Live Heap Space Analysis for Languages with Garbage Collection

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

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

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

Abstract

The *peak heap consumption* of a program is the maximum size of the *live* data on the heap during the execution of the program, i.e., the minimum amount of heap space needed to run the program without exhausting the memory. It is well-known that garbage collection (GC) makes the problem of predicting the memory required to run a program difficult. This paper presents, the best of our knowledge, the first *live heap space* analysis for garbage-collected languages which infers accurate upper bounds on the peak heap usage of a program's execution that are not restricted to any complexity class, i.e., we can infer exponential, logarithmic, polynomial, etc., bounds. Our analysis is developed for an (sequential) object-oriented bytecode language with a *scoped-memory* manager that reclaims unreachable memory when methods return. We also show how our analysis can accommodate other GC schemes which are closer to the *ideal* GC which collects objects as soon as they become unreachable. The practicality of our approach is experimentally evaluated on a prototype implementation. We demonstrate that it is fully automatic, reasonably accurate and efficient by inferring live heap space bounds for a standardized set of benchmarks, the JOlden suite.

Categories and Subject Descriptors F3.2 [*Logics and Meaning of Programs*]: Program Analysis; F2.9 [*Analysis of Algorithms and Problem Complexity*]: General; D3.2 [*Programming Languages*]

General Terms Languages, Theory, Verification, Reliability

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

1. Introduction

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. It is widely recognized that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime. Another motivation is to configure real-time garbage collectors to avoid mutator starvation. Besides, upper bounds on the memory requirement of programs have been proposed for resource-bound certification [10] where certificates encode security properties involving resource usage requirements, e.g., the (untrusted) code must adhere to specific bounds on its memory usage. On the other hand, automatic memory management (also known as garbage collection) is a very powerful and useful mechanism which is increasingly used in high-level languages such as Java. Unfortunately, GC makes the problem of predicting the memory required to run a program difficult.

A first approximation to this problem is to infer the *total memory allocation*, i.e., the *accumulated* amount of memory allocated by a program ignoring GC. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual memory requirement. *Live heap space analysis* [18, 5, 8] aims at approximating the size of the *live* data on the heap during a program's execution, which provides a much tighter estimation. This paper presents a general approach for inferring the *peak heap consumption* of a program's execution, i.e., the maximum of the live heap usage along its execution. Our live heap space analysis is developed for (an intermediate representation of) an object-oriented *bytecode* language with automatic memory management. Programming languages which are compiled to bytecode and executed on a *virtual machine* are widely used nowadays. This is the approach used by Java bytecode and .NET.

Analysis of live heap usage is different from total memory allocation because it involves reasoning on the memory consumed at *all program states* along an execution, while total allocation needs to observe the consumption at the *final* state only. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [20] has been applied only to infer total allocation. Intuitively, given a program, this approach produces a *cost relation system* (CR for short) which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the CR. This approach leads to very accurate cost bounds as it is not limited to any complexity class (infers polynomial, logarithmic, exponential consumption, etc.) and, besides, it can be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, it is not suitable to infer peak heap consumption because it is not an accumulative resource of a program's execution as CR capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate CR, live heap space analysis is currently restricted to polynomial bounds and non-recursive methods [5] or to linear bounds dealing with recursion [8].

Inspired by the basic techniques used in cost analysis, in this paper, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, we need to characterize the behavior of

ISMM'09, June 19–20, 2009, Dublin, Ireland.

the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method. Then, such bound can be manipulated, by relying on information computed by *escape analysis* [4], to extract from it an upper bound on its escaped memory.
2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption* CR which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our CRs is that they can be solved by using existing tools for solving standard CRs.
3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.
4. *Implementation.* We report on a prototype implementation which is integrated in the COSTA system [2] and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

