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.*
[b] *CONICET.*
[c] *LIFL, Université Lille 1 & CNRS.*

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 §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

*Corresponding author

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 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 §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 §2 we informally introduce our analysis through a set of simple but illustrative examples of growing complexity. In §3 we define the notion of live objects summaries In §4 we propose an algorithm to compute them, using external analyses. In §5 we describe some details of our implementation which computes summaries as piecewise integral polynomials and which instantiates the helper analyses. In §6 we present our empirical evaluation. Finally, in §7 and §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 $\{1 \leq j \leq i \leq n\}$, 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 using Daikon [19] as explained in [10, 11]. In contrast to our previous approach, here we resort to method-local invariants, instead of whole-program ones. In our example, we use an invariant characterizing the iteration space for the allocation at line 6:

$$Inv_6 = 1 \leq j \leq m$$

The number of integer solutions to this invariant (in terms of m) is m. Thus, a *summary* of the maximum number of objects which are alive during the execution of `line`, denoted $\mathsf{MaxLive}_{\texttt{line}}$, can be expressed as a function of m as follows:

$$\mathsf{MaxLive}_{\texttt{line}} = \sum_{Inv_6} 1 = \sum_{1 \leq j \leq m} 1 = m$$

Let SC be a *symbolic calculator* providing an operation *SC.Summate* which computes the sum of an expression over an iteration space, described by an invariant. The previous equation becomes:

$$\mathsf{MaxLive}_{\texttt{line}} = SC.Summate(1, Inv_6, m) = m$$

Remark. More generally: let $\bar{x}$ be variables, $\bar{p}$ be parameters, and $Inv(\bar{x}, \bar{p})$ be an invariant; an *iteration space* $IS(Inv, \bar{p})$ is the set of all possible values for $\bar{x}$ so that the invariant Inv is verified for some value of the parameters $\bar{p}$. In general, *SC.Summate* is a symbolic operation returning the sum of a parametric expression $e(\bar{x}, \bar{p})$ over $IS(Inv, \bar{p})$ as a closed-form in terms of parameters $\bar{p}$:

$$SC.Summate(e, Inv, \bar{p}) = \sum_{IS(Inv, \bar{p})} e$$

Hereinafter, we will use *SC.Summate*. □

In order to be able to compose the results, we need to distinguish whether the objects allocated in the method live longer than the method itself (i.e. they *escape* its scope). All objects allocated at line 6 are assigned to the parameter a and, thus, they may live longer than `line`.

So, a summary of the amount of objects escaping `line`, in terms of its parameter m, is:

$$\mathsf{Esc}_{\texttt{line}} = \mathsf{MaxLive}_{\texttt{line}} = m$$

2.2. Analyzing method invocations

Consider now the method `triangle` in Program 1 invoking `line(a, i)` at line 2 within a loop in which i varies at each iteration. Therefore, its memory profile depends on the behavior of `line`, for which we already have a summary. Instantiating the summary of `line` with the arguments of the call ($m = i$) we get that i objects will be allocated at iteration i.

The invariant characterizing the iteration space for the invocation is:

$$Inv_2 = 1 \leq i \leq n$$

Objects escaping `line` will live in `triangle`'s scope and remain alive from one iteration to the other. Thus, to summarize the amount of living objects one needs to *accumulate* the number of escaping objects. More formally:

$$\begin{aligned} \mathsf{MaxLive}_{\texttt{triangle}} &= SC.Summate(\mathsf{Esc}_{\texttt{line}}[m/i], Inv_2, n) \\ &= SC.Summate(i, 1 \leq i \leq n, n) = \frac{n(n+1)}{2} \end{aligned}$$

where $e[x/y]$ stands for replacing all occurrences of x in e by y.

All objects escaping method `line`, also escape from `triangle` because they are referenced by parameter a. So, the quantitative dynamic-memory profile of `triangle` can be expressed by the following summary:

$$\mathsf{Esc}_{\texttt{triangle}} = \mathsf{MaxLive}_{\texttt{triangle}} = \frac{n(n+1)}{2}$$

Remark. To infer the summary of a method we need to obtain the summaries of the method its calls. In general, we will apply a bottom-up traversal of the call graph to ensure that a summary for a callee is always available (see Section 5.5 for more details on recursion). □

2.3. Distinguishing between escaping and non-escaping objects

In this work, we study object lifetime with method granularity. That is, we assume dead objects are collected upon method exit.[1] Therefore, object lifetime can be characterized by which objects *escape from*, i.e., live longer than, a method.

Consider the code in Program 2 defining a `map` operation over lists using a class `Op` containing a method `apply`. In this example this method is used to multiply an integer value by a random number.

Let us first look at `apply`, since it is invoked by `map`. This method performs 3 allocations. An object created at line 16 *escapes* the scope of the method since it is returned. Objects allocated at lines 12 and 14 can be safely reclaimed when `apply` finishes its execution. Besides, `apply` only invokes methods that do not perform allocations. Thus, the behavior of `apply` can be summarized as follows:

$$\begin{aligned} \mathsf{MaxLive}_{\texttt{apply}} &= 3 \\ \mathsf{Esc}_{\texttt{apply}} &= 1 \end{aligned}$$

[1] Clearly, this is an over-approximation. Nevertheless, the granularity of the analysis could be made finer at the expense of computational cost.

```
0  public List map(List list, Op op) {
1    List res = new LinkedList();
2    Iterator it = list.iterator();
3    for (;;it.hasNext()) {
4      Object li = it.next();
5      Object o=op.apply(li);
6      res.add(o);
7    }
8    return res;
9  }
```

```
10 public class Op {
11   public Object apply(Object o) {
12     Date d = new Date();
13     Random r;
14     r = new Random(d.getTime());
15     int v = ((Integer) o).intValue();
16     return new Integer(v*r.nextInt());
17   }
18 }
```

Program 2: A method implementing a generic map operation

2.4. Invoking several methods

Let us now consider the statements in `map` that allocate memory: the `new` in line 1, and 3 method invocations, namely in lines 2, 5 and 6, with the last two inside a loop. To compute a summary for method `map` we will use intermediate results to make the formulas easier to read. These intermediate results are attached to lines of the source corresponding to statements that allocate memory.

Lines 1 and 2 are the simplest cases. We can associate the following simple summary to any `new`:

$$\mathsf{MaxLive}_1 = \mathsf{MaxLive}_{\texttt{new}} = 1$$
$$\mathsf{Esc}_1 = \mathsf{Esc}_{\texttt{new}} = 1$$

Similarly for the iterator:

$$\mathsf{MaxLive}_2 = \mathsf{MaxLive}_{\texttt{iterator}} = 1$$
$$\mathsf{Esc}_2 = \mathsf{Esc}_{\texttt{iterator}} = 1$$

The call to the method `apply` at line 5 is inside a loop. Therefore, we resort to an invariant to characterize the iteration space corresponding to that statement:

$$\mathit{Inv}_5 = 1 \leq i \leq \mathit{list\_size}$$

where *list_size* is a formal variable representing the length of the list (i.e., `list.size`). Objects escaping from `apply` will live in `map`'s scope. Clearly, such number contributes to MaxLive of `map`. Thus, the amount of escaping objects from `apply` at each iteration must be accumulated as it contributes to the total amount of live objects in the scope of `map`. Besides, objects not escaping from `apply` are also alive during (part of) the execution of `map`. Therefore, they should also be counted in MaxLive of `map`. However, since the number of those objects may be different at each iteration, it would be sufficient to count only the maximum such number over all iterations to obtain an over-approximation.

That is,

$$\begin{aligned}\mathsf{MaxLive}_5 &= \bigsqcup_{Inv_5} (\mathsf{MaxLive}_{\texttt{apply}} - \mathsf{Esc}_{\texttt{apply}}) + SC.Summate(\mathsf{Esc}_{\texttt{apply}}, Inv_5, list\_size) \\ &= \bigsqcup_{1 \le i \le list\_size} (3-1) + SC.Summate(1, 1 \le i \le list\_size, list\_size) \\ &= 2 + list\_size\end{aligned}$$

Notice that we subtract the number of escaping objects from the overall number of live objects of `apply` since this amount is indeed accounted for in the *SC.Summate*.

Let the symbolic calculator *SC* provide an operation *SC.UpBound* which computes the max of an expression over a domain. In this case, we could write:

$$\begin{aligned}\mathsf{MaxLive}_5 = SC.UpBound(&\mathsf{MaxLive}_{\texttt{apply}} - \mathsf{Esc}_{\texttt{apply}}, Inv_5, list\_size) \\ &+ SC.Summate(\mathsf{Esc}_{\texttt{apply}}, Inv_5, list\_size)\end{aligned}$$

Remark. More precisely, given an expression $e(\bar{x}, \bar{p})$ and an invariant $Inv(\bar{x}, \bar{p})$, *SC.UpBound* computes a symbolic expression in terms of $\bar{p}$ that maximizes e for all values of $\bar{x}$ such that $Inv(\bar{x}, \bar{p})$ holds. Formally,

$$SC.UpBound(e, Inv, \bar{p}) = \bigsqcup_{IS(Inv, \bar{p})} e$$

Hereinafter, we will use *SC.UpBound*. □

Let us look now at Esc_5. All objects escaping from `apply` are added to the returned list. Consequently, they also escape from `map`. Therefore:

$$\begin{aligned}\mathsf{Esc}_5 &= SC.Summate(\mathsf{Esc}_{\texttt{apply}}, Inv_5, list\_size) \\ &= SC.Summate(1, 1 \le i \le list\_size, list\_size) = list\_size\end{aligned}$$

To compute the intermediate result for line 6, we make use of the summary of method called at this line, that is `add`. Its summary is:

$$\begin{aligned}\mathsf{MaxLive}_{\texttt{add}} &= 1 \\ \mathsf{Esc}_{\texttt{add}} &= 1\end{aligned}$$

Since the call is in the loop, we proceed as for `apply`. The invariant at line 6 is:

$$Inv_6 = 1 \le i \le list\_size$$

MaxLive and Esc at at line 6 is computed as follows:

$$\begin{aligned}\mathsf{MaxLive}_6 &= SC.UpBound\big(\mathsf{MaxLive}_{\texttt{add}} - \mathsf{Esc}_{\texttt{add}}, Inv_6, list\_size\big) \\ &\quad + SC.Summate(\mathsf{Esc}_{\texttt{add}}, Inv_6, list\_size) \\ &= 0 + list\_size = list\_size \\ \mathsf{Esc}_6 &= SC.Summate(\mathsf{Esc}_{\texttt{add}}, Inv_6, list\_size) = list\_size\end{aligned}$$

Finally, to obtain the summary of `map` we compose the previous results as follows. MaxLive for `map` is the sum of the corresponding Esc for all lines plus the the maximum MaxLive of all lines but discounting the corresponding Esc:

$$\begin{aligned}\mathsf{MaxLive}_{\texttt{map}} &= \bigsqcup_{l\in\{1,2,5,6\}} \left(\mathsf{MaxLive}_l - \mathsf{Esc}_l\right) + \sum_{l\in\{1,2,5,6\}} \mathsf{Esc}_l \\ &= \sqcup\ \{1-1, 1-1, \mathit{list\_size}+2-\mathit{list\_size}, \mathit{list\_size}-\mathit{list\_size}\} \\ &\quad + (1+1+\mathit{list\_size}+\mathit{list\_size}) \\ &= 2 + (2 + 2\times \mathit{list\_size}) = 2\times \mathit{list\_size} + 4\end{aligned}$$

Esc of `map` should count all objects that (may) escape from `map`.[2] From all objects accounted for in Esc_l, for $l \in \{1,2,5,6\}$, only the iterator allocated at line 2 is captured by `map`. All other objects escape. Thus, we get:

$$\begin{aligned}\mathsf{Esc}_{\texttt{map}} &= \mathsf{Esc}_1 + \mathsf{Esc}_5 + \mathsf{Esc}_6 \\ &= 1 + \mathit{list\_size} + \mathit{list\_size} \\ &= 2\times \mathit{list\_size} + 1\end{aligned}$$

2.5. Analyzing virtual method calls

Here we show how we handle dynamic binding. Consider the code in Program 3. It shows a subclass of `Op` that overrides `apply`. This version returns a two-element array containing a new random `Integer` but also a copy of the original one.

```
0 public class Op2 extends Op {
1   public Object apply(Object object) {
2     Date d = new Date();
3     Random r = new Random(d.getTime());
4     int v = ((Integer) object).intValue();
5     return new Integer[]{new Integer(value), new Integer(v * r.nextInt())};
6   }
7 }
```

Program 3: Extending the class `Op`.

`Op2.apply` always allocates the same number of objects: it returns a new array of size 2 containing 2 newly created objects. Our analysis considers an array of n cells just as if they were n objects (and not simply one). This is because the actual memory required for the array will depend on n. Notice that, although the actual memory consumption for an array may depend of other (static and run-time) parameters of the VM, it is in general highly correlated with its size[3].

[2] The "may" refers to the need to compute an over-approximation to ensure correctness.

[3] In some JVMs for embedded systems, such as KVM [32] and JITS [17], the memory allocated for arrays is directly proportional to their number of cells.

`Op2.apply` also allocates 2 other objects (lines 2 and 3) which do not escape the method. Thus, its summary is:

$$\begin{aligned}\mathsf{MaxLive}_{\texttt{Op2.apply}} &= 6\\ \mathsf{Esc}_{\texttt{Op2.apply}} &= 4\end{aligned}$$

Now, to analyze `map`, we need to consider that the call to `apply` at line 5 could be one of `Op.apply` or `Op2.apply`. In order to get a safe over-approximation of the memory requirements at this line, we must take into account the maximum of both MaxLive and Esc for each candidate. Since the call is inside a loop, the maximum could be achieved by a *different* method at each iteration. Thus, we end up computing:

$$\begin{aligned}\mathsf{MaxLive}_5 &= SC.UpBound(\bigsqcup_{u\in\{\texttt{Op.apply},\texttt{Op2.apply}\}} (\mathsf{MaxLive}_u - \mathsf{Esc}_u), Inv_5, list\_size)\\ &\quad + SC.Summate(\bigsqcup_{u\in\{\texttt{Op.apply},\texttt{Op2.apply}\}} \mathsf{Esc}_u, Inv_5, list\_size)\\ &= SC.UpBound(\sqcup\{2,2\}, Inv_5, list\_size)\\ &\quad + SC.Summate(\sqcup\{1,4\}, Inv_5, list\_size)\\ &= 2 + 4 \times list\_size\end{aligned}$$

We proceed in the same way for the escaping objects. In this case, all objects escaping from both `Op.apply` and `Op2.apply` also escape from `map`. Thus, we obtain:

$$\begin{aligned}\mathsf{Esc}_5 &= SC.Summate(\bigsqcup_{u\in\{\texttt{Op.apply},\texttt{Op2.apply}\}} \mathsf{Esc}_u, Inv_5, list\_size)\\ &= SC.Summate(\sqcup\{1,4\}, Inv_5, list\_size)\\ &= 4 \times list\_size\end{aligned}$$

Therefore, the new summary for `map` is:

$$\begin{aligned}\mathsf{MaxLive}_{\texttt{map}} &= 5 \times list\_size + 4\\ \mathsf{Esc}_{\texttt{map}} &= 5 \times list\_size + 1\end{aligned}$$

Remark. Works like [2, 11] consider the set of candidates of a virtual call as if they were all be called. Clearly, this leads to an over-approximation. Here, we propose a more precise approach, which consists in taking into account the *most consuming part of each candidate* as the consumption of the virtual call.

