Science of Computer Programming 92 (2014) 56–84

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Summary-based inference of quantitative bounds of live heap objects

Víctor Braberman[a,b], Diego Garbervetsky[a,b,*], Samuel Hym[c], Sergio Yovine[a,b]

[a] Departamento de Computación, FCEyN, UBA, Argentina
[b] CONICET, Argentina
[c] LIFL, Université Lille 1 & CNRS, France

ARTICLE INFO

Article history:
Received 31 July 2012
Received in revised form 26 October 2013
Accepted 17 November 2013
Available online 28 November 2013

Keywords:
Resource consumption
Quantitative analysis
Heap memory requirements

ABSTRACT

This article presents a symbolic static analysis for computing parametric upper bounds of the number of simultaneously live objects of sequential Java-like programs. Inferring the peak amount of irreclaimable objects is the cornerstone for analyzing potential heap-memory consumption of stand-alone applications or libraries. The analysis builds method-level summaries quantifying the peak number of live objects and the number of escaping objects. Summaries are built by resorting to summaries of their callees. The usability, scalability and precision of the technique is validated by successfully predicting the object heap usage of a medium-size, real-life application which is significantly larger than other previously reported case-studies.

1. Introduction

There is an increasing interest in understanding and analyzing the use of resources in software and hardware systems [16,11,3,25,22]. Indeed, assessing application behavior in terms of resource usage is of uttermost importance for engineering a wide variety of software-intensive systems ranging from embedded to cloud computing.

Heap-memory consumption is a particularly challenging case of resource-usage analysis due to its non-cumulative nature. Inferring memory consumption for Java-like programs requires not only computing bounds for allocations but also considering potential deallocations made by automatic reclaiming.

Although there have been significant results on this research topic (see Section 7), scalability remains an open challenge towards handling real-life object-oriented applications. Only a few small real-life programs were reported as analyzed in the literature by approaches targeting Java-like languages. Those approaches were based on recurrence equation solving, implemented in Costa [3,6], or on symbolic calculation over iteration spaces, implemented in JConsume [11,21]. Costa translates programs into sets of recurrence equations. It is fully automatic but, in general, programs need to be adapted or simplified in order to be analyzed. On the other hand, JConsume does not require the code to be modified but it imposes program invariants to be provided. In many cases, such information could be obtained automatically, but it sometimes demands manual annotations. They both suffer from scalability problems as they cannot even deal with medium-size applications. JConsume has been reported to be able to cope with a few small real-life programs in [21]. However, those examples possibly represent the largest programs which could be successfully handled by that approach with reasonable amount of human effort. In order to overcome the scalability issue, using compositional approaches is a promising direction. By compositional analysis we mean techniques where information of a module (in this case, memory consumption) is computed by using

* Corresponding author.
E-mail addresses: vbraber@dc.uba.ar (V. Braberman), diegog@dc.uba.ar (D. Garbervetsky), samuel.hym@univ-lille1.fr (S. Hym), syovine@dc.uba.ar (S. Yovine).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.11.036

the information (specifications, annotations or summaries) of the modules it uses (calls). We believe compositional techniques should not only be developed to achieve algorithmic scalability. Indeed, humans should be able to annotate pieces of code with summaries to deal with accidental or inherent limitations of particular analyses (e.g., treating unavailable or unanalyzable pieces of code, handling imprecision, etc.).

This paper is focused on computing summaries that include parametric expressions that over-approximate the maximum number of simultaneously *live* objects ever created by a method, where by live we mean irreclaimable by *any* reachability-based garbage collector. Computing the number of live objects is a key underlying step in all client analysis techniques aiming at computing dynamic-memory requirements and it seems to be a reasonable proxy to understand how consumption depends on parameters and to assess alternatives. More precisely, this paper presents a *summary-based* quantitative, static and flow-insensitive analysis aimed at inferring non-linear upper bounds on the number of irreclaimable objects that may be stored in the heap at any point in the execution of Java-like programs, including dynamic binding and implicit memory management. The analysis is based on the construction of modular *live objects summaries* for every method under analysis. More concretely, it over-approximates both a) the maximum amount of fresh objects which are simultaneously alive *during* the execution of the method, and b) the number of created objects that may remain alive *after* the method finishes its execution. Summaries are built up using method-local information (e.g., its own allocations), precomputed summaries of callees, and object lifetime information. Since the behavior of a method varies according to the values of the arguments it is called with, summaries are parametric in order to provide bounds depending on the arguments.

The technique is illustrated on a real-life Java program. It is applied in a highly automated way yielding non-obvious and very precise results. To our knowledge, this case study is about 10 times larger in terms of number of lines of code, methods, classes and allocations, than the largest one previously reported in [21], as well as programs handled by other tools (see Section 7). The application features non-trivial programming idioms such as lazy initialization, dynamic binding and reflection. We obtained very tight bounds almost automatically, with a very limited but key user intervention. We also analyze if there exists a potential loss of precision with respect to non-compositional techniques by evaluating them using a common benchmark.

1.1. Paper structure

In Section 2 we informally introduce our analysis through a set of simple but illustrative examples of growing complexity. In Section 3 we define the notion of live objects summaries. In Section 4 we propose an algorithm to compute them, using external analyses. In Section 5 we describe some details of our implementation which computes summaries as piecewise integral polynomials and which instantiates the helper analyses. In Section 6 we present our empirical evaluation. Finally, in Section 7 and Section 8 we discuss related and future work, respectively.

2. Informal presentation through motivating examples

This section gives a step-by-step informal presentation of our summary-based technique for building method-level, live-objects summaries, through several motivating examples of increasing complexity.

2.1. Counting allocations

```
0  void triangle(A[][] a, int n) {
1     for(int i = 1; i <= n; i ++)
2        line(a, i);
3  }

4  void line(A[][] a, int m) {
5     for(int j = 1; j <= m; j ++)
6        a[m−1][j−1] = new A();
7  }
```

Program 1. A simple program with one method allocating objects in a loop and one client.

Consider Program 1. Calling the method `triangle` will put newly-allocated objects in the 2-dimensional array `a` following a triangular shape: it generates i new objects stored in the i-th line.

This is divided into two methods. Method `line` fills in a line of `a`: at line 6, one object is allocated and stored in a cell of `a`. Method `triangle` invokes `line` at line 2 to fill in each row. Let us assume here that this runs without any exception being thrown.

A tool like Costa [3,6] transforms the program into a set of recurrence equations, and returns as result a count of n^2 allocations. Our previous work [11], based on sizing *iteration spaces*, is able to obtain the exact bound of $\frac{n(n+1)}{2}$ allocations, which is smaller than n^2. To do so, it first infers the whole-program loop-invariant $\{1 \leqslant j \leqslant i \leqslant n\}$, and then counts the number of integer points in this invariant as a function of `triangle` argument n. Although this technique works fine for this example, it does not scale.

Therefore, we propose to analyze each method at a time in a *compositional* fashion. We can start by analyzing method `line`. This method performs at line 6 an allocation within a loop that will be executed m times. Our method is flow-insensitive (i.e., we do not care about the ordering of allocations and method calls) but uses invariants to represent iteration spaces corresponding to loops. Such invariants can either be provided by hand by the programmer or inferred