2. Bytecode: Syntax and Semantics

Bytecode programs are complicated for both human and automatic analysis because of their unstructured control flow, operand stack, etc. Therefore, it is customary to formalize analyses on intermediate representations of the bytecode (e.g., [3, 19, 13]). We consider a rule-based *procedural* language (in the style of any of the above) in which a *rule-based program* consists of a set of *procedures* and a set of classes. A procedure p with k input arguments $\bar{x} = x_1, \ldots, x_k$ and m output arguments $\bar{y} = y_1, \ldots, y_m$ is defined by one or more *guarded rules*. Rules adhere to the following grammar:

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

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_t$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, c)$, which succeeds if the runtime class of x is exactly c. A class c is a finite set of typed field names, where the type can be integer or a class name. The superscript i on a class c is a unique identifier which associates objects with the program points where they have been created. The key features of this language are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* blocks

```
class Test {
  static Tree m(int n) {
    if ( n>0 ) return new
      Tree(m(n-1),m(n-1),f(n));
    else return null;
  }
  static int f(int n) {
    int a=0,i=n;
    while ( n>1 ) {
      a += g(n).intValue();
      n=n/2;
    }
    for(; i>1; i=i/2)
      a *= h(i).intValue();
    return a;
  }
  static Integer g(int n) {
    Integer x=new Integer(n);
    return new Integer(x.intValue()+1);
  }
  static Long h(int n) {
    return new Long(n-1);
  }
} // end of class Test

class Tree {
  Tree l,r;
  int d;
  Tree(Tree l,Tree r,int d) {
    this.l = l;
    this.r = r;
    this.d = d;
  }
}
```

Figure 1. Java code of running example

$$
\begin{array}{ll}
(1) & \mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= \\
& \quad \mathbf{n > 0}, \\
& \quad \mathrm{s}_0 := \mathsf{new}\ \mathrm{Tree}^1; \\
& \quad \mathrm{s}_1 := \mathrm{n} - 1, \\
& \quad \mathrm{m}(\langle \mathrm{s}_1 \rangle, \langle \mathrm{s}_1 \rangle), \\
& \quad \mathrm{s}_2 := \mathrm{n} - 1, \\
& \quad \mathrm{m}(\langle \mathrm{s}_2 \rangle, \langle \mathrm{s}_2 \rangle), \\
& \quad \mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{s}_3 \rangle), \\
& \quad \mathrm{init}(\langle \mathrm{s}_0, \mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3 \rangle, \langle \rangle), \\
& \quad \mathrm{r} = \mathrm{s}_0.
\end{array}
$$

$$
\begin{array}{ll}
(2) & \mathrm{m}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= \\
& \quad \mathbf{n \leq 0}, \\
& \quad \mathrm{r} := \mathsf{null}.
\end{array}
$$

$$
\begin{array}{ll}
(3) & \mathrm{f}(\langle \mathrm{n} \rangle, \langle \mathrm{a} \rangle) ::= \\
& \quad \mathrm{a} := 0, \\
& \quad \mathrm{i} := \mathrm{n}, \\
& \quad \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle), \\
& \quad \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle).
\end{array}
$$

$$
\begin{array}{ll}
(4) & \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::= \\
& \quad \mathbf{n > 1}, \\
& \quad \mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{s}_0 \rangle), \\
& \quad \mathrm{intValue_1}(\langle \mathrm{s}_0 \rangle, \langle \mathrm{s}_0 \rangle) \\
& \quad \mathrm{a} := \mathrm{a} + \mathrm{s}_0, \\
& \quad \mathrm{n} := \mathrm{n}/2, \\
& \quad \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle).
\end{array}
$$

$$
\begin{array}{ll}
(5) & \mathrm{f_c}(\langle \mathrm{n}, \mathrm{a} \rangle, \langle \mathrm{n}, \mathrm{a} \rangle) ::= \\
& \quad \mathbf{n \leq 1}.
\end{array}
$$

$$
\begin{array}{ll}
(6) & \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::= \\
& \quad \mathbf{i > 1}, \\
& \quad \mathrm{h}(\langle \mathrm{i} \rangle, \langle \mathrm{s}_0 \rangle), \\
& \quad \mathrm{intValue_2}(\langle \mathrm{s}_0 \rangle, \langle \mathrm{s}_0 \rangle) \\
& \quad \mathrm{a} := \mathrm{a} * \mathrm{s}_0, \\
& \quad \mathrm{i} := \mathrm{i}/2, \\
& \quad \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle).
\end{array}
$$