2.6. Introducing new parameters

In Program 4 we present an example illustrating the ability to introduce summary parameters.

```
0  List copy(List list) {
1    List res = new LinkedList();
2    Iterator it = list.iterator();
3    for (;; it.hasNext())
4      res.add(it.next());
5    return res;
6  }
7
8  List safeMap(List list,Op op) {
9    List cp = copy(list);
10   return map(cp, op);
11 }
12 List test(List<List> ls, Op op){
13   List res = new LinkedList();
14   Iterator it = ls.iterator();
15   for(;; it.hasNext()){
16     List l = it.next();
17     l = safeMap(l, op);
18     res.add(l);
19   }
20   return res;
21 }
```

Program 4: An example with a more complex iteration pattern.

Method `safeMap` invokes `copy` and `map`. Method `copy` just makes a copy of the input by traversing it using an iterator. Following the approach detailed so far, we obtain the summary of `copy`:

$$\mathsf{MaxLive}_9 = \mathsf{MaxLive}_{\texttt{copy}} = \mathit{list\_size} + 2$$
$$\mathsf{Esc}_9 = \mathsf{Esc}_{\texttt{copy}} = \mathit{list\_size} + 1$$

The summary of `map` has already been computed above. Notice that the call to `map` is done with *cp*. Thus, we need to instantiate *list_size* with *cp_size*. Besides, we make use of the invariant[4] *cp_size* = *list_size* to obtain the intermediate result at line 9:

$$\begin{aligned}\mathsf{MaxLive}_{10} &= \mathit{SC.UpBound}(\mathsf{MaxLive}_{\texttt{map}}[\mathit{list\_size}/\mathit{cp\_size}] - \mathsf{Esc}_{\texttt{map}}[\mathit{list\_size}/\mathit{cp\_size}],\\ &\qquad \mathit{list\_size} = \mathit{cp\_size}, \mathit{list\_size})\\ &= \mathit{list\_size} + 2\\ \mathsf{Esc}_{10} &= \mathit{SC.Summate}(\mathsf{Esc}_{\texttt{map}}[\mathit{list\_size}/\mathit{cp\_size}], \mathit{list\_size} = \mathit{cp\_size}, \mathit{list\_size})\\ &= \mathit{list\_size} + 1\end{aligned}$$

Applying the technique explained previously for composing the intermediate results obtained for all lines, the summary of `SafeMap` becomes:

$$\mathsf{MaxLive}_{\texttt{safeMap}} = 6 \times \mathit{list\_size} + 5$$
$$\mathsf{Esc}_{\texttt{safeMap}} = 5 \times \mathit{list\_size} + 1$$

The method `test` receives a list of lists and returns a new list generated by applying `safeMap` to each element of the input. Proceeding as in §2.4, we would like to obtain:

[4]The formalization of this mechanism will be explained later

$$
\begin{aligned}
\mathsf{MaxLive}_{17} &= \mathit{SC.UpBound}\big(\mathsf{MaxLive}_{\texttt{safeMap}}[\text{for } \texttt{ls[i]}] \\
&\qquad\qquad - \mathsf{Esc}_{\texttt{safeMap}}[\text{for } \texttt{ls[i]}], \mathit{Inv}_{17}, \mathit{list\_size}\big) \\
&\quad + \mathit{SC.Summate}\big(\mathsf{Esc}_{\texttt{safeMap}}[\text{for } \texttt{ls[i]}], \mathit{Inv}_{17}, \mathit{list\_size}\big) \\
&= \max_{1 \le i \le ls\_size} (ls[i]\_size + 4) + \sum_{1 \le i \le ls\_size} (5 \times ls[i]\_size + 1) \\
\mathsf{Esc}_{17} &= \mathit{SC.Summate}\big(\mathsf{Esc}_{\texttt{safeMap}}[\text{for } \texttt{ls[i]}], \mathit{Inv}_{17}, \mathit{list\_size}\big) \\
&= \sum_{1 \le i \le ls\_size} (5 \times ls[i]\_size + 1)
\end{aligned}
$$

where

$$\mathit{Inv}_{17} = 1 \le i \le \mathit{list\_size}$$

With such a formulation, the number of parameters in $\mathsf{MaxLive}_{17}$ and Esc_{17} is unbounded, since they depend on *ls_size* and on all the *ls*[*i*]*_size*. This makes the expressions tricky to handle symbolically. But in cases where *ls*[*i*]*_size* can be represented with a fixed number of parameters (e.g., when the *ls* is a matrix, with all the *ls*[*i*]*_size* equal, or when *ls*[*i*]*_size* is an expression of i, etc.) the expressions may be simplified.

To deal with such cases we allow the programmer to introduce new parameters. In this case, one could introduce a parameter *maxSize* bounding the size of lists in *ls*. Using *maxSize*, we can get the following summary:

$$
\begin{aligned}
\mathsf{MaxLive}_{17} &= \mathit{SC.UpBound}\big(\mathit{maxSize} + 4, \mathit{Inv}_{17}, \{\mathit{list\_size},\ \mathit{maxSize}\}\big) \\
&\quad + \mathit{SC.Summate}\big(5 \times \mathit{maxSize} + 1, \mathit{Inv}_{17}, \{\mathit{list\_size},\ \mathit{maxSize}\}\big) \\
&= \mathit{maxSize} + 4 + \mathit{ls\_size} \times (5 \times \mathit{maxSize} + 1) \\
\mathsf{Esc}_{17} &= \mathit{SC.Summate}\big(5 \times \mathit{maxSize} + 1, \mathit{Inv}_{17}, \{\mathit{list\_size},\ \mathit{maxSize}\}\big) \\
&= \mathit{ls\_size} \times (5 \times \mathit{maxSize} + 1)
\end{aligned}
$$

This yields the following summary for `test`:

$$
\begin{aligned}
\mathsf{MaxLive}_{\texttt{test}} &= 1 + 1 + \mathsf{MaxLive}_{17} + \mathit{ls\_size} \\
&= \mathit{ls\_size} \times (5 \times \mathit{maxSize} + 2) + \mathit{maxSize} + 6 \\
\mathsf{Esc}_{\texttt{test}} &= 1 + \mathsf{Esc}_{17} + \mathit{ls\_size} \\
&= \mathit{ls\_size} \times (5 \times \mathit{maxSize} + 2) + 1
\end{aligned}
$$

Of course, this is an over-approximation but the obtained bound depends on parameters more amenable to a symbolic analysis.

2.7. Refining Esc

As we pointed out in the previous examples, Esc by itself is not quite sufficient in a modular setting. Indeed, when we compute Esc for a method `m`, we

need to distinguish, in the Esc of the methods it calls, the part of the objects that will further escape from `m`. To this end, we need to refine Esc.

Moreover, in all previous examples, the only way objects escape out of a method is because they are returned at the end. In general, objects may escape in many different ways: through modifications of arguments, through global variables, etc. So we refine Esc regarding escape channels.

Let us consider some simple methods to illustrate how we proceed.

```
0  public B two(A a) {
1    a.f = new B();
2    return new B();
3  }
4
5  public B keepOne() {
6    return two(new A());
7  }
8  public B aliasIt(A a) {
9    a.f = new B();
10   return a.f;
11 }
```

Program 5: Need for refined Esc

First consider `two` and `keepOne` in Program 5. We can easily see that the two objects allocated in `two` escape its scope, one via the return value, the other via its parameter `a`. When this method is called by `keepOne`, the returned object escapes, the other is captured. In order to get the precise result that only one object escapes from `keepOne`, we need to refine the $\mathsf{Esc}_{\texttt{two}}$ by parametrizing Esc with the way the objects escape:

$$\mathsf{Esc}_{\texttt{two}}(\{ret\}) = 1$$
$$\mathsf{Esc}_{\texttt{two}}(\{a.f\}) = 1$$

Thus, they can be plugged in the analysis of `keepOne` so that we can obtain:

$$\mathsf{Esc}_{\texttt{keepOne}}(\{ret\}) = 1$$

instead of 2.

Moreover, if we simply define Esc for each channel through which an object can escape, we also lose some precision. Consider for instance `aliasIt` in Program 5. As for `two`, we have:

$$\mathsf{Esc}_{\texttt{aliasIt}}(\{ret\}) = 1$$
$$\mathsf{Esc}_{\texttt{aliasIt}}(\{a.f\}) = 1$$

but it is the same object that escapes through both channels, so we will precise:

$$\mathsf{Esc}_{\texttt{aliasIt}}(\{ret, a.f\}) = 1$$

In this example, we have used ret and $a.f$ to describe the sub-parts of the heap that escape. The analysis we present here can be deployed with different families of descriptors for these sub-heaps. This is explained in more detail in §3.

Up until now, we have only used Esc without any sub-heap specification, to mean a bound to the number of objects that might escape through any of the escape channels. So, for instance, $\mathsf{Esc}_{\texttt{test}}$ of Section 2.6 stands for $\mathsf{Esc}_{\texttt{test}}(\{ret\})$ since objects can only escape from `test` via its returned value, which means that the full set of its sub-heap descriptors is $\{ret\}$.

In the following sections we formalize the concept of live objects summaries and present an analysis following the ideas presented here.

3. Live objects summaries

In this section we formally define the notions informally presented in Section 2, and their expected properties.

The main goal of our work is to provide bounds for the number of objects that could be simultaneously alive during any execution of a given program. To do that, our technique computes *live objects summaries* that enable *modular* treatment of a program. That is, *live objects summaries* contain enough information to compute the summary of a method solely in terms of the summaries of all the methods it calls.

For our purpose, we analyze the code assuming that it runs with all the memory required since we want to obtain a bound for the general case: running some program with a bounded memory could only use less memory, in case it aborts when it reaches the limit. We therefore assume that no `OutOfMemoryException` is raised during the runs.

3.1. Invocation Runs

Let *Prog* be a sequential Java-like program containing a set of methods. A method m has a name $m.name$, a signature $m.signature$ including formal parameters $\bar{fp}$, and a body $m.body$.

The soundness properties of the technique will be explained in terms of an abstraction of (Java) program runs[5]. We assume in this work that no exception is thrown during the considered runs. Furthermore our analysis deals with code that cannot be statically analyzed (native methods, reflection, etc.) using externally-provided summaries.

We assume an infinite set *Ref* of references (or object identifiers) and a set *BasicTypes* of values of basic types (*int*, *boolean*, etc.). We note *Val* the set of all values, namely $Ref \cup BasicTypes$. We model objects as indexed tuples of the form $(identifier : Ref, f_1 : Val, \ldots, f_n : Val)$, where the f_i are the object fields, and we write *Obj* for the set of all objects. Finally, we also assume an infinite set *Var* of program variables. As usual, all these sets are mutually disjoint.

A heap h is a finite mapping from *Ref* to *Obj*. A heap h is well-formed when, for all r in $\mathrm{dom}(h)$, $h(r).identifier = r$. Note that a well-formed heap must then be injective: this property corresponds to the fact that two distinct objects must always be at different heap locations.

[5] Actually it applies to runs of any heap-based programming language.

An environment e is a finite mapping from *Var* to *Val*. An environment e is compatible with a heap h if $range(e) \cap Ref \subseteq \mathrm{dom}(h)$.

A state is a pair $\sigma = (e, h)$ where e is compatible with h and h is well-formed. $\sigma.e$ refers to the environment part of the state σ and $\sigma.h$ refers to its heap part. For simplicity sometimes we will use $\sigma(v)$ instead of $h(e(v))$ for references and of $e(v)$ for basic types.

Consider some method $m \in \mathcal{M}$ of signature

```
T0 m(T1 fp1, ..., Tn fpn)
```

An invocation run $r_m = \sigma_0, \sigma_1, \ldots$ of a method m is a sequence of states following the semantics of the Java Virtual Machine starting at the invocation of m with an assignment of formal parameters given by $\sigma_0.e(\texttt{fp1})$, ..., $\sigma_0.e(\texttt{fpn})$ ($\sigma_0(\bar{\texttt{fp}})$ for short), and $\sigma_0.h$ is the initial heap for that invocation. Note that this sequence includes in particular the states the execution goes through while executing the invocations made from m.

If the run r is finite $|r|$ denotes its size. If it is infinite, $|r|$ stands for ∞.

If the invocation of the method finishes, the run finishes exactly after returning from that invocation to m. This state (i.e., $\sigma_{|r_m|-1}$), denoted as σ_{ret}, refers to the moment when local variables are no longer available and the remaining variables accessible from m are only *initial* parameters assignments and the return value. More formally, $dom(\sigma_{ret}.e) = dom(\sigma_0.e) \cup \texttt{ret}$ and for all $v \in dom(\sigma_0.e)$, $\sigma_0.e(v) = \sigma_{ret}.e(v)$. We will also assume that the heap domain in the run are adjusted to the *reachable* part of the heap. That is, only the set of references reachable from $\sigma_i.e$ (through the heap) are considered in $\sigma_i.h$[6]. References on the heap are unique and never reused even if the referenced object is collected (they are unique across invocation runs). In this setting, $dom(h)$ is actually the set of references to live objects of the heap.

R_m is the set of all the invocation runs to m. Hereinafter we will use r_i or σ_i for denoting the i^{th} element of a run r.

Given an invocation run r, a key concept to state the soundness of the method is the notion of an object being *live* and *fresh* at a given state. Given $0 \leq t_1 \leq t_2 \leq |r| - 1$, $\mathsf{fresh}_r(t_1, t_2)$ is defined as $dom(\sigma_{t_2}.h) - dom(\sigma_{t_1}.h)$, that is objects reachable at t_2 that were not present at t_1.

Definition 3.1. The *live and fresh* heap of a run r, denoted h_r, is $\sigma_{ret}.h$ restricted to the domain $\mathsf{fresh}_r(0, |r| - 1)$ (i.e., $\sigma_{ret}.h \restriction \mathsf{fresh}_r(0, |r| - 1)$).

A method m in a program location l can feature an invocation that could be resolved to method m' several times in the run (i.e., a loop). That means that in an invocation run r there can be found subsequences r' that are actually invocation runs of m', that is $r' \in R_{m'}$, produced by invocations to m' at l. The set of such subsequences for r is denoted $R^r_{l,m'}$.

[6]The restriction to project only the reachable part of the heap is a consequence of the interest in measuring the number of potentially usable objects (live objects).

3.2. Object Lifetime information

Summaries will also predicate on how many objects created by a method invocation end up escaping to the callers. More concretely, summaries assume the existence of previously computed or defined object lifetime information. The specification of escaping objects is actually partitioned into *sub-heaps* which represent sets of objects with similar lifetimes. This is done once again to make the composition of summary information more precise. As it will be seen later, expressions will bound the number of escaping objects at sub-heaps granularity to enable a more precise modular computation of the number of objects escaping the caller. Sub-heaps are identified using *descriptors*. We will use sh to range over descriptors and SH over sets of descriptors.

