] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004. 1, 2.2

[16] D. Gay and A. Aiken. Language support for regions. In *PLDI 01*, pages 70–80, 2001. 1

[17] O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002. 1, 7

[18] M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, New Orleans, LA, January 2003. 7

[19] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*, 1999. 7

[20] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996. 7

[21] Ch. Kloukinas, Ch. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *EMSOFT'03*, Philadelphia, USA, October 2003. 1

[22] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, August 2007. 2.2

[23] G. Salagnac, S. Yovine, and D. Garbervetsky. Fast escape analysis for region-based memory management. *ENTCS*, 131:99–110, 2005. 2.2

[24] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997. 7

[25] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Optimized live heap bound analysis. In *VMCAI 03*, volume 2575 of *LNCS*, pages 70–85, January 2003. 7