$$
\begin{array}{ll}
(7) & \mathrm{f_d}(\langle \mathrm{i}, \mathrm{a} \rangle, \langle \mathrm{i}, \mathrm{a} \rangle) ::= \\
& \quad \mathbf{i \leq 1}.
\end{array}
$$

$$
\begin{array}{ll}
(8) & \mathrm{g}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= \\
& \quad \mathrm{x} := \mathsf{new}\ \mathrm{Integer}^2, \\
& \quad \mathrm{init_1}(\langle \mathrm{x}, \mathrm{n} \rangle, \langle \rangle), \\
& \quad \mathrm{intValue_1}(\langle \mathrm{x} \rangle, \langle \mathrm{s}_0 \rangle), \\
& \quad \mathrm{s}_0 := \mathrm{s}_0 + 1. \\
& \quad \mathrm{r} := \mathsf{new}\ \mathrm{Integer}^3, \\
& \quad \mathrm{init_1}(\langle \mathrm{r}, \mathrm{s}_0 \rangle, \langle \rangle).
\end{array}
$$

$$
\begin{array}{ll}
(9) & \mathrm{h}(\langle \mathrm{n} \rangle, \langle \mathrm{r} \rangle) ::= \\
& \quad \mathrm{s}_0 := \mathrm{n} - 1. \\
& \quad \mathrm{r} := \mathsf{new}\ \mathrm{Long}^4, \\
& \quad \mathrm{init_2}(\langle \mathrm{r}, \mathrm{s}_0 \rangle, \langle \rangle).
\end{array}
$$

$$
\begin{array}{ll}
(10) & \mathrm{init}(\langle \mathrm{this}, \mathrm{l}, \mathrm{r}, \mathrm{d} \rangle, \langle \rangle) ::= \\
& \quad \mathrm{this.l} := \mathrm{l}, \\
& \quad \mathrm{this.r} := \mathrm{r}, \\
& \quad \mathrm{this.d} := \mathrm{d}.
\end{array}
$$

Figure 2. Intermediate representation of running example.

guarded by a type check, and (5) procedures may have *multiple return* values. The translation from (Java) bytecode to the rule-based form is performed in two steps. First, a *control flow graph* (CFG) is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. E.g., this translation is explained in more detail in [3]. For simplicity, our language does not include advanced features of Java bytecode, such as exceptions, interfaces, static methods and fields, access control (e.g., the use of public, protected and private modifiers) and primitive types besides integers and references. Such features can be easily handled in our framework and indeed our implementation deals with full (sequential) Java bytecode.

EXAMPLE 2.1. *Fig. 1 depicts our running example in Java, and Fig. 2 depicts its corresponding rule-based representation where the procedures are named as the method they represent and "*$\mathrm{f_c}$*" and "*$\mathrm{f_d}$*" denote intermediate procedures for* f. *The Java program is included only for clarity as the analyzer generates the rule-based representation from the corresponding bytecode only. As an example, we explain rules (1) and (2) which correspond to method* m. *Each rule is guarded by a corresponding condition, resp.* $\mathbf{n} > \mathbf{0}$ *and* $\mathbf{n} \leq \mathbf{0}$. *Variable names of the form* $\mathrm{s_i}$ *indicate that they originate from stack positions. In rule (1), the "*$\mathsf{new}\ \mathrm{Tree}^1$*" instruction creates an object of type* Tree *(the superscript* 1 *is the unique identifier for this program point) and assigns the variable* $\mathrm{s_0}$ *to its reference (which corresponds to pushing the reference on the stack in the original bytecode). Then, the local variable* n *is decremented by one and the result is assigned to* $\mathrm{s_1}$. *Next, the method* m *is recursively invoked which receives as input argument the result of the previous operation (*$\mathrm{s_1}$*) and returns its result in* $\mathrm{s_1}$. *Similar invocations to methods* m, f *and* init *follow. In Java bytecode, constructor methods are named* init. *In both rules, the return value is* r *which in (1) is assigned to the object reference and in (2) to* null. *It can be observed that, like in bytecode, all guards and instructions correspond to three-address code, except for calls to procedures which may involve more variables as parameters. The methods* $\mathrm{intValue_1}$ *and* $\mathrm{init_1}$ *belong to class* Integer, *and* $\mathrm{intValue_2}$ *and* $\mathrm{init_2}$ *belong to class* Long. □