For instance, in §2.7 we identified the escaping objects returned by the method with the descriptor *ret*. The descriptors domain depend on the underlying analysis used to compute them: if a points-to-based escape analysis is used (e.g., [36, 7]), a sub-heap descriptor could be a node (or a set of nodes) in a points-to graph or a path expression referring to nodes reachable from it; if an interference-based analysis is used (e.g., [35]), descriptors could be method parameters representing sets of possibly connected objects. Several particular descriptors are explained in detail later in §5. As an example, Figure 3 (see p. 24) shows different descriptors for `apply` and `map` of Program 2.

Lifetime information system (**LIS**) abstractly models how objects escape methods. A **LIS** includes the sub-heap descriptors, a semantic mapping from descriptors to object ids and the escape function that links sub-heap descriptors of a caller to sub-heap descriptors of callees.

Definition 3.2. A Lifetime Information System (**LIS**) for a program $Prog$ consists of three elements:

- For each method m, a set $\mathcal{SH}_m$ (noted as $\mathbf{LIS}.\mathcal{SH}_m$) of all its sub-heap descriptors.
- A set of semantic functions $[.]^m_r$ for each method m of $Prog$ and invocation run $r \in R_m$.

  Each function takes a set of descriptors $SH \subseteq \mathcal{SH}_m$, and it yields a set of object ids included in $dom(h_r)$ meant to be represented by the descriptors of SH. That function must satisfy the following properties: $[\mathcal{SH}_m]^m_r = dom(h_r)$; $[(SH_1 \cup SH_2)]^m_r \subseteq [SH_1]^m_r \cup [SH_2]^m_r$
- Function $\mathbf{LIS}_m.esc_{l,m'}$ that binds sub-heap descriptors of caller m to callee sub-heap descriptors (callee m' invoked at l). Basically, $\mathbf{LIS}_m.esc_{l,m'} : \mathcal{P}(\mathcal{SH}_m) \mapsto \mathcal{P}(\mathcal{SH}_{m'})$ receives a set of sub-heap descriptors $SH \subseteq \mathcal{SH}_m$ and yields a set of sub-heap descriptors $SH' \subseteq \mathcal{SH}_{m'}$ where $m' \in M(l)$. These sub-heaps contain objects escaping from m' that may end up escaping also from m through the sub-heaps corresponding to SH. The property that the function must satisfy is that $\mathbf{LIS}_m.esc_{l,m'}(SH) \supseteq \{sh' \in \mathcal{SH}_{m'} | \exists r \in R_m, r' \in R^r_{l,m'}, [sh']^{m'}_{r'} \cap [SH]^m_r \neq \emptyset\}$. This property enforces that the result $\mathbf{LIS}_m.esc_{l,m'}(SH)$ must be a safe approximation (in terms

of sub-heap descriptors) to the actual objects escaping both the callee m' and the caller m. Given a set of sub-heap descriptors representing objects escaping a caller m, the function must return a superset of sub-heap descriptors of the callee that represents all objects created in invocations to the callee m' at l that end up escaping m by the given sub-heaps.

These properties will later ensure soundness of the modular summary computation.

3.3. Definition of summaries

The main component of the *live objects summary* of a method m is MaxLive, an expression that bounds the maximum number of objects that could be potentially allocated (fresh) and be simultaneously live (reachable) during any execution of m. The live objects summaries add a few necessary ingredients to properly define and compute MaxLive. Let us then define the various components of the *live objects summaries*.

First, the bound needs to be parametrized: for instance, in Program 2 (Section 2.3), we could not define a finite MaxLive for `map` that would not depend on the size of its argument *list*. So we use *parametric expressions* for MaxLive.

Notation. We write $\mathbb{E}$ for the set of *parametric expressions*, i.e. possibly with variables.

Definition 3.3 (Parameters $\bar{p}$)**.** The *parameters* of a live objects summary is a sequence of variables.

We write $\bar{p}_m$ for the sequence $p_1, \ldots, p_n$ of parameters for a summary of the method m, and simply $\bar{p}$ when m follows from context.

These parameters are actually *model variables*. In this section, we keep the notion of *expressions* completely generic, which is why the notion of parameters does not constrain their type. In our implementation, detailed in §5, the expressions are actually piecewise polynomials on $\mathbb{Z}$.

The summary of a method will also provide a set of equalities to bind those model variables to expressions given in terms of method formal parameters (actually, for the sake of presentation, we are assuming all preexisting values used by the method are computable from those parameters).

Definition 3.4 (Binding equalities $\mathbf{B}$)**.** For a method m, the binding equalities $\mathbf{B}_m$ of its live objects summary is a conjunction of equalities binding the parameters $\bar{p}_m$ of the live objects summary to the formal parameters $\bar{\texttt{fp}}$ of m.

Informally speaking, $\mathsf{MaxLive}_m$ is an expression of $\mathbb{E}$ such that, for any run r in R_m, $\mathsf{MaxLive}_m$ is an upper bound to the number of fresh objects simultaneously live at any time during r_m. The variables of that expression must be in $\bar{p}_m$ and are interpreted into concrete values according to $\mathbf{B}$.

In the simplest case $\bar{p}_m$ and $\mathbf{B}$ can be automatically derived from method parameters $\bar{\texttt{fp}}$ (e.g., when method parameters are scalars), but in general more

involvement is required, like in the examples presented in §2. In [10] we discussed some techniques for inferring model variables using the method body.

To enable a modular computation of $\mathsf{MaxLive}_m$, summaries also need to provide Esc, an over-approximation of the *escaping* objects allocated by the method, i.e. the part of MaxLive that can escape from the method and is therefore reachable from the caller. Indeed, if summaries contained only MaxLive, the bound on the number of live objects of the caller would be too rough if computed as the sum of MaxLive of the callees. For instance, consider a method that just calls two other methods, `m1` and `m2`. With only MaxLive, the only over-approximation we could end up with would be the sum of the MaxLive of the called methods. But some of the objects allocated during the run of `m1` might be collected and the corresponding space reused for allocating objects created by `m2` or vice versa.

Definition 3.5 (Esc)**.** Consider some method m. $\mathsf{Esc} : \mathcal{P}(\mathcal{SH}_m) \mapsto \mathbb{E}$ maps a set of sub-heap descriptors SH to an expression. As it will be stated later, that expression is meant to over-approximate the maximum number of objects allocated during any execution of m escaping via at least one of the sub-heaps in SH. The variables of those expressions must be in $\bar{p}_m$.

Definition 3.6 (Live objects summaries)**.** A *live objects summary* of a method m is a tuple:

$$S_m = \langle \bar{p}_m, \mathbf{B}_m, \mathsf{MaxLive}_m, \mathsf{Esc}_m \rangle$$

3.4. Validity of live objects summaries

Now we define a key notion for our approach. Informally, a summary for a method m is *valid* if MaxLive is an over-approximation of the maximum number of fresh and live objects for all invocations of m and Esc over-approximates the number of fresh objects escaping m. To precisely define the notion of escaping using sub-heap descriptors we need to refine the notion of fresh for sub-heap descriptors.

Definition 3.7. Given an invocation run r of a method m, $\mathsf{fresh}_r \restriction SH$ is a restriction of fresh_r to a determined set of sub-heap descriptors.

$$\mathsf{fresh}_r \restriction SH = \mathsf{fresh}_r(0, |r| - 1) \cap [SH]_r^m, \text{ with } SH \subseteq \mathcal{SH}_m$$

Definition 3.8 (Validity of a summary)**.** $\langle \bar{p}, \mathbf{B}, \mathsf{MaxLive}, \mathsf{Esc} \rangle$ is a *valid* live objects summary for m if and only if the following formulas are always true:

$$\forall r \in R_m,\ \forall t, 0 \leq t < |r|, \forall \bar{p}\ \cdot\ \left(\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{MaxLive}\ \geq \max_{t' \leq t}\{|\mathsf{fresh}_r(t', t)|\}\right)$$

$$\forall SH \subseteq \mathcal{SH}_m,\ \forall r \in R_m, \forall \bar{p}\ \cdot\ \left(\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{Esc}(SH)\ \geq \left|\mathsf{fresh}_r \restriction SH\right|\right)$$

Notice that MaxLive and Esc are expressed in terms of the *summary* parameters $\bar{p}$, $\mathbf{B}$ is a predicate binding *method* parameters $\bar{\mathtt{fp}}$ with *summary* parameters $\bar{p}$. Then, the predicate $\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})]$ represents the substitution of *method* parameters with actual values from a concrete run r of a method m. Thus, in the previous formulas, MaxLive and Esc can be regarded as instantiated with the actual *method* arguments ($r_0(\bar{\mathtt{fp}})$) from each concrete run r of the method m.

4. Computing Live Objects Summaries

The technique for building summaries is essentially based on a technique that quantifies the number of visits to statements (e.g., allocations, method calls) when the iteration space is given as an invariant predicate that involves the formal parameter of the method under analysis [10].

Having these ideas in mind, the presentation of the inference mechanism will be based on a simplified intermediate representation of the program under analysis in which each method m is translated into a set of tuples, each made of a program location l in the body of m, the method invocation $\mu(\bar{\mathtt{v}})$ that appears in m at l, and an adequate invariant $\mathcal{I}$. The invariant in some sense over-approximates the set of all invocations at l that may be executed given the values of formal parameters of m.

Formally, a method is uniquely identified by its complete signature, denoted m. A method has a *name*, denoted $m.name$ or μ. The set of method names is denoted $\mathcal{M}$. In order to have a compact presentation, we assume there is a distinguished method named $new \in \mathcal{M}$. Methods different from *new* have a *body*, denoted $m.body$, defined by the following grammar:

$$\begin{array}{rcl} s & ::= & \langle \ell, \texttt{invoke}\ \mu(\bar{\mathtt{v}}), \mathcal{I} \rangle \\ S & ::= & \emptyset \mid s \mid S_1; S_2 \end{array}$$

where $\mu \in \mathcal{M}$, $\bar{\mathtt{v}}$ are the method invocation arguments, ℓ is a label and $\mathcal{I}$ is a predicate. We assume that labels are unique. Notice that the `new` statement is represented by `invoke` *new*.

Fig. 1 shows an example of how Program 1 can be encoded in the analysis representation. A Java-like program under analysis can be encoded in this simplified representation as follows: methods are kept with the same signature, the method body is codified as a set of invocations including the original program location and an invariant predicate binding the invocation arguments with the caller formal parameters and also describing the statement's *iteration space* [10]. Invariants are used to infer the number of traversals through that line as well as the values of the local variables used as arguments at each pass. Essentially, given a fixed value to the formal parameters of the caller, the number of integer solutions (i.e., the number of times the invariant is true) should be an upper bound of the number of times the statement is executed.

As mentioned, object allocations are also codified as invocations to a special method *new*. Array allocations could be considered as allocations of one object each or could take into consideration the size of the array. Since one of the applications of our analysis is to compare the resource usage of different implementations, counting the number of slots gives a better view of memory consumption. Therefore, array allocations are viewed as loops of `new` statements which are encoded using proper invariants that define an iteration space whose size is exactly the size of the array (more details in [10]).

Fig. 2 presents the equations defining MaxLive and Esc. The equations take a method m, the parameters $\bar{p}_m$, the binding predicate $\mathbf{B}$ (binding $\bar{p}_m$ with

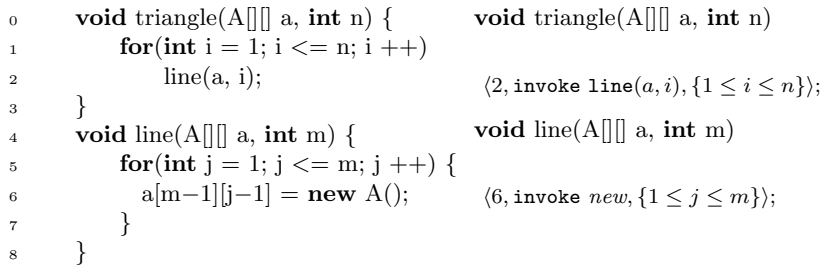

Figure 1: Translation of a Java method into the analysis representation

$\bar{\mathtt{fp}}_m$) for the summary to be built and a set of already computed summaries $\mathbb{S}$ and they compute $\mathsf{MaxLive}_m$ and Esc_m. They use a *Lifetime Information System* (**LIS**), with the properties specified in §3.2, to determine whether and how objects escape from a method. They also rely on a *symbolic calculator* (*SC*) to sum and maximize parametric expressions as described in §2 (see also §5.2 for implementation details).

We define the equations inductively on the structure of the intermediate representation. MaxLive defines the notion illustrated in §2.4. That is, MaxLive computes, among all invocations, the maximum number of fresh but non escaping objects (UB) and the sum of the escaping objects (SUM).

UB for the case of one statement (i.e., an invocation possibly within a loop), is an expression representing a bound of the maximum number of live and fresh objects (defined in DIFF) among all possible invocations to μ'. These invocations are described in the iteration space given by CTXT (recall that loops are abstracted away using the invariants). This maximization is computed using the operation *SC.UpBound*. UB joins the result for different statements using the least upper bound operator. Similarly, SUM computes the sum of the escaping objects (defined in FESC) over the whole iteration space, using *SC.Summate*. Several statements are just summed up (escaping objects are accumulative).

SC.UpBound and *SC.Summate* perform respectively maximization and sum of a parametric expression over an iteration space described by a predicate defined in terms of m parameters. The predicate required for these two operations is defined in CTXT by connecting the given invocation's iteration space $\mathcal{I}$ with the callee's invocation arguments $\bar{\mathtt{v}}$. It also binds the callee's formal parameters $\bar{\mathtt{fp}}_{m'}$ with the summary parameters $\bar{p}_{m'}$ using the predicate $\mathbf{B}_{m'}$.

Our analysis supports virtual calls relying on a call graph **CG** to resolve virtual method invocations. When it is not possible to determine statically a unique callee, the call is resolved to a set of potential candidate methods. Then, the analysis considers the summaries of all callee candidates and compute the maximum of all the possible results. Notice that each callee candidate might have its own set of sub-heap descriptors. For example, consider an invocation `a.m(p)` with two callee candidates `A.m(P p)` and `B.m(P p)`. In one candidate, an object may escape by the return value while in the other it may escape by

$$
\begin{aligned}
\mathsf{MaxLive}[m:S] &= \mathsf{UB}[m:S] + \mathsf{SUM}[m:S] \\
\mathsf{UB}[m:\emptyset] &= 0 \\
\mathsf{UB}[m:s] &= \mathit{SC.UpBound}(\mathsf{DIFF}[m:s], \mathsf{CTXT}[m:s], \bar{p}_m) \\
\mathsf{UB}[m:S_1;S_2] &= \mathsf{UB}[m:S_1] \sqcup \mathsf{UB}[m:S_2] \\
\mathsf{SUM}[m:\emptyset] &= 0 \\
\mathsf{SUM}[m:s] &= \mathit{SC.Summate}(\mathsf{FESC}[m:s], \mathsf{CTXT}[m:s], \bar{p}_m) \\
\mathsf{SUM}[m:S_1;S_2] &= \mathsf{SUM}[m:S_1] + \mathsf{SUM}[m:S_2] \\
\mathsf{CTXT}[m:\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I}\rangle] &= \mathbf{B}_m \wedge \mathcal{I} \wedge \bigwedge_{m' \in \mathbf{CG}.\mathit{call}(\ell,\mu')} \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'}/\bar{\mathtt{v}}] \\
\mathsf{DIFF}[m:\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I}\rangle] &= \bigsqcup_{m' \in \mathbf{CG}.\mathit{call}(\ell,\mu')} \mathbb{S}[m'].\mathsf{MaxLive} - \mathbb{S}[m'].\mathsf{Esc}(\mathbf{LIS}.\mathcal{SH}_{m'}) \\
\mathsf{FESC}[m:\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I}\rangle] &= \bigsqcup_{m' \in \mathbf{CG}.\mathit{call}(\ell,\mu')} \mathbb{S}[m'].\mathsf{Esc}(\mathbf{LIS}.\mathcal{SH}_{m'}) \\
\mathsf{Esc}[m:\emptyset] &= \lambda SH.\ 0 \\
\mathsf{Esc}[m:s] &= \lambda SH.\ \mathit{SC.Summate}(\mathsf{IESC}[m:s](SH), \mathsf{CTXT}[m:s], \bar{p}_m) \\
\mathsf{Esc}[m:S_1;S_2] &= \mathsf{Esc}[m:S_1] + \mathsf{Esc}[m:S_2] \\
\mathsf{IESC}[m:\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I}\rangle] &= \lambda SH.\ \bigsqcup_{m' \in \mathbf{CG}.\mathit{call}(\ell,\mu')} \mathbb{S}[m'].\mathsf{Esc}(\mathbf{LIS}.\mathit{esc}_{(\ell,m')}(SH)) \\
\mathbb{S}[\mathit{new}].\mathsf{MaxLive} &= 1, \mathbb{S}[\mathit{new}].\mathsf{Esc} = \lambda SH.\ SH \neq \emptyset ?\ 1\ :\ 0 \\
\mathbf{B}_{\mathit{new}} &= \{\}, \bar{p}_{\mathit{new}} = []
\end{aligned}
$$

Figure 2: Equations for inferring Live Object summaries

the parameter p. So, **LIS**.*esc* must depend both on the line and on the actual method invoked.

DIFF, FESC and IESC resolve dynamic method invocations. Given a virtual call they compute aggregated information about all potential callees, producing one result representing an over approximation of all of them. Note they take the information from summaries of callees.

DIFF is the difference $\mathsf{MaxLive}_l - \mathsf{Esc}_l$ but considering that there can be more than one callee and, thus, taking the maximum among them. Note that, since $\mathbf{LIS}.\mathcal{SH}_{m'}$ is the set of all sub-heap descriptors for m', $\mathsf{Esc}[m'](\mathbf{LIS}.\mathcal{SH}_{m'})$ represents all objects escaping m'. FESC computes the maximum possible amount of escaping objects for all the candidate callees.

As we have mentioned in §2.7, there are various ways for an object to escape: it could be returned or pointed to in a field of one of the parameters, or both via aliasing, etc. Depending on the way an object escapes, it might or might not be captured in the calling method. Equations use operation **LIS**.*esc* that, given an invocation, binds caller sub-heap descriptors to callee sub-heap descriptors.

That is, it indicates which objects escaping the caller come from the callee invoked at that program point.

The computation for Esc is similar to the one for SUM but does not sum all objects escaping the callee. Instead, it only accumulates the objects escaping the callee by the given sub-heap descriptors. IESC quantifies this number. Given a set of caller sub-heap descriptors, it retrieves the summaries of all potential callees, finds out the corresponding callee sub-heap descriptors, and finally computes the maximum over these callees.

As the base case, there is a summary for `new` statements: $\mathsf{MaxLive}_{new} = 1$ and $\mathsf{Esc}_{new} = 1$. The intuition is that `new` statements yield the same memory consumption as a method that would allocate an object and that would let this object escape to the caller scope. However, we consider the special case for $\mathsf{Esc}(\emptyset)$ which represents a query for objects that are captured in the method doing the `new`. This condition allows us to precisely capture the case when an object does not escape the caller method (i.e., when $\mathbf{LIS}.esc_{(\ell,\texttt{new})}(SH) = \emptyset$).

Example. Let's apply the equations to the method `line` introduced in Fig. 1. It has only one statement: $s = \langle 6, \texttt{invoke}\ new, \{1 \le j \le m\}\rangle$. We use $\bar{p}_{\texttt{line}} = m$ and $\mathbf{B}_{\texttt{line}} = \{\mathbf{m} = m\}$.

$$
\begin{aligned}
&\mathsf{DIFF}[\texttt{line} : \langle 6, \texttt{invoke}\ new, \{1 \le j \le m\}\rangle] = \\
&\quad = \bigsqcup_{m' \in \{\texttt{new}\}} \mathbb{S}[m'].\mathsf{MaxLive} - \mathbb{S}[m'].\mathsf{Esc}(\mathbf{LIS}.\mathcal{SH}_{m'}) \\
&\quad = \mathbb{S}[\texttt{new}].\mathsf{MaxLive} - \mathbb{S}[\texttt{new}].\mathsf{Esc}(\mathbf{LIS}.\mathcal{SH}_{\texttt{new}}) = 1 - 1 = 0
\end{aligned}
$$

$$
\mathsf{FESC}[\texttt{line} : \langle 6, \texttt{invoke}\ new, \{1 \le j \le m\}\rangle] = \mathbb{S}[\texttt{new}].\mathsf{Esc}(\mathbf{LIS}.\mathcal{SH}_{\texttt{new}}) = 1
$$

$$
\begin{aligned}
&\mathsf{CTXT}[\texttt{line} : \langle 6, \texttt{invoke}\ new, \{1 \le j \le m\}\rangle] \\
&\quad = \mathbf{B}_{\texttt{line}} \wedge \{1 \le j \le m\} \wedge \mathbf{B}_{\texttt{new}}[] = \{\mathbf{m} = m \wedge 1 \le j \le m\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{MaxLive}[\texttt{line} : \{s\}] = \mathsf{UB}[\texttt{line} : \{s\}] + \mathsf{SUM}[\texttt{line} : \{s\}] \\
&\quad = \mathsf{UB}[\texttt{line} : s] + \mathsf{SUM}[\texttt{line} : s] \\
&\quad = \big(SC.UpBound(\mathsf{DIFF}[\texttt{line} : s], \mathsf{CTXT}[\texttt{line} : s], \bar{p}_{\texttt{line}})\big) \\
&\qquad + SC.Summate(\mathsf{FESC}[\texttt{line} : s], \mathsf{CTXT}[\texttt{line} : s], \bar{p}_{\texttt{line}}) \\
&\quad = SC.UpBound(0, \{\mathbf{m} = m \wedge 1 \le j \le m\}, m) \\
&\qquad + SC.Summate(1, \{\mathbf{m} = m \wedge 1 \le j \le m\}, m) \\
&\quad = 0 + \mathbf{m} = \mathbf{m}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{IESC}[\texttt{line} : \langle 6, \texttt{invoke}\ new, \{1 \le j \le m\}\rangle] = \\
&\quad = \lambda SH. \bigsqcup_{m' \in \{\texttt{new}\}} \mathbb{S}[m'].\mathsf{Esc}(\mathbf{LIS}.esc_{(6,m')}(SH)) \\
&\quad = \lambda SH.\ \mathbb{S}[\texttt{new}].\mathsf{Esc}(\mathbf{LIS}.esc_{(6,\texttt{new})}(SH))
\end{aligned}
$$

$$\mathsf{Esc}[\texttt{line} : \{s\}] = \mathsf{Esc}[\texttt{line} : s] =$$
$$= \lambda SH.\ SC.Summate(\mathsf{IESC}[\texttt{line} : s](SH), \mathsf{CTXT}[\texttt{line} : s], \bar{p}_{\texttt{line}})$$

If the sub-heap descriptor SH includes the content of the array `a`, this formula becomes:

$$SC.Summate(1, \{\mathbf{m} = m \wedge 1 \leq j \leq m\}, m) = \mathbf{m}$$

Otherwise, it will simply be 0.

4.1. Correctness

Our technique builds methods summaries in a compositional fashion, relying on the existence of summaries for their callees. Thus, the obtained summaries will be valid if the provided summaries for the callees are also valid.

Let $Invoked(S)$ be the set of methods invoked in S :

$$\begin{aligned} Invoked(\emptyset) &= \emptyset \\ Invoked(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle) &= \mathbf{CG}.call(\ell, \mu') \\ Invoked(S_1; S_2) &= Invoked(S_1) \cup Invoked(S_2) \end{aligned}$$

Proposition 1. For every method m, if for all $m' \in Invoked(m.body)$, $\mathsf{Esc}[m']$ is valid, then $\mathsf{Esc}[m]$ is valid.

Proposition 2. For every method m, if for all $m' \in Invoked(m.body)$, $\mathsf{Esc}[m']$ and $\mathbb{S}[m'].\mathsf{MaxLive}$ are valid, then $\mathsf{MaxLive}[m]$ is valid.

Proof. See §Appendix A. The validity of the summary follows by induction on the structure of the equations and the hypothesis about the validity of the summaries. It relies on the properties of the **LIS** and *SC*. One important aspect of the proof is an observation about the fact, that given adequate invariants, the operations of the *SC* perform over-approximation of maximums and sums over invocation runs.

Observation 1. Given a method m and $\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle \in m.body$ and $m' \in Invoked(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle)$, such that $\mathcal{I}$ is a valid iteration space, then for every invocation run $r \in R_m$ and for all $\bar{p}_m$, we have:

$$\mathbf{B}_m[\bar{\mathtt{fp}}_m / r_0(\bar{\mathtt{fp}}_m)] \implies SC.UpBound(E, \mathcal{I}, \bar{p}_m) \geq \max_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'} / r'_0(\bar{\mathtt{fp}}_{m'})]} E$$

$$\mathbf{B}_m[\bar{\mathtt{fp}}_m / r_0(\bar{\mathtt{fp}}_m)] \implies SC.Summate(E, \mathcal{I}, \bar{p}_m) \geq \sum_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'} / r'_0(\bar{\mathtt{fp}}_{m'})]} E$$

where $vars(E) \in \bar{p_{m'}}$.

5. Implementation notes

The concrete algorithm closely follows the equations presented in §4. Here we present instantiations of our compositional algorithm to compute valid summaries. We present the implementations of two possible instances of the lifetime oracle **LIS**. Then, we discuss some technical aspects of the symbolic calculator *SC*. We also discuss some methods used to infer the invariants used to describe iteration spaces. Finally, we discuss some technical limitations of the technique and the underlying tools and some possible solutions to overcome them.

*5.1. Object lifetime analysis (**LIS**)*

The first implementation is based on an interference analysis [35] and the second is a more precise one based on an adaptation of Salcianu-Rinard points-to and escape analysis [36, 7].

Recall that *esc* maps sets of caller sub-heap descriptors to sets of callee sub-heap descriptors: to a set of caller sub-heap descriptors SH_m, it associates the set of callee sub-heap descriptors the content of which might escape by a sub-heap of SH_m. As in §3, we treat the `new` and `newarray` instructions just as simple methods that return the newly-allocated objects.

5.1.1. Escape analysis based on object interference

In a nutshell, the analysis presented in [35] performs an interference analysis which computes for each method a collection of equivalence classes, each class representing a set of potentially connected objects. The relation $\sim$ is the smallest equivalence relation over program variables such that one variable interferes with another when they appear in the same assignment (i.e. $v \sim u$ if $v = u$, $v = u.f$ or $v.f = u$). The analysis is performed on a Static Single Assignment (SSA) representation of the program [18]. Thus, every variable is set only once, at its definition site. So the $\sim$ relation is extended to labels: $l \sim v$ when $v =$ `new C(...)` at label l. Note that, due to SSA, we could directly use the variable v to refer to the allocation, but we include the label just to make allocations more explicit. The effect of a method call is similar to an assignment between the arguments in the caller and the parameters in the callee. This has also the effect of forcing the equivalence between the variable the callee returns and the variable of the caller to which it is assigned.

Any element of an equivalence class can be used as sub-heap descriptor for that class. We use method parameters and the return value as they are visible outside the method boundary.

Let us consider a method m which calls at line l the method m'. Given the relation $\sim$ we define:

$$\mathbf{LIS}.esc_{l,m'}(SH_m) = \{sh' \mid sh' \in \mathcal{SH}_{m'}, \exists sh \in SH_m, sh' \sim sh\}$$

Notice that if an object allocated in the callee m' escapes, it will be equivalent to either some parameter or the return value of m', so that it will be in $\mathcal{SH}_{m'}$.

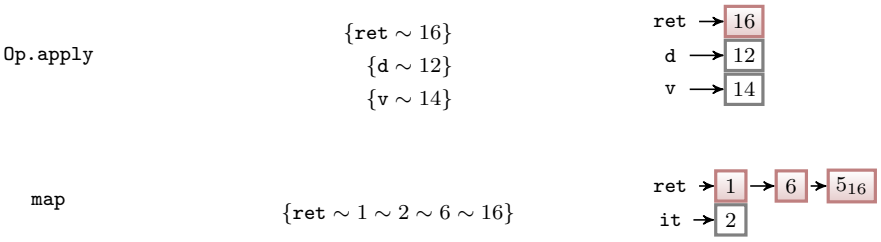

Figure 3: Comparing the two implementations of **LIS** (left: interference-based; right: points-to-based).

5.1.2. Escape analysis based on points-to graph

The second technique is based on [36, 39, 7], slightly modified to make it context sensitive. It builds a points-to graph (PTG) for each method, providing an over-approximation of the heap visible at its exit point. A PTG is a graph (N,E) where each node n_l represents the set of objects created at location l and each edge e represents accesses to objects (i.e. $v = u.f$ or $v.f = u$). For method calls the analysis transfers the nodes escaping the callee to the caller. Those nodes are re-labeled to denote the calling context that brings them to the caller. Following the notation in [39], we write $\mu_{m',l,m}$ for the map from the callee PTG to the caller PTG (when m calls at some method m' at line l) and $n' \xrightarrow{\mu_{m',l,m}} n$ to indicate that the node n' (of the callee PTG) is mapped to the node n (of the caller PTG) by μ. Obviously, this means in particular that the node n' escapes from the callee.

Nodes are partitioned into inside nodes, representing objects created by the method under analysis m or its callees, or outside nodes, representing pre-existent objects. We use inside nodes as sub-heap descriptors which offer a very fine granularity to summaries having a sub-heap descriptor for each allocation site. Only inside nodes are required since we are interested in quantifying just the objects created by m (or its callees).