Observe in the example that, in our syntax, with the aim of simplifying the presentation, we do not distinguish between calls to methods and calls to intermediate procedures. For instance, $\mathrm{f_c}$ and $\mathrm{f_d}$ are intermediate procedures while f is the method. This distinction can be made observable in the translation phase trivially and, when needed, we assume such distinction is available.

2.1 Semantics

The execution of bytecode in rule-based form is exactly like standard bytecode; a thorough explanation is outside the scope of this paper (see [14]). An *operational semantics* for rule-based bytecode is shown in Fig. 3. An *activation record* is of the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a sequence of instructions and tv a variable mapping. Executions proceed between *configurations* of the form $A; h$, where A is a stack of activation records and h is the *heap* which is a partial map from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by the memory location r in h and $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$. An object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the type of the fields.

Intuitively, rule (1) accounts for all instructions in the bytecode semantics which perform arithmetic and assignment operations. The evaluation $eval(exp, tv)$ returns the evaluation of the arithmetic or Boolean expression exp for the values of the corresponding variables from tv in the standard way, and for reference variables, it returns the reference. Rules (2), (3) and (4) deal with objects. We assume that $\mathsf{newobject}(c^i)$ creates a new object of class c and initializes its fields to either 0 or null, depending on their types. Rule (5) (resp., (6)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}, \bar{y}']$ records the association between the formal and actual return variables. It is assumed that newenv creates a new mapping of local variables for the corresponding method, where each variable is initialized as newobject does.

An execution starts from an *initial configuration* of the form $\langle \bot, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ and ends when we reach a *final configuration* $\langle \bot, \epsilon, tv'\rangle; h'$ where tv and h are initialized to suitable initial values, tv' and h' include the final values, and $\bot$ is a special symbol indicating an initial state. We assume that any object stored in the initial heap h is reachable from (at least) one of the x_i, namely there are not *collectable* objects that can removed from h at the initial state. Note that $\mathrm{dom}(tv) = \mathrm{dom}(tv') = \bar{x} \cup \bar{y}$. Finite executions can be regarded as *traces* $S_0 \rightsquigarrow S_1 \rightsquigarrow \cdots \rightsquigarrow S_\omega$, denoted $S_0 \rightsquigarrow^* S_\omega$, where S_ω is a final configuration. Infinite traces correspond to non-terminating executions.

3. Total Memory Allocation Analysis

Let us first define the notion of total memory consumption. We let $size(c)$ denote the amount of memory required to hold an instance object of class c, $size(o)$ denotes the amount of memory occupied by an object o, and $size(h)$ denotes the amount of memory occupied by all objects in the heap h, namely $\Sigma_{r \in \mathrm{dom}(h)} size(h(r))$. We consider the semantics in Fig. 3 where no GC is performed. Given a trace $t \equiv A_1; h_1 \rightsquigarrow^* A_n; h_n$, the *total memory allocation* of t is defined as $total(t) = size(h_n) - size(h_1)$.

In this section, we briefly overview the application of the cost analysis framework, originally proposed by Wegbreit [20], to total memory consumption inference of bytecode as proposed in [3]. The original analysis framework [1] takes as input a program and a cost model $\mathcal{M}$, and outputs a closed-form upper bound that describes its execution cost w.r.t. $\mathcal{M}$. The cost model $\mathcal{M}$ defines the cost that we want to accumulate. For instance, if the cost model is the number of executed instructions, $\mathcal{M}$ assigns cost 1 to all instructions. The application of this framework to total memory consumption of bytecode takes as input a bytecode program and the following cost model $\mathcal{M}^t$, which is a simplification for our language of the cost model for heap space usage of [3].