Within this setting, and writing $\mu_{m',l,m}$ for the map of the call to m' at line l in m, we can simply define $\mathbf{LIS}.esc_{l,m}$ as such:

$$\mathbf{LIS}.esc_{l,m}(SH_m) = \{n' \mid n' \in \mathcal{SH}_{m'}, \exists n \in SH_m, n' \xrightarrow{\mu_{m',l,m}} n\}$$

5.1.3. Comparison of the two escape analyses

To illustrate the difference of these two analyses, Figure 3 compares their outcome for methods `map` and `Op.apply` in Program 2 from §2. On the left side is the result of the interference-based analysis and on the right side of the points-to-graph-based analysis (colored are escaping the method). For `Op.apply` both analyses provide a similar result. In contrast, for `map` the interference analysis includes the iterator (from line 2) in the same equivalence class as the list, making it escape while the other recognizes it as captured. In addition, the PTG produces 3 different escaping sub-heap descriptors enabling a client

method to analyze every object in isolation. Notice also how the PTG keeps tracks of calling contexts. The node 5_{16} represents the allocation originated at line 16 of `apply` transferred to method `map` after the call to `apply` at line 5.

5.1.4. Implementing Esc

The function Esc receives a set of sub-heap descriptors and yields the number of fresh objects escaping through these sub-heaps. When implementing Esc we decided to define it only for singleton sets of sub-heap descriptors. Nevertheless, the equations still ask for non-trivial sub-heap sets when querying for Esc of callee summaries. This is useful for hand-specified live objects summaries: it thus allows the programmer to provide a finer Esc for some sub-heap descriptor. When the result specifically required is not readily available, it is always possible to compute a safe over-approximation by decomposing the set of sub-heap descriptors into singleton sets:

$$\mathsf{Esc}_m(SH) \leq \sum_{sh \in SH} \mathsf{Esc}_m(\{sh\})$$

The over-approximation comes from the fact that the method m could generate aliasing and thus let escape one object via various sub-heaps, as it was illustrated in §2.7.

5.2. Symbolic calculator (SC)

We implemented the set of operations required by the algorithm, namely *SC.UpBound*, *SC.Summate*, $+^{SC}$ and $\sqcup^{SC}$ with the aid of the symbolic calculator ISCC [38]. This tool manipulates affine integer sets and relations, providing functionality to count the number of elements of those sets as well as performing maximization and sum of polynomials within affine domains.

An affine integer set or domain is a subset of $\mathbb{Z}^d$ defined as a set of integer vectors constrained by affine inequalities on the indices, on the parameters, and on integer-valued existentially-quantified variables. They can be used to describe iteration spaces. The interest for using iteration spaces is the fact that the number of integer solutions of an iteration space for a program point (e.g, a loop invariant) describes the number of possible valuations of the variables at that location. If the variables appearing in the space are *inductive*[7], then this number of valuations over-approximates the number of visits to the statement [10].

For instance, consider Program 1 and the call at line 2. An iteration space or domain describing the possible variable valuations at this point can be described in ISCC syntax as:

```
IS := [n] -> { [i] : 0 < i <= n }
```

[7]By inductive variable we mean a variable that affects the numbers of times a statement is visited

In a nutshell, the square brackets are used in this syntax to introduce variables. The variable `n`, introduced outside the braces by `[n]`, is a parameter of the iteration space. The variable `i`, introduced inside the braces, is bound in the definition; thus the syntax recalls the standard notation for sets: $\{i \mid 0 \leq i \leq n\}$.

5.2.1. SC.Summate

The operation *SC.Summate*$(e, Inv, \bar{p})$ is expressed in ISCC syntax with the command `sum`:

$$\texttt{sum ([}\bar{p}\texttt{] -> \{[}\bar{v}\texttt{] -> e : } \mathit{Inv}\texttt{\});}$$

where all the variables appearing in `e` and *Inv* that are not among the parameters $\bar{p}$ must be bound in $\bar{v}$.

As an example, the number of objects allocated in Program 1, line 6 in §2 can be expressed as:

```
sum( [m] -> { [j] -> 1 : 1 <= j <= m } );
```

This syntax is slightly more complex than the one presented in the previous section. It allows us to associate a polynomial to each point of the domain $\{j \mid 1 \leq j \leq m\}$ (parametric in m), and to sum those polynomials. In this case, the polynomial is just the constant 1, meaning we are performing the simple sum $\sum_{1 \leq j \leq m} 1$ thus yielding the result m whenever $m \geq 1$ (and 0 otherwise):

```
[m] -> { m : m >= 1 }
```

5.2.2. SC.UpBound

The operation *SC.UpBound*$(e, Inv, \bar{p})$ is expressed in ISCC using the command `ub` as follows:

$$\texttt{ub ([}\bar{p}\texttt{] -> \{[}\bar{v}\texttt{] -> e : } \mathit{Inv}\texttt{\});}$$

Note that, when `e` is not linear (when it is any polynomial, for instance) this operation needs to solve symbolically a non-linear maximization problem. The calculator is able to solve this problem using a Bernstein based technique, as explained in [15].

In the example in Program 4 when `test` calls `safeMap`, we can use ISCC to compute:

```
ub ([ls_size, maxSize] ->
         { [i, list_size] -> list_size + 4
         : 1 <= i <= ls_size and list_size = maxSize });
```

which returns:

```
[ls_size, maxSize] -> { max((4 + maxSize)) : ls_size >= 1 }
```

5.2.3. $+^{SC}$ and $\sqcup^{SC}$

The last operations required to manipulate parametric expressions are $+^{SC}$ and $\sqcup^{SC}$. Naturally $+^{SC}$ sums two expressions. Given two parametric expressions with the same parameters, $\sqcup^{SC}$ yields a new one which is the smallest expression (possibly defined by cases) greater than the two.

ISCC provides the operators `+` for $+^{SC}$ and `.` which tries to solve $\sqcup^{SC}$ (see limitations in §5.2.4).

5.2.4. Limitations

ISCC requires iteration spaces and domains to be sets of linear constraints and operates with polynomials over these domains. This implies that programs invariants (generated or manually provided) must be linear while the inferred expressions will be polynomials[8].

Another limitation is that in certain circumstances ISCC is not able to compute the least upper bound. This complicates the computation of the escaping part of the summary in virtual calls inside loops, which involves summing least upper bounds between the escaping objects of the callee candidates (see §2.5), because ISCC does not support the application of `sum` over unsolved least upper bounds.

To overcome this (incidental) problem, our implementation either over-approximates $a \sqcup^{SC} b$ somehow (several strategies are available) or delays the resolution of $\sqcup^{SC}$ until more context is available. The former impacts precision, the latter impacts performance because it maintains a set of unsolved operations.

Notice that in case of methods with constant allocation (e.g., `new` statements) we can compute the number of objects by simply *counting* the number of visits to the method invocation. To handle these particular cases, we can then rely on techniques that are specifically suited to them, such as techniques that use program invariants to deduce the visits ([10] or [3]) or techniques based on code templates ([22]).

5.3. Computing invariants

In order to reduce the annotation burden we rely on external tools in order to obtain loop invariants. As explained in the previous section, the loop invariants we look for are linear, in order for the symbolic calculator to be able to handle them.

We have been using mainly Daikon [19] which is a tool that is able to obtain different kinds of likely invariants. Of course, we had to limit Daikon to the discovery of linear invariants as we mentioned. Linear invariants also have the advantage to be easier to check automatically using tools like ESC/JAVA [34] or other tools relying on SMT solvers.

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

[8] Some cases of exponential consumption might still be expressed by performing tricks such as variable substitution.

and path expressions that have to appear in the invariant (length of arrays and strings, size of collections, instance and static fields, etc, see [20]) and the set of inductive variables. The tool also tries to obtain invariants for common iterations patterns such as iterators.

5.4. Dealing with interfaces

Our algorithm can automatically deal with virtual calls, by computing the supremum of all summaries coming from each potential callee. Using a similar approach we can infer summaries for interface methods. Basically, for each method of the interface we analyze its implementations in all the classes that implement the interface and compute the supremum of those summaries. If the developer implements a new class for the interface we can just perform the supremum between the interface and the new class.

The tool, in addition to the inference mechanism, allows the specification of summaries for concrete methods, native and abstract/interface methods. In case of coexisting summaries for concrete methods and their interfaces, the user can choose to use either the interface summary or the more concrete one.

5.5. Dealing with recursive programs

The implementation works straightforwardly when summaries are computed bottom-up. That assumes a non-cyclic structure of summary dependencies. Thus our current implementation does not support directly memory-allocating recursive methods (or those making use of reflection). For those cases, a consumption summary has to be provided by the user (either by hand or with the help of another tool) for a strongly connected component.

Nevertheless, the approach can, potentially, deal with recursions since the symbolic application of equations naturally yields recurrence relations between summaries of recursive methods. Those are well-defined and have solutions provided the summary parameters decrease in a well-founded order. Major issues to automatically compute such summaries are of practical nature and, in several cases, recursive method summaries can be actually inferred and modularly used when the actual consumption can be described using piecewise polynomials.

One technical problem is that recurrence relations built from the equations feature operations like *Summate* and $\sqcup$ which are not straightforwardly treated by recurrence equation solvers. While $\sqcup$ can be replaced many times by max operator, *Summate*, which is instrumental for the precise computation in presence of iteration spaces, cannot be processed by recurrence solvers like Mathematica [40]. Fortunately, the issue arises only when recursive calls are performed inside loops. Notice that other approaches like COSTA do not cope with this issue because they translate loops into recursions (but probably at a cost in terms of precision as shown in Program 1 in §2).

Note also that the computation of Esc can be done by solving recurrence relations that do not involve MaxLive. In turn, MaxLive equations can be rewritten by inlining the closed expressions that solve Esc. This typically eases the work to be done by recurrence solvers.

The following example illustrates those ideas. Of course, more work should be done to implement a general treatment of recursion including mutually recursive methods for which the corresponding recurrence relations seem to be challenging for solvers.

```
0 void m(int i) {
1     if (i <= 0) return;
2     o = new Object();
3     this.addToList(o);
4     this.m(i−1);
5 }
```

Program 6: A simple recursive method

Consider the method `m` in Program 6. For simplicity, we present the formulation using the formulas like the ones presented in §2, but the same idea applies using the equations presented in §4. Let us assume the following summary for `addToList`:

$$\begin{aligned}\mathsf{MaxLive}_{\texttt{addToList}} &= 2\\ \mathsf{Esc}_{\texttt{addToList}} &= 1\end{aligned}$$

To compute the summary for `m` we can leverage the fact that this program has no loops. That means that the symbolic operations for operating over iteration spaces, namely *Summate* and *Maximize*, can be trivially simplified.

$$\begin{aligned}\mathsf{Esc}_2(i) &= \mathit{Summate}(\mathsf{Esc}_{\texttt{new}}, \{\}) = 1\\ \mathsf{Esc}_3(i) &= \mathit{Summate}(\mathsf{Esc}_{\texttt{addToList}}, \{\}) = 1\\ \mathsf{Esc}_4(i) &= \mathit{Summate}(\mathsf{Esc}_{\texttt{m}}(i-1), \{i' = i-1, i > 0\}) = \mathsf{Esc}_{\texttt{m}}(i-1)\\ \mathsf{Esc}_{\texttt{m}}(i) &= \sum_{l \in \{2,3,4\}} \mathsf{Esc}_l(i) = 1 + 1 + \mathsf{Esc}_{\texttt{m}}(i-1) = 2 + \mathsf{Esc}_{\texttt{m}}(i-1)\end{aligned}$$

Using the base case, i.e. $\mathsf{Esc}_{\texttt{m}}(0) = 0$, this recurrence equation could be solved as $\mathsf{Esc}_{\texttt{m}}(i) = 2i$. Now, we can compute $\mathsf{MaxLive}_{\texttt{m}}$ using $\mathsf{Esc}_{\texttt{m}}$. In the following formulas we directly remove *Maximize* and *Summate* for the sake of succinctness. Let us assume $i \geq 1$. We can replace $\mathsf{Esc}_{\texttt{m}}(i)$ by $2i$.

$$
\begin{aligned}
\mathsf{MaxLive}_2(i) &= (\mathsf{MaxLive}_{\mathtt{new}} - \mathsf{Esc}_{\mathtt{new}}) + \mathsf{Esc}_{\mathtt{new}} = \mathsf{MaxLive}_{\mathtt{new}} = 1 \\
\mathsf{MaxLive}_3(i) &= (\mathsf{MaxLive}_{\mathtt{addToList}} - \mathsf{Esc}_{\mathtt{addToList}}) + \mathsf{Esc}_{\mathtt{addToList}} \\
&= \mathsf{MaxLive}_{\mathtt{addToList}} = 2 \\
\mathsf{MaxLive}_4(i) &= (\mathsf{MaxLive}_{\mathtt{m}}(i-1) - \mathsf{Esc}_{\mathtt{m}}(i-1)) + \mathsf{Esc}_{\mathtt{m}}(i-1) = \mathsf{MaxLive}_{\mathtt{m}}(i-1) \\
\mathsf{MaxLive}_{\mathtt{m}}(i) &= \bigsqcup_{l \in \{2,3,4\}} (\mathsf{MaxLive}_l(i) - \mathsf{Esc}_l(i)) + \sum_{l \in \{2,3,4\}} \mathsf{Esc}_l(i) \\
&= \sqcup\{1-1, 2-1, \mathsf{MaxLive}_{\mathtt{m}}(i-1) - \mathsf{Esc}_{\mathtt{m}}(i-1)\} \\
&\quad + 1 + 1 + \mathsf{Esc}_{\mathtt{m}}(i-1) \\
&= \sqcup\{0, 1, \mathsf{MaxLive}_{\mathtt{m}}(i-1) - 2(i-1)\} + 1 + 1 + 2(i-1) \\
&= \sqcup\{1, \mathsf{MaxLive}_{\mathtt{m}}(i-1) - 2(i-1)\} + 2i
\end{aligned}
$$

Note that $\mathsf{MaxLive}_{\mathtt{m}}(1) = \sqcup\{1, 0\} + 2 = 3$ meaning that $\mathsf{MaxLive}_{\mathtt{m}}(i) > 1$ for all $i \geq 1$. Thus, we can remove the $\sqcup$ leading to a simpler recurrence equation.

$$\mathsf{MaxLive}_{\mathtt{m}}(i) = \mathsf{MaxLive}_{\mathtt{m}}(i-1) - 2(i-1) + 2i = \mathsf{MaxLive}_{\mathtt{m}}(i-1) + 2$$

This recurrence equation could be solved as $\mathsf{MaxLive}_{\mathtt{m}}(i) = 2i + 1$.

6. Empirical evaluation

We are interested in answering the following research questions:

1. How does our compositional technique perform with respect to non-compositional techniques in terms of precision?
2. Is our technique capable of dealing with medium-size applications, with a moderate level of human intervention and yet produce tight bounds?

To answer these questions we developed a prototype tool implementing the technique presented in §4 and the two escape analyses presented in §5.1.1 and §5.1.2 (denoted resp. *Madeja* and *Rinard*).

The tool computes an over-approximation of the number of live objects of a given program. Unfortunately, there are objects that our current implementation cannot detect because they are objects created internally by the VM or by non-analyzable methods for which we had not provided a summary. Thus, in order to perform a fair analysis of the precision of the obtained bounds, we implemented a runtime monitor, using the Java Path Finder VM[9] [24], that measures the actual number of live objects in a run. Essentially, the monitor performs a continuous garbage collection reclaiming objects as soon as they become unreachable. The monitor is able to distinguish objects that were visible to our analysis tool from those that were not. To do so, the program is previously instrumented in order to tag as analyzable each object that can be seen by the compositional analysis.

[9] We also implemented our own version using JVMTI obtaining similar results.

benchmark	MaxLive			ΔEsc
	Rinard	*Madeja*	Non-comp.	
`running.copy(l)`	$l.size + 2$	$l.size + 2$	$l.size + 2$	-1
`safeMap(L)`	$5 + 6L$	$6 + 6L$	$5 + 7L$	$-(2 + L)$
`test(M,L)`	$6 + M + (2 + 5M)L$	$(5 + 6M)L + 4$	$6 + M + (2 + 6M)L$	$-(M + 3)$
`em3d.create(n,d)`	$8 + (4 + 6d)n$	$8 + (4 + 6d)n$	$8 + (4 + 6d)n$	$-(6 + 2n)$
`compute`	2	2	2	-2
`main(d,n,i)`	$6dn + 4n + 15$	$6dn + 2i + 4n + 12$	$6dn + 4n + 14$	0
`mst.addEdges(v)`	$2v^2$	$2v^2$	$2v^2$	0
`compute(v)`	v	v	v	0
`main(v)`	$\frac{9}{4}v^2 + 4v + 6$	$\frac{9}{4}v^2 + 4v + 6$	-	-
`biSort.inOrder`	1	1	1	0
`main(s)`	$6 + s$	$6 + s$	$4 + s$	0

Table 1: Simple benchmark (L = *list_size*, M = *maxSize*, S = *ls_size*).

6.1. Existing benchmarks

To answer the first question we compared our tool with our previous non-compositional one ([11]). For that we considered the non-recursive JOLDEN benchmarks [13]. We used these simple programs to measure the precision of the technique and to study the trade-offs between performance and precision using the escape analyses. Table 1 summarizes the analysis of our running example and three benchmarks from JOLDEN. It shows the inferred MaxLive using the compositional analyses with *Madeja* and *Rinard* oracles, and the MaxLive obtained with the non-compositional algorithm of [11]. We also exhibit the difference between the total Esc of the two escape analysis algorithms in order to show how they affect the overall precision. For these small examples the analysis took around 20 seconds using *Rinard* ($\sim 25\%$ less using *Madeja*).

In general, the bounds for `mst`, `em3d` and `bisort` are similar to [11]. The most significant difference is originated in method `compute` from `em3d`. In this case the obtained value of MaxLive is identical but the *Rinard* is able to collect more elements, producing no escaping consumption, and thus, obtaining a more accurate result than *Madeja*. It is very interesting to notice that although the difference on the number of escaping objects seems negligible, namely 2 objects more, the effect on the MaxLive of `main` is extremely relevant because `compute` is invoked inside a loop. Although `bisort` contains recursive fragments, we could analyze it by providing a summary for `inOrder` which features a recursive pattern not supported by the tool. In this case also we obtained a very similar bound. In the case of `test`, both instances of the compositional analysis work better than the non-compositional one, by giving tighter MaxLive for `safeMap`. The impact on MaxLive of `test` needs some explanation. Using *Rinard*, MaxLive of `test` is always smaller than with [11]. With *Madeja*, the compositional analysis turns out to be better than the non-compositional one when the maximum length of the sub-lists is larger than the length of the list, more precisely when $M > 3L - 2$. This result is explained by the fact that the non-compositional analysis of [11] sums up the consumption of all potential callees of a virtual call, while the compositional one computes the maximum.

With this benchmark we mainly wanted to see if the compositional approach affected the overall precision of the bounds, comparing it to a baseline non-compositional analysis. Since the obtained bounds are quite similar to [11] we did not find necessary to show a comparison between the bounds and the actual allocation using the run-time monitor. We will do that in the next section for our largest case study.

6.2. JLayer

To validate the scalability of our approach we analyzed a medium-size application which uses polymorphism and reflection. JLayer[10] is an open source library that decodes/plays/converts MPEG 1/2/2.5 Layer 1/2/3 in real time for the Java platform. It is a medium-size application which involves about 45 classes, 398 methods, 25k lines of code, and 3k `new` statements. The compositionality of our technique is central for such an application: the call stack for the examples analyzed in [11, 21] never has more than 8 calls, while JLayer requires approximately 35.

The main component is an MPEG Audio decoder (`Decoder`) which supports three different layers. The `Player` uses the `Decoder` services to reproduce audio files. `Player` includes the method `decodeFrame` in charge of reading frames, using a `SampleBuffer`, and of reproducing it by writing it to an `AudioDevice`. The method `play` basically performs a loop that decodes and reproduces every frame. To decode a frame it creates specific instances of `FrameDecoder` depending on the type of the frame being analyzed.

This benchmark is interesting since it brings in several challenges that are representative of Java programs and, therefore, must be handled by any analysis tool. We started by running our analysis in a fully automated way. Not surprisingly, we obtained some overly conservative approximations. To get more accurate bounds we had to resort to slight but focused user intervention by improving method-local information. We show in the next sections some interesting cases we had to deal with.

6.2.1. Modeling lazy initialization

```
0  public boolean play(int frames) {
1    ...
2    for(i=0;i < frames && ret;i++){
3      ret = decodeFrame();
4      ...
5    }
6  }
```

```
7  protected boolean decodeFrame() {
8    ...
9    if (!this.decoder.initialized)
10     this.decoder.initialize(h);
11     ...
12     Header h=bitstream.readFrame();
13 }
```

Program 7: Fragments of `decodeFrame` and `play` of `Player`

[10] `http://www.javazoom.net/javalayer/javalayer.html`

Consider the code snippet in Program 7. Method `decodeFrame` is responsible for decoding frames. The first call to `decodeFrame` initializes the `decoder` (line 10). This generates an *escaping* sub-heap because the lifetime of the created object exceeds that of `decodeFrame`. Our tool infers the size of these sub-heaps as:

$$\mathsf{Esc}_{\texttt{initialize}}(\{\texttt{this}\}) = (m = 3)?4016 : 5136$$
$$\mathsf{Esc}_{\texttt{initialize}}(\{\mathit{static}\}) = 544$$

where m represents the frame mode and where we use for its compactness the Java expression notation $b?x : y$ to stand for x when b is true and y when it is false. Since `decodeFrame` is called at each iteration of the `play` loop (line 3) the escaping objects gets accumulated, yielding:

$$\mathsf{Esc}_{\texttt{play}}(\{\texttt{this.decoder}\}) = ((m = 3)?4016 : 5136) \times \mathit{frames}$$

Clearly, this upper bound is too conservative. To improve it, we need to know whether `decoder` is initialized. This can be done by explaining to the tool that `this.decoder.initialized` should be a summary parameter of `decodeFrame` and make sure the invariant at line 10 specifies that

$$\texttt{this.decoder.initialized} = \mathit{false}$$

We write *initialized* for `this.decoder.initialized`. Once it is included as a summary parameter, our technique manages to take advantage of this domain constraint and returns the following bound:

$$\mathsf{Esc}_{\texttt{decoder}}(\{\texttt{this}\}) = (\mathit{not\ initialized})?[(m = 3)?4016 : 5136]:0$$

This new summary parameter must be appropriately bound at line 3 of `play` to reflect the current initialization state. That can be achieved by correlating *initialized* with the index loop variable (i.e. $\mathit{initialized} \iff (i \neq 0)$). Then, the computed accumulated escaping part of the summary is:

$$\mathsf{Esc}_{\texttt{play}}(\{\texttt{this.decoder}\}) = (m = 3)?4016 : 5136.$$

This *lazy initialization* pattern is quite common in Java programs. For instance, the Singleton pattern is usually implemented following a lazy initialization approach where the singleton instance is initialized the first time it is required.

6.2.2. Dealing with unanalyzable methods

Program 8 shows part of `createAudioDeviceImpl`. It uses reflection to create an instance of `JavaSoundAudioDevice`. Since there is just one available implementation of that class, we know which instance should be created, but the analysis is unable to figure this out automatically, considering this method as not analyzable. To overcome this limitation we provide summaries for such methods. The amount of memory required to create a new instance of `JavaSoundAudioDevice` is that of its constructor which is indeed analyzable.

```
0 protected JavaSoundAudioDevice createAudioDeviceImpl() {
1   ClassLoader loader = getClass().getClassLoader();
2   try { return (JavaSoundAudioDevice)instantiate(loader,
3       DEVICE_CLASS_NAME);
4   } catch (Exception ex) ...
```

Program 8: `createAudioDevice` implementation.

Class	Method	MaxLive
LIDecoder	decodeFrame	$34 + ns$
LIIDecoder	decodeFrame	$34 + 10ns + [7ns, 7i, 7ns, 0]_m$
LIIIDecoder	decodeFrame	0

Table 2: Summaries for the possible instances of `FrameDecoder`

This yields a total of 4096 escaping through `this`. Therefore we can provide the following summary:

$$\begin{aligned}\mathsf{Esc}_{\texttt{createAudioDeviceImpl}}(\{ret\}) &= \mathsf{Esc}_{\texttt{initialize}}(\{\texttt{this}\}) \\ &= 4096\end{aligned}$$

6.2.3. Handling virtual calls

As mentioned our analysis is able to automatically deal with polymorphic calls. JLayer features several of such calls. For instance, `Decoder` relies on an instance of `FrameDecoder` to decode every frame. The actual implementation of this interface is unknown at compile-time because it depends on the frame being read. The code provides three implementations: `LIDecoder`, `LIIDecoder` and `LIIIDecoder`.

As explained in §4, to solve this virtual call, the algorithm computes the maximum number of live objects among the three implementations, as shown in Table 2. The notation $[e_1, e_2, \dots]_m$ means the mth element of the array $[e_1, e_2, \dots]$ (here m stands for the frame mode). The consumption for `FrameDecoder.decodeFrame` is that of `LIIDecorder` as it is always greater than the other two.

Class	Method	MaxLive
Decoder	retrieveDecoder	$l = \{1, 2\}?3; l = 3?45662$
Decoder	initialize	$m = 3?4016 : 5136$
Decoder	decodeFrame	$34 + 10ns + [7ns, 7i, 7ns, 0]_m$
Player	<init>	$2335 + hS$
Player	decodeFrame	$52 + 10ns + [7ns, 7i, 7ns, 0]_m + F_{len} + K^m_{init}$
Player	play	$50840 + F_{MFL} + 17MS$
jpl	**play**	$57273 + F_{MFL} + 17MS + hS$
with $F_x \equiv x > 2048?2x + 1024 : 0$ $K^m_{init} \equiv !\, initialized?50788 - [0, 0, 0, 1120]_m : 0$		

Table 3: Summaries of most relevant methods of JLayer

analysis	time (sec.)	
	Init	Refined
Madeja	331	352
Rinard	457	499

file	% w. case error		% avg error	
	Init	Refined	Init	Refined
mp2	437	0.16	376.7	0.13
mp3	0.92	0.02	0.92	0.02

Table 4: Inference time and relative errors

6.2.4. Summarizing results

Table 3 shows the consumption of the most interesting methods using the methodology explained previously.

`jpl.play` is the main method, which creates an instance of `Player` and calls its `play` method. The consumption depends on the header size of the input file (hS), the maximum number of sub-bands appearing in the file (MS) and the maximum frame length (F_{MFL}).

One of the most interesting methods is `Play.decodeframe`. Its consumption depends on several parameters such as the frame mode (m), frame type or layer (l), the frame length (*len*, later bounded by MFL), the number of sub-bands (ns later bounded by MS), the stereo intensity (i) and whether the class was initialized (*initialized*). In this method, we can see the effect of modeling initialization using a parameter: K^m_{init} is 0 when *initialized* is true. Similarly, we can observe that the expressions reveal that the player requires more memory when the frame length is above 2048.

6.2.5. Introducing assumptions on inputs

While we compared the summaries against real executions of several audio files we found the bounds to be not very accurate (see Table 4, relative errors on columns entitled "Init"). Looking at the program, it turns out that it assumes that an input file may contain all possible layers, which is indeed unlikely since typical mp1/mp2/mp3 files only contain frames of the same type. To cope with multiple layers the program initializes a decoder for each frame type. This makes the analysis to compute a memory bound to accommodate for the three of them. In addition, the analysis considers the most memory-consuming candidate in the virtual call to `Decoder.decodeFrame`. Since for a given file all frames might be of a less consuming type, this results in an over-approximation. This is particularly notorious when input files are mp2, because the computed bounds consider also space for mp3s (Table 4).

We can make the analysis result more precise by assuming frame types do not change for a given input. For this, we set the frame layer l as a summary parameter of the whole application. That is, promoting l up to `main` method. By doing this, the result of the analysis of the virtual call no longer returns the maximum between candidates, but rather a conditional expression depending on l. Of course, we need to be sure that l is actually fixed during the execution of play (e.g., including an assertion in the code).

Table 5 shows the refined summaries. The new bounds are very tight as shown in Table 4 columns "Refined" resulting in negligible error. Regarding the capabilities of escape analyses, in this case we found no gain in using the *Rinard*

Class/Method	MaxLive
Decoder decodeFrame	$l = 1?34 + ns$ $l = 2?34 + 10ns + [7ns, 7i, 7ns, 0]_m$ $l = 3?0$
Player decodeFrame	$l = 1?52 + ns + F_{len} + K^m_{init}$ $l = 2?52 + 10ns + [7ns, 7i, 7ns, 0]_m + F_{len} + K^m_{init}$ $l = 3?20 + F_{len} + K^m_{init}$
Player play	$l = 1?5176 + MS - [0, 0, 0, 1120]_m + F_{MFL}$ $l = 2?5176 + 17MS - [0, 0, 0, 7MS + 1120]_m + F_{MFL}$ $l = 3?50840 - [0, 0, 0, 1120]_m + F_{MFL}$
jpl **play**	$l = 1?11609 + MS - [0, 0, 0, 1120]_m + F_{MFL} + hS$ $l = 2?11609 + 17MS - [0, 0, 0, 7MS + 1120]_m + F_{MFL} + hS$ $l = 3?57235 - [0, 0, 0, 1120]_m + F_{MFL} + hS$

Table 5: Refined summaries modeling the frame type as parameter

which is a more precise but more expensive (about 50% slower) analysis than *Madeja*.

7. Related Work

Quantitative program analysis is currently the subject of intensive research and has seen a noticeable progress in techniques that infer resource usage bounds [14, 11, 21, 2, 3, 37, 25].

In [11, 21] we presented a technique to infer parametric upper bounds of dynamic memory consumption in Java-like programs. Unlike the technique presented here, that analysis was not summary-based. In fact, in order to compute consumption we needed to model the global state of every reachable allocation. This involved the use of program invariants including the variables of the allocating method *and the variables of the methods in the whole call stack*. This monolithic approach jeopardizes usability and scalability, making it very hard for developers to understand the rationale behind the obtained consumption. In addition, that analysis was overly conservative when dealing with polymorphic calls.