DEFINITION 3.1 (heap consumption cost model [3]). *Given a bytecode instruction b, the heap consumption cost model is defined as*

$$\mathcal{M}^t(b) = \begin{cases} size(c^i) & b \equiv x := \mathsf{new}\ c^i \\ 0 & \textit{otherwise} \end{cases}$$

For a sequence of instructions, $\mathcal{M}^t(b_1 \cdots b_n) = \mathcal{M}^t(b_1) + \cdots + \mathcal{M}^t(b_n)$. □

3.1 Inference of Size Relations

The aim of the analysis is to approximate the memory consumption of the program as an upper bound function in terms of its input *data size*s. As customary, the *size* of data is determined by its variable type: the size of an integer variable is its value; the size of an array is its length; and the size of a reference variable is the length of the longest path that can be traversed through the corresponding object (e.g., length of a list, depth of a tree, etc.). To keep the presentation simple, we use the original variable names (possible primed) to refer to the corresponding abstract (size) variables; but we write the size in *italic* font. For instance, let x be a reference to a tree, then x represents the depth of x. When we need to compute the sizes $\bar{v}$ of a given tuple of variables $\bar{x}$, we use the notation $\bar{v} = \alpha(\bar{\mathrm{x}}, tv, h)$, which means that the integer value v_i is the size of the variable $\mathrm{x_i}$ in the context of the variables table tv and the heap h. For instance, if x is the reference to a tree, we need to access the heap h where the tree is allocated to compute its depth and obtain v. If x is an integer variable, then its size (value) can be obtained from the variable table tv.

Standard *size analysis* is used in order to obtain relations between the sizes of the program variables at different program points. For instance, associated to procedure $\mathrm{f_c}$, we infer the size relation $n' = n/2$ which indicates that the value of n decreases by half when calling $\mathrm{f_c}$ recursively. We denote by φ_r the conjunction of linear constraints which describes the relations between the abstract variables of a rule r and refer to [9, 3] for more information.

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

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

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

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

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

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

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

(1)	$m(n){=}size(\mathrm{Tree}^1){+}m(s_1){+}m(s_2){+}$ $f(n){+}init(s_0, s_1, s_2, s_3)$	$\{n{>}0, s_0{=}1,$ $s_1{=}n{-}1, s_2{=}n{-}1\}$
(2)	$m(n){=}0$	$\{n{\leq}0\}$
(3)	$f(n){=}f_c(n, a) + f_d(i, a')$	$\{a{=}0, i{=}n\}$
(4)	$f_c(n, a){=}g(n){+}f_c(n', a')$	$\{n{>}1, n'{=}n/2\}$
(5)	$f_c(n, a){=}0$	$\{n{\leq}1\}$
(6)	$f_d(i, a){=}h(i){+}f_d(i', a')$	$\{i{>}1, i'{=}i/2\}$
(7)	$f_d(i, a){=}0$	$\{i{\leq}0\}$
(8)	$g(n){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)$	$\{x{=}1\}$
(9)	$h(n){=}size(\mathrm{Long}^4)$	$\{r{=}1, s_0{=}n{-}1\}$
(10)	$init(this, l, r, d){=}0$	$\{\}$

Figure 4. Total Allocation CR.

3.2 Generation of Cost Relations

In a nutshell, given a bytecode program P, the analysis of [3] proceeds in three steps: (1) it first transforms it into an equivalent rule-based program (our work directly starts from such rule-based form), (2) it infers size relations as explained above, (3) it generates a CR which describes the total memory consumption of the program as follows.