Albert et al. [6] propose another technique to compute parametric bounds on the heap memory needs of a program in the context of garbage-collected languages, relying on the COSTA analyzer [1]. Following the framework of that analyzer, they first transform the program into a set of *rules*, to extract from them, in a second step, a set of *cost relations*. These are possibly-recursive equations, yielding a bound on the memory consumption of the methods they correspond to. In [4], they developed a technique to compute closed-form expressions, in terms of method parameters, providing safe upper bounds to those recursive equations, in some cases producing expressions beyond polynomials. However, this technique might not be able to produce parametric forms for every kind of recurrence first inferred, even if some of them represent polynomial consumption. At the moment, rather than a fine handling of recursion, our approach is focused on getting a precise account for loops. We illustrated this aspect in Program 1 page 3, where our technique computes that **new** A() will be performed at most $\frac{n(n+1)}{2}$ times, while COSTA returns that it might be run

n^2 times. We plan to extend our work in order to handle recursive code. In that sense the use of recurrence equations as used in COSTA is a sensible approach, but in our case we still aspire at maintaining the level of precision we have for loops.

Even though they explored in [5] how to recompute only what is needed in cases of incremental changes to the source code in the general COSTA framework, their technique is not quite *modular*. The expression they compute for every method m contains precise information about the objects created in the methods they invoke (and transitively the objects created by their callees). In a few words, they do not compute aggregated information about escaping objects[11]. They propagate information from callee to callers in order to decide later when those objects can be collected. In contrast, we introduced the idea of the Esc part in summaries that can produce aggregated information about the callees consumption, even hiding details about their allocations.

Even if both analyses rely on some form of points-to analysis to infer the lifetime of objects, our analysis and [6] differ in the fact that they take into account various possible policies for when the garbage collection is triggered, while, at the moment, we only consider collections at the end of methods: since our analysis is flow-insensitive, we cannot perform collection inside a method. We started working on turning our analysis into a flow-sensitive one, in particular in order to allow for collections at other interesting points of the program. But [6] also relies heavily on call unfoldings in order to be able to collect inside a method some objects that were allocated before it was called. We did not use that approach: even if the results can get closer to the actual needs of the program, we argue that breaking the modularity of the analysis renders its results harder to understand and use by the programmer.

Hofmann et al. use a type-based amortized analysis to bound heap memory requirements. The main idea of their technique is to associate a *potential* to each refined type so that the requirements of evaluating a function or method can be covered by the potentials associated with the arguments of the call.

They applied this idea on a core object-oriented calculus with explicit "free" instructions in [27, 28, 29]. In their framework, every allocation uses memory cells taken from a *free list*, so that the initial length of that list provides a bound to the overall memory requirements. To maintain this free list using a type system, they associate types to objects and methods. The object types are the usual classes refined with *views*; and to each refined type is associated a potential that indicates the number of memory cells already reserved for use by methods manipulating an object of that type. The method types are (possibly infinite) sets of tuples indicating the refined types of the arguments and of the corresponding result along with two numbers: the initial size the free list should have when the method is started and the number of memory cells that will be

[11] There is no formal description of summaries for heap memory requirements in the associated publications, but the tool COSTA provides an option to save and load assertions that can be interpreted as summaries.

returned to the free list at the end. While their first articles required manual type annotations, the latest, [29], defines an automatic type inference for linear bounds.

They had first introduced their technique of amortized types to bound resource use in [26] for functional languages and they had shown in [8] how to translate the Camelot functional language to JVM bytecode using a free list. In those works, the bounds are linear; in more recent works [25], they identify a specific form of polynomials well-suited to bounding the consumption of resources in their language. But, just as in their object-oriented works, their system is designed with a manual memory management, where the memory allocated to some value can be recovered when that value is filtered with a destructive variant of pattern matching. It seems that it would be rather tricky to combine this approach with a garbage collector.

The technique in [14] infers heap and stack bounds for low-level imperative programs relying on sized types. The technique naturally handle recursion but it infers linear expressions, does not handle polymorphism and requires explicit deallocation.

For functional languages, [37] proposes a non-compositional technique that, given a function, constructs a bound function that symbolically mimics the memory allocations of the former.

Our technique requires invariants that bound iterations. Various works, such as [23, 9, 33, 12], developed in the recent years techniques to automatically infer some program invariants. For instance [9] details a technique to infer a polynomial bound to the number of iterations of a fairly generic form of `for`-loops. These works look very promising for the goal of automatically inferring the majority of the invariants required to analyze even very large programs.

8. Conclusions and Future Work

We present a summary-based analysis to over-approximate the maximum number of objects created by a method that may be reachable (alive) during any of its runs. The analysis resorts to object lifetime information precomputed by an external points-to analysis. The compositionality is achieved by means of summaries that condense both the peak of objects created by a method that may be simultaneously reachable during its execution and the amount of created objects that may escape lifetime of methods.

Our prototype implementation is capable of inferring parametric polynomial bounds. We were able to analyze a medium-size real-life Java application with moderate human intervention, obtaining very tight bounds. We believe this is the first report of an analysis of such a large case study and serves as a proof of concept of the ideas exposed in the paper.

We plan to handle programs of larger sizes (e.g., those included in DaCapo benchmarks) —which are currently out of scope of existing techniques. Major challenge to achieve this goal is both the discovery of summary parameters and the manual annotation of invariants on creation and call sites. As a promising

note, vast majority of the invariants in JLAYER (95% +) are simple and they might be potentially discovered mechanically by powerful invariant detection tools such as [23]. The same can be said about the parameters of summaries and the associated binding. We are currently working on detecting recurrent patterns to automate discovery of summary parameters and required invariants.

We also working in automating the detection of common patterns described in §6 such as lazy initialization. In this direction, we would like to include new kinds of summary information such as escaping but non-accumulative sub-heaps which appear to be necessary to model with precision some of those patterns.

We are working on the design of a specification language for consumption summaries including tool support for checking the correctness of the annotations. We think this will facilitate user intervention and interoperability between different analyses. We are currently working on a more fine grained algorithm taking flow sensitiveness into account and enabling object reclaiming at the statement level.

Computing the peak of live objects is a key building block to assess memory consumption and it can serve as proxy of memory consumption behavior when comparing implementations of interfaces (such as what we explained for JLAYER in 6) or, more generally, to compare different algorithms.

Moreover, [10] shows how, using information of sizes for class types, the peak of live objects can be turned into an estimation of actual memory consumption for a particular setting. This can further allow to compute region sizes for RTSJ (see [11]).

A more general memory consumption analysis will also require to solve challenges posed by particularities of Java virtual machines like their garbage collectors, the fragmentation of memory and the representation of arrays. For the arrays, one possible solution could be to use special summaries for their allocation instruction: those summaries could then depend not only on their size but also on other VM parameters (e.g, block and page sizes, etc.).

Acknowledgments

Thanks to S. Verdoolaege for helping us with the ISCC calculator and to M. Rouaux, M. Grunberg and G. Krasny for participating in the development of the tool. This work partially supported by the projects UBACYT W0813, UBACYT F075, CONICET PIP 11220110100596CO, MinCyT PICT-2010-235, ANPCYT PICT 2011-1774, FRIC10/02-QUATRIX, MINCYT-BMWF AU/10/19, INRIA Associated Team ANCOME, LIA INFINIS and the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS).

[1] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Costa: Design and implementation of a cost and termination analyzer for Java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 113–132. Springer, 2007. ISBN 978-3-540-92187-5.

[2] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In Kolodner and Jr. [31], pages 129–138. ISBN 978-1-60558-347-1.

[3] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In Jan Vitek and Doug Lea, editors, *ISMM*, pages 121–130. ACM, 2010. ISBN 978-1-4503-0054-4.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning*, 46(2): 161–203, 2011.

[5] Elvira Albert, Jesús Correas, Germán Puebla, and Guillermo Román-Díez. Incremental resource usage analysis. In Oleg Kiselyov and Simon Thompson, editors, *PEPM*, pages 25–34. ACM, 2012. ISBN 978-1-4503-1118-2.

[6] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Heap space analysis for garbage collected languages. *Sci. Comput. Program.*, 78(9): 1427–1448, 2013.

[7] Michael Barnett, Manuel Fändrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) precise points-to analysis. In *Proceedings of the 2nd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'07)*, pages 11–18, 2007.

[8] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In Franz Baader and Andrei Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 347–362. Springer, 2004. ISBN 3-540-25236-3.

[9] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. Abc: Algebraic bound computation for loops. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2010. ISBN 978-3-642-17510-7.

[10] Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[11] Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In Jones and Blackburn [30], pages 141–150. ISBN 978-1-60558-134-7.

[12] David Cachera, Thomas P. Jensen, Arnaud Jobin, and Florent Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In Antoine Miné and David Schmidt, editors, *SAS*, volume 7460 of *Lecture Notes in Computer Science*, pages 58–74. Springer, 2012. ISBN 978-3-642-33124-4.

[13] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *IEEE PACT*, pages 280–291. IEEE Computer Society, 2001. ISBN 0-7695-1363-8.

[14] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. Analysing memory resource bounds for low-level programs. In Jones and Blackburn [30], pages 151–160. ISBN 978-1-60558-134-7.

[15] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Trans. VLSI Syst.*, 17(8):983–996, 2009.

[16] Byron Cook, Ashutosh Gupta, Stephen Magill, Andrey Rybalchenko, Jirí Simsa, Satnam Singh, and Viktor Vafeiadis. Finding heap-bounds for hardware synthesis. In *FMCAD*, pages 205–212. IEEE, 2009. ISBN 978-1-4244-4966-8.

[17] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Trans. Embedded Comput. Syst.*, 9(3), 2010.

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35. ACM Press, 1989. ISBN 0-89791-294-2.

[19] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3): 35–45, 2007.

[20] Diego Garbervetsky. *Parametric specification of dynamic memory utilization*. PhD thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, 2007.

[21] Diego Garbervetsky, Sergio Yovine, Víctor A. Braberman, Martín Rouaux, and Alejandro Taboada. Quantitative dynamic-memory analysis for Java. *Concurrency and Computation: Practice and Experience*, 23(14):1665–1678, 2011.

[22] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 292–304. ACM, 2010. ISBN 978-1-4503-0019-3.

[23] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 375–385. ACM, 2009. ISBN 978-1-60558-392-1.

[24] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000. ISSN 1433-2779.

[25] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 357–370. ACM, 2011. ISBN 978-1-4503-0490-0.

[26] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 185–197. ACM, 2003. ISBN 1-58113-628-5.

[27] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006. ISBN 3-540-33095-X.

[28] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009. ISBN 978-3-642-04026-9.

[29] Martin Hofmann and Dulma Rodriguez. Automatic type inference for amortised heap-space analysis. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 593–613. Springer, 2013. ISBN 978-3-642-37035-9.

[30] Richard E. Jones and Stephen M. Blackburn, editors. *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, 2008. ACM. ISBN 978-1-60558-134-7.

[31] Hillel Kolodner and Guy L. Steele Jr., editors. *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009, Dublin, Ireland, June 19-20, 2009*, 2009. ACM. ISBN 978-1-60558-347-1.

[32] Sun Microsystems. J2ME building blocks for mobile devices, May 2000.

[33] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 683–693. IEEE, 2012. ISBN 978-1-4673-1067-3.

[34] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. *Electr. Notes Theor. Comput. Sci.*, 55(2):255–276, 2001.

[35] Guillaume Salagnac, Christophe Rippert, and Sergio Yovine. Semi-automatic region-based memory management for real-time Java embedded systems. In *RTCSA*, pages 73–80. IEEE Computer Society, 2007.

[36] Alexandru Salcianu and Martin C. Rinard. Pointer and escape analysis for multithreaded programs. In Michael T. Heath and Andrew Lumsdaine, editors, *PPOPP*, pages 12–23. ACM, 2001. ISBN 1-58113-346-4.

[37] Leena Unnikrishnan and Scott D. Stoller. Parametric heap usage analysis for functional programs. In Kolodner and Jr. [31], pages 139–148. ISBN 978-1-60558-347-1.

[38] Sven Verdoolaege. *isl*: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010. ISBN 978-3-642-15581-9.

[39] Frédéric Vivien and Martin C. Rinard. Incrementalized pointer and escape analysis. In Michael Burke and Mary Lou Soffa, editors, *PLDI*, pages 35–46. ACM, 2001. ISBN 1-58113-414-2.

[40] Stephen Wolfram. *The Mathematica Book*. Wolfram Media, fifth edition, August 2003. ISBN 1579550223.

Appendix A. Proof of Correctness

Here we show that the equations presented in §4 can be used to build valid summaries given valid summaries for the callees.
Let $\mathit{Invoked}(S)$ be the set of methods invoked in S :

$$\begin{aligned} \mathit{Invoked}(\emptyset) &= \emptyset \\ \mathit{Invoked}(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle) &= \mathbf{CG}.\mathit{call}(\ell, \mu') \\ \mathit{Invoked}(S_1; S_2) &= \mathit{Invoked}(S_1) \cup \mathit{Invoked}(S_2) \end{aligned}$$

Recall that R_m is the set of invocation traces of the method m and that, for some trace $r \in R_m$, $R^r_{l,m'}$ is the set of invocation traces of m' in r at location l (note in particular that it is included in $R_{m'}$).

Observation 1. Given a method m and $\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle \in m.body$ and $m' \in \mathit{Invoked}(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I} \rangle)$, such that $\mathcal{I}$ is a valid iteration space, then for every invocation run $r \in R_m$ and for all $\bar{p}_m$, we have:

$$\mathbf{B}_m[\bar{\mathtt{fp}}_m / r_0(\bar{\mathtt{fp}}_m)] \implies \mathit{SC.UpBound}(E, \mathcal{I}, \bar{p}_m) \geq \max_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'} / r'_0(\bar{\mathtt{fp}}_{m'})]} E$$

$$\mathbf{B}_m[\bar{\mathtt{fp}}_m / r_0(\bar{\mathtt{fp}}_m)] \implies \mathit{SC.Summate}(E, \mathcal{I}, \bar{p}_m) \geq \sum_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'} / r'_0(\bar{\mathtt{fp}}_{m'})]} E$$

where $\mathit{vars}(E) \in \bar{p_{m'}}$.

Proof. It can be proved by induction on the set of invocation traces, using the definition of *SC.Summate* and *SC.UpBound*.

Observation 2. For every m such that $m.body = \langle \ell, \texttt{invoke}\ \mu'(\bar{\mathrm{v}}), \mathcal{I} \rangle$, $m' \in \mathit{Invoked}(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathrm{v}}), \mathcal{I} \rangle)$, $SH' = \mathbf{LIS}.esc_{(\ell,m')}(SH)$, then

$$\forall r \in R_m, \qquad \sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right| \geq \left|\mathsf{fresh}_r \upharpoonright SH\right|.$$

Proof. $\mathsf{fresh}_{r'} \upharpoonright SH$ is just a filter of objects using a sub-heap descriptor. By property of $\mathbf{LIS}_m.esc$ ($SH' = \mathbf{LIS}_m.esc_{l,m'}(SH) \supseteq \{sh' \in \mathcal{SH}_{m'} | \exists r \in R_m, r' \in R^r_{l,m'}, [sh']^{m'}_{r'} \cap [SH]^m_r \neq \emptyset\}$) we know that $\mathbf{LIS}_m.esc$ yields a superset of callee sub-heaps that corresponds to sub-heap descriptors existing in the caller. Since the body of method m is just a single invocation (possibly within a loop), every object escaping an invocation of m' will survive to m and potentially be part of its set of fresh objects and live objects. □

Proposition 1. For every method m, if for all $m' \in \mathit{Invoked}(m.body)$, $\mathbb{S}[m'].\mathsf{Esc}$ is valid, then $\mathsf{Esc}[m]$ is valid.