DEFINITION 3.2 (total allocation CR [3]). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle){::=}\mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and the size relations* φ_r *computed for* r*. We distinguish the subsequence of all calls to procedures* $\mathrm{b}_{\mathrm{i}_1} \ldots \mathrm{b}_{\mathrm{i}_k}$ *in* r*, with* $1 \leq i_1 \leq \cdots \leq i_k \leq n$ *and assume* $\mathrm{b}_{\mathrm{i}_\mathrm{j}} = \mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle\bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$*. Then, the cost equation for* r *is:*

$$p(\bar{x}) = \mathcal{M}^t(g, b_1, \ldots, b_n) + \Sigma_{j=1}^{k}\, q_{i_j}(\bar{x}_{i_j}),\ \ \varphi_r$$

Given a program P*, we denote by* $\mathcal{S}_P$ *the cost relation generated for each rule in* P *w.r.t. the heap consumption cost model* $\mathcal{M}^t$. □

Note that each call in the rule $\mathrm{q}_{\mathrm{i}_\mathrm{j}}(\langle\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}\rangle, \langle\bar{\mathrm{y}}_{\mathrm{i}_\mathrm{j}}\rangle)$ has a corresponding abstract version $b^{\alpha}_{i_j} = q_{i_j}(\bar{x}_{i_j})$ where $\bar{x}_{i_j}$ are the size abstractions of $\bar{\mathrm{x}}_{\mathrm{i}_\mathrm{j}}$. The output variables are ignored in the CR as the cost is a function of the *input* data sizes, however it should be noted that the possible effect of output variables on the cost has been already modeled by the size relation φ_r. For simplicity, the same procedure name is used to define its associated cost relation, but in *italic* font.

EXAMPLE 3.3. *The CR generated for the rule-based program in Fig. 2 w.r.t.* $\mathcal{M}^t$ *is depicted in Fig. 4. To simplify the presentation, we assume that the total heap consumption of all external methods (*init_1, $\mathrm{intValue}_1$, init_2 *and* $\mathrm{intValue}_2$*) is* 0 *and we do not show them in the equations from now on. Consider, for example, equation (4). It states that the memory consumption of executing* $\mathrm{f_c}(\langle\mathrm{n}, \mathrm{a}\rangle, \langle\mathrm{n}, \mathrm{a}\rangle)$ *is the total memory consumption of executing* $\mathrm{g}(\langle\mathrm{n}\rangle, \langle\mathrm{r}\rangle)$ *plus the one of* $\mathrm{f_c}(\langle\mathrm{n}', \mathrm{a}'\rangle, \langle\mathrm{n}', \mathrm{a}'\rangle)$*. The set of constraints attached to equation (4) includes information on: (1) how the sizes of the data change when the program moves from one rule to another, e.g., the constraint* $n' = n/2$ *indicates that the value of* n *decreases by half when calling* $\mathrm{f_c}$ *recursively; and (2) numeric conditions (obtained by abstracting the guards) under which the corresponding rule is applicable, e.g.,* $n > 1$ *indicates that the equation can be applied only when* n *is greater than* 1. □

An important observation is that, as discussed in Sec. 1, this analysis approach is intrinsically designed to infer the *total cost* (memory allocation in this case) of the program's execution and not to infer its peak consumption. This is because the equations accumulate the cost of all instructions and rules together as it can be observed in the CR for the example above.

3.3 Closed-Form Upper Bounds

Once the CR is generated, a cost analyzer has to use a CR solver in order obtain closed-form upper bounds, i.e., expressions without recurrences. The technical details of this process are not explained in the paper as our analysis does not require any modification to such part. In what follows, we rely on the CR solver of [3] (which can be accessed online through a web interface) to obtain closed-form upper bounds for our examples. The soundness of the overall analysis, as stated in the next theorem, requires that the equations generated as well as their closed-form upper bounds are sound.

THEOREM 3.4 (soundness [3]). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle) \cdot bc, tv_1\rangle \cdot A; h_1 \rightsquigarrow^* \langle \mathrm{q}, bc, tv_n\rangle{\cdot}A; h_n$*, then* $p(\bar{v}) \geq total(t)$ *where* $\bar{v} = \alpha(\bar{x}, tv_1, h_1)$. □

Observe that the trace t in the theorem represents an execution of procedure p for some specific input data (properly stored in tv_1 and h_1) where the first configuration corresponds to calling p and the last one to returning from that specific call. As already mentioned in Sec. 3.1, $\bar{v}$ denotes the size of the input data.

EXAMPLE 3.5. *Solving the equations of Fig. 4 results in the following closed-form upper bounds for* f, m, g *and* h*:*

$$\begin{array}{rl} m(n) = & (2^{\mathsf{nat}(n)} - 1)*(f(n)+size(\mathrm{Tree}^1)) \\ f(n) = & \log_2(\mathsf{nat}(n-1)+1)*(size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3))+ \\ & \log_2(\mathsf{nat}(n-1)+1)*size(\mathrm{Long}^4)) \\ f_c(n,a) = & \log_2(\mathsf{nat}(n-1)+1)*(size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3)) \\ f_d(i,a) = & \log_2(\mathsf{nat}(i-1)+1)*size(\mathrm{Long}^4)) \\ g(n) = & size(\mathrm{Integer}^2)+size(\mathrm{Integer}^3) \\ h(n) = & size(\mathrm{Long}^4) \end{array}$$

where the expression $\mathsf{nat}(l)$ *is defined as* $\max(l, 0)$ *to avoid negative evaluations. As expected, method* m *has an exponential memory consumption due to the two recursive calls, which in turn is multiplied by the allocation at each iteration (i.e., the consumption of* f *plus the creation of a* Tree *object). The solver indeed substitutes* $f(n)$ *by its upper bound shown below. The memory consumption of* f *has two logarithmic parts: the leftmost one corresponds to the first loop which accumulates the allocation performed along the execution of* $g(n)$*, the rightmost one corresponds to the second loop with the allocation of* $h(n)$*.* □

A fundamental observation is that the above upper bounds on the memory consumption can be tighter if one considers the effect of GC. For instance, a more precise upper bound for m can be inferred if we take into account that the memory allocated by f can be entirely garbage collected upon return from f. Likewise, the upper bound for f can be more precise if we take advantage of the fact that not all memory escapes from g. The goal of the rest of the paper is to provide automatic techniques to infer accurate bounds by taking into account the memory freed by scoped-GC.

4. Escaped Memory Upper Bounds

In a real language, GC removes objects which become *unreachable* along the program's execution. Given a configuration $A; h$, we say that an object $o = h(r)$ where $r \in \mathrm{dom}(h)$ is not reachable, if it cannot be accessed (directly or indirectly) through the variables table tv of any activation record in A. To develop our analysis, we assume a scoped-memory manager, which at the level of the source language, meets these conditions: (1) it reclaims memory *only* upon return from methods and, (2) it collects *all* unreachable objects which have been created *during* the execution of the corresponding method call.

In order to simulate the behavior of such garbage collector at the level of the corresponding rule-based bytecode, it is enough to assume that the memory manager reclaims memory only upon return from procedures that correspond to methods but not from procedures that correspond to intermediate states like $\mathrm{f_c}$ and $\mathrm{f_d}$. We use $\rightsquigarrow_{gc}$ to denote $\rightsquigarrow$-transitions with a scoped-memory manager which meets the two conditions above. In this context, the *escaped memory* of a procedure execution is defined as follows. Given a trace $t \equiv \langle q, p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle q, bc, tv_n \rangle \cdot A; h_n$ whose first configuration corresponds to calling p and the last one to returning from that specific call, the escaped memory of t is $escaped(t) = size(h_n) - size(h_1)$, which corresponds to the amount of memory allocated during the execution of p and still live in the memory upon exit from p. Our first contribution is an automatic technique to infer *escaped memory upper bounds*.

4.1 Inference of Escape Information

We say that an object *escapes* from a procedure p, in the context of a scoped-memory manager, if it is created during the execution of p, and still available in the heap upon exit from p. Note that if p corresponds to an intermediate procedure, such object might be unreachable but still has not been garbage collected because GC is applied only when exiting from procedures that correspond to methods in the original program. As a preprocessing phase, for each procedure p, we need to over-approximate the set of allocation instructions "new c^i" that might be executed when calling p and its transitive calls such that it is guaranteed that *all* objects they create are not in memory upon exit from p, i.e., they have been garbage collected. Recall that an allocation instruction "new c^i" is uniquely identified by the *tagged* class c^i. We use the notation $A \setminus B$ for the difference on sets.