Proof. We prove that $\mathsf{Esc}[m : m.body]$ is valid by induction on the structure of $m.body$.

We first need to define a more general way to compute live and fresh parts of the heap. Given a set of invocations S such that $S \subseteq m.body$. Let $\mathsf{fresh}_r(S)$ be:

$$\mathsf{fresh}_r(S) = \bigcup_{r' \in R^r_{l,m'}, m' \in \mathit{Invoked}(\langle \ell, \texttt{invoke}\ \mu'(\bar{\mathrm{v}}), \mathcal{I} \rangle), \langle \ell, \texttt{invoke}\ \mu'(\bar{\mathrm{v}}), \mathcal{I} \rangle \in S} \mathsf{fresh}_{r'}(0, |r'|-1)$$

$\mathsf{fresh}_r(S)$ is the amount of live fresh objects obtained by all invocations included in S. We use the normal restriction to sub-heaps, $\mathsf{fresh}_r(S) \upharpoonright SH$.

The inductive hypothesis we will prove is:

$$\forall S \subseteq m.body, \forall SH \subseteq \mathcal{SH}_m,\ \forall r \in R_m,$$
$$\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{Esc}[m : S](SH) \geq \left|\mathsf{fresh}_r(S) \upharpoonright SH\right|$$

1. It trivially holds for $m.body = \emptyset$.
2. Case $m.body = s = \langle \ell, \texttt{invoke}\ \mu'(\bar{\mathrm{v}}), \mathcal{I} \rangle$. For this case, since the body only contains one statement, for all SH, $\mathsf{fresh}_r(S) \upharpoonright SH = \mathsf{fresh}_r \upharpoonright SH$. We have $\mathit{Invoked}(m) = \mathbf{CG}.call(\ell, \mu')$.
   (a) $\mathit{Invoked}(m) = \{m'\}$.

$$\mathsf{Esc}[m : s](SH) = SC.Summate(\mathsf{Esc}[m'](SH'), \mathcal{I} \wedge \mathbf{B}(m), \bar{p}_m)$$

Let $r \in R_m$ be an invocation trace for m. By hypothesis, we have that

$$\forall SH' \subseteq \mathcal{SH}_{m'},\ \forall r' \in R_{m'},$$
$$\mathbf{B}[\bar{\mathtt{fp}}'/r'_0(\bar{\mathtt{fp}}')] \implies S[m'].\mathsf{Esc}(SH') \geq \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right|$$

Thus, by applying *SC.Summate* to $S[m'].\mathsf{Esc}(SH')$, the observations 1 and 2 (considering that the only statement in m is the invocation to m'):

$$\forall SH \subseteq \mathcal{SH}_m,\ \forall r \in R_m,$$
$$\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies$$
$$\mathsf{Esc}[m:s](SH) \geq \sum_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'}/r'_0(\bar{\mathtt{fp}}_{m'})]} S[m'].\mathsf{Esc}(SH')$$
$$\geq \sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right| \geq \left|\mathsf{fresh}_r \upharpoonright SH\right|.$$

(b) $|\mathit{Invoked}(m)| > 1$.

Let $r \in R_m$ be an invocation trace for m.
By hypothesis, we have that

$$\forall SH' \subseteq \mathcal{SH}_{m'},\ \forall r' \in R_{m'}, m' \in \mathit{Invoked}(m)$$
$$\mathbf{B}[\bar{\mathtt{fp}}'/r'_0(\bar{\mathtt{fp}}')] \implies \mathbb{S}[m'].\mathsf{Esc}(SH') \geq \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right|$$

Let $R^{\sqcup,r}_{l,m'} = \bigcup_{m' \in \mathit{Invoked}(m)} R^r_{l,m'}$ and $\mathbf{B}^{\sqcup}_{l,m'} = \bigwedge_{m' \in \mathit{Invoked}(m)} \mathbf{B}(m')$.
By properties of least upper bound:

$$\forall SH' \subseteq \mathcal{SH}_{m'},\ \forall r' \in R^{\sqcup,r}_{l,m'},$$
$$\mathbf{B}^{\sqcup}_{l,m'}[\bar{\mathtt{fp}}'/r'_0(\bar{\mathtt{fp}}')]$$
$$\implies \bigsqcup_{m' \in \mathit{Invoked}(m)} S[m'].\mathsf{Esc}(SH') \geq \max_{r' \in R^{\sqcup,r}_{l,m'}} \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right|$$

Thus, by applying *SC.Summate* to $\bigsqcup_{m' \in \mathit{Invoked}(m)} S[m'].\mathsf{Esc}(SH)$, the observations 1 and 2 (considering that the only statement in m is the invocation to m'):

$$\forall SH \subseteq \mathcal{SH}_m,\ \forall r \in R_m,$$
$$\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies$$
$$\mathsf{Esc}[m:s](SH) \geq \sum_{r' \in R^r_{l,m'} \wedge \mathbf{B}_{m'}[\bar{\mathtt{fp}}_{m'}/r'_0(\bar{\mathtt{fp}}_{m'})]} \Big(\bigsqcup_{m' \in \mathbf{CG}.\mathit{call}(\ell,\mu')} S[m'].\mathsf{Esc}(SH')\Big)$$
$$\geq \sum_{r' \in R^r_{l,m'}} \max_{r'' \in R^{\sqcup,r}_{l,m'}} \left|\mathsf{fresh}_{r''} \upharpoonright SH'\right|$$
$$\geq \sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right| \geq \left|\mathsf{fresh}_r \upharpoonright SH\right|.$$

3. Case $m : S_1; S_2$. By the inductive hypothesis, we know

$$\mathsf{Esc}[m : S_i](SH) \geq \left|\mathsf{fresh}_r(S_i) \upharpoonright SH\right|$$

Combining these facts, we obtain:

$$\begin{aligned}\mathsf{Esc}[m : S](SH) &= \mathsf{Esc}[m : S_1](SH) + \mathsf{Esc}[m : S_2](SH)\\ &\geq \left|\mathsf{fresh}_r(S_1) \upharpoonright SH\right| + \left|\mathsf{fresh}_r(S_2) \upharpoonright SH\right|\\ &\geq \left|\mathsf{fresh}_r(S_1) \upharpoonright SH \cup \mathsf{fresh}_r(S_2) \upharpoonright SH\right|\\ &\geq \left|\mathsf{fresh}_r(S) \upharpoonright SH\right|\end{aligned}$$

Hence, $\mathsf{Esc}[m]$ is valid. □

Proposition 2. For every method m, if for all $m' \in \mathit{Invoked}(m.\mathit{body})$, $\mathsf{Esc}[m']$ and $\mathbb{S}[m'].\mathsf{MaxLive}$ are valid, then $\mathsf{MaxLive}[m]$ is valid.

Proof. We prove $\mathsf{MaxLive}[m : m.\mathit{body}]$ is valid by induction on the structure of $m.\mathit{body}$.

Given $S \subseteq m.\mathit{body}$, the function $\mathsf{fresh}_r(S, t_1, t_2)$ is a restriction of $\mathsf{fresh}_r(t_1, t_2)$ to objects created only during the execution of the statements in S. That is, the set of fresh and live objects at r_2, with respect to r_1 but considering only the allocations made in S.

The inductive hypothesis, we will prove is:

$$\begin{aligned}&\forall S \subseteq m.\mathit{body}, \forall r \in R_m, \forall t, 0 \leq t < |r|,\\ &\qquad \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{MaxLive}[m : S] \geq \max_{t' \leq t}\{|\mathsf{fresh}_r(S, t', t)|\}\end{aligned}$$

1. It trivially holds for $m.\mathit{body} = \emptyset$.
2. Case $m.\mathit{body} = s = \langle \ell, \mathtt{invoke}\ \mu'(\bar{\mathtt{v}}), \mathcal{I}\rangle$. For this case, since the body only contains one statement, $\mathsf{fresh}_r(S, t_1, t_2) = \mathsf{fresh}_r(t_1, t_2)$.
   (a) $\mathit{Invoked}(m) = \{m'\}$. By definition, we have that

$$\begin{aligned}\mathsf{UB}[m : s] &= \mathit{SC.UpBound}(\mathsf{MaxLive}[m'] - \mathsf{Esc}[m'](\mathcal{SH}_{m'}),\\ &\qquad \mathcal{I} \wedge \mathbf{B}_m, \bar{p}_m)\\ \mathsf{SUM}[m : s] &= \mathit{SC.Summate}(\mathsf{Esc}[m'](\mathcal{SH}_{m'}), \mathcal{I} \wedge \mathbf{B}_m, \bar{p}_m)\\ \mathsf{MaxLive}[m : s] &= \mathsf{UB}[m : s] + \mathsf{SUM}[m : s]\end{aligned}$$

   Let $r \in R_m$ be an invocation trace for m. By hypothesis, we have that

$$\begin{aligned}&\forall r' \in R_{m'}, \forall t, 0 \leq t < |r'|,\\ &\qquad \mathbf{B}[\bar{\mathtt{fp}}\mathtt{'}/r'_0(\bar{\mathtt{fp}}')] \implies \mathbb{S}[m'].\mathsf{MaxLive} \geq \max_{t' \leq t}\{|\mathsf{fresh}_r(t', t)|\}\end{aligned}$$

   and

$$\begin{aligned}&\forall SH' \subseteq \mathcal{SH}_{m'},\ \forall r' \in R_{m'},\\ &\qquad \mathbf{B}[\bar{\mathtt{fp}}\mathtt{'}/r'_0(\bar{\mathtt{fp}}')] \implies \mathbb{S}[m'].\mathsf{Esc}(SH') \geq \left|\mathsf{fresh}_{r'} \upharpoonright SH'\right|\end{aligned}$$

Then, by applying *SC.UpBound* to $\mathbb{S}[m']$.MaxLive and by observation 1 we have:

$$\begin{aligned}
&\forall r \in R_m, r' \in R^r_{l,m'} \forall t, 0 \le t < |r'|, \\
&\quad \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}}')] \implies \mathsf{UB}[m:s] \ge \max_{t' \le t}\{|\mathsf{fresh}_{r'}(t',t)| - \left|\mathsf{fresh}_{r'} \restriction SH_{m'}\right|\} \\
&\quad = \max_{t' \le t}\{|\mathsf{fresh}_{r'}(t',t)|\} - \left|\mathsf{fresh}_{r'} \restriction SH_{m'}\right| \\
&\quad \ge \max_{t' \le t}\{|\mathsf{fresh}_{r'}(t',t)|\} - \sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \restriction \mathcal{SH}_{m'}\right|
\end{aligned}$$

On the other hand (by applying *SC.Summate* to $\mathbb{S}[m']$.$\mathsf{Esc}(\mathcal{SH}_m)$, and observation 2):

$$\begin{aligned}
&\forall r \in R_m, \\
&\quad \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{SUM}[m:s] \ge \sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \restriction \mathcal{SH}_{m'}\right|
\end{aligned}$$

Then, taking $\mathsf{UB}[m:s] + \mathsf{SUM}[m:s]$, simplifying $\sum_{r' \in R^r_{l,m'}} \left|\mathsf{fresh}_{r'} \restriction \mathcal{SH}_{m'}\right|$ we have:

$$\begin{aligned}
&\forall r \in R_m, r' \in R^r_{l,m'} \forall t, 0 \le t < |r'|, \\
&\quad \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}}')] \implies \mathsf{MaxLive}[m:s] \ge \max_{t' \le t}\{|\mathsf{fresh}_{r'}(t',t)|\}
\end{aligned}$$

Then, since the unique statement in m is the invocation to m' and that MaxLive is an upper bound for every invocation to m' we can conclude:

$$\begin{aligned}
&\forall r \in R_m, \forall t, 0 \le t < |r'|, \\
&\quad \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}}')] \implies \mathsf{MaxLive}[m:s] \ge \max_{t' \le t}\{|\mathsf{fresh}_{r}(t',t)|\}
\end{aligned}$$

(b) $|\mathit{Invoked}(m)| > 1$. This case is more involved but follows similarly as above. Essentially there is an additional max operator (for the summaries of callee candidates which has to be valid) inside the maximization of the difference in UB.

3. Case $m : S_1; S_2$.

$$\begin{aligned}
\mathsf{MaxLive}[m:S_1;S_2] &= \mathsf{UB}[m:S_1] \sqcup \mathsf{UB}[m:S_2] \\
&\quad + \mathsf{SUM}[m:S_1] + \mathsf{SUM}[m:S_2] \\
&= (\mathsf{UB}[m:S_1] + \mathsf{SUM}[m:S_1] + \mathsf{SUM}[m:S_2]) \\
&\sqcup (\mathsf{UB}[m:S_2] + \mathsf{SUM}[m:S_1] + \mathsf{SUM}[m:S_2]) \\
&= (\mathsf{MaxLive}[m:S_1] + \mathsf{SUM}[m:S_2]) \\
&\sqcup (\mathsf{MaxLive}[m:S_2] + \mathsf{SUM}[m:S_1])
\end{aligned}$$

By inductive hypotesis:

$$\forall r \in R_m, \forall t, 0 \leq t < |r|,$$
$$\mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{MaxLive}[m : S_i] \geq \max_{t' \leq t}\{|\mathsf{fresh}_r(S_i, t', t)|\}$$

Using a similar reasoning to the one we used for Esc, we get that:

$$\forall r \in R_m, \mathbf{B}[\bar{\mathtt{fp}}/r_0(\bar{\mathtt{fp}})] \implies \mathsf{SUM}[m : S_i] \geq |\mathsf{fresh}_r(S_i)|$$

Then (omitting quantifiers for simplicity):

$$\begin{aligned}
\mathsf{MaxLive}[m : S_1; S_2] &= (\mathsf{MaxLive}[m : S_1] + \mathsf{SUM}[m : S_2]) \\
&\quad \sqcup ([\mathsf{MaxLive}[m : S_2] + \mathsf{SUM}[m : S_1]) \\
&\geq (\max_{t' \leq t}\{|\mathsf{fresh}_r(S_1, t', t)|\} + |\mathsf{fresh}_r(S_2)|) \\
&\quad \sqcup (\max_{t' \leq t}\{|\mathsf{fresh}_r(S_2, t', t)|\} + |\mathsf{fresh}_r(S_1)|) \\
&\geq (\max_{t' \leq t}\{|\mathsf{fresh}_r(S_1, t', t)|\} + \max_{t' \leq t}\{|\mathsf{fresh}_r(S_2, t', t)|\}) \\
&\quad \sqcup (\max_{t' \leq t}\{|\mathsf{fresh}_r(S_2, t', t)|\} + \max_{t' \leq t}\{|\mathsf{fresh}_r(S_1, t', t)|\}) \\
&= (\max_{t' \leq t}\{|\mathsf{fresh}_r(S_1, t', t)|\} + \max_{t' \leq t}\{|\mathsf{fresh}_r(S_2, t', t)|\}) \\
&\geq \max_{t' \leq t}\{|\mathsf{fresh}_r(S_1 \cup S_2, t', t)|\}
\end{aligned}$$

Hence, $\mathsf{MaxLive}[m]$ is valid. $\square$