DEFINITION 4.1 (collectable objects). *Given a procedure* p*, we denote by* $collectable(\mathrm{p})$ *the set of all allocation instructions, identified by their* tagged *classes, defined as follows.*

$c^i \in collectable(\mathrm{p})$ *iff the following conditions hold:*

1. *"*new c^i*" is a reachable instruction from* p*;*
2. *for any trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$*, it holds that* $\forall r \in \mathrm{dom}(h_n) \setminus \mathrm{dom}(h_1)$ *the object* $h_n(r)$ *is not an instance of* c^i*.* □

The set of collectable objects can be approximated from the information computed by escape analysis [15, 4]. The goal of escape analysis is to determine all program points where an object is reachable and whether the lifetime of the object can be proven to be restricted only to the current method. In our implementation, we use the approach described in [21] which, as our experiments show, behaves well in practice.

EXAMPLE 4.2. *The escape information is computed for all procedures (both methods and intermediate rules) defined in Fig. 2:*

$$collectable(\mathrm{m}) = collectable(\mathrm{f}) = \{\mathrm{Integer}^2, \mathrm{Integer}^3, \mathrm{Long}^4\}$$
$$collectable(\mathrm{f_c}) = collectable(\mathrm{g}) = \{\mathrm{Integer}^2\}$$
$$collectable(\mathrm{f_d}) = collectable(\mathrm{h}) = \emptyset$$

As an example, the information in the set $collectable(\mathrm{f})$ *states that the objects created with class tags* $\mathrm{Integer}^2$*,* $\mathrm{Integer}^3$ *and* Long^4 *during the execution of* f *by the transitive calls to* g *and* h*, do not escape from* f*. Also,* $collectable(\mathrm{f_d}) = \emptyset$ *means that the object* Long^4 *created in* h *might escape from* $\mathrm{f_d}$*. An important observation is that this object is not reachable upon exit from* $\mathrm{f_d}$*, but since GC is applied only upon exit from procedures that correspond to methods, it will be collected only upon exit from* f*. This issue will be further discussed in Sec. 6.* □

4.2 Upper Bounds on the Escaped Memory

Intuitively, our technique to infer upper bounds on the escaped memory consists of two steps. In the first step, we generate equations for the total allocation (exactly as stated in Def. 3.2) which accumulate *symbolic* expressions of the form $size(c^i)$ to represent the heap allocation for the instruction new c^i, rather than its concrete allocation size. From these equations, we obtain an upper bound for the total memory allocation as a symbolic expression which contains residual $size(c^i)$ sub-expressions. The main novelty is that, in a second step, we tighten up such total allocation upper bound to extract from it only its escaped memory as follows. Given a procedure p, and its total heap consumption upper bound $p(\bar{x})$, we obtain the upper bound on the escaped memory by replacing expressions of the form $size(c^i)$ by 0 if it is guaranteed that all corresponding objects are not available in the memory upon exit from p, namely $c^i \in collectable(\mathrm{p})$. Given an expression exp and a substitution σ from sub-expressions to values, $exp[\sigma]$ denotes the application of σ on exp.

DEFINITION 4.3 (escaped memory upper bound). *Given a procedure* p*, its escape information* $collectable(\mathrm{p})$*, and its (symbolic) upper-bound for the total memory allocation* $p(\bar{x}) = exp$*, the escaped memory upper-bound of* p *is defined as:* $\check{p}(\bar{x}) = exp[\forall c^i \in collectable(\mathrm{p}).size(c^i) \mapsto 0]$*.* □

Observe that, in the above definition, it is required that the set $collectable(\mathrm{p})$ contains the information for objects created in transitive calls from p, as stated in Def. 4.1, because escaped memory upper-bounds for a method p are obtained by using only the information in $collectable(\mathrm{p})$ and not in any other $collectable(\mathrm{q})$ with $\mathrm{q} \neq \mathrm{p}$. This is an essential difference w.r.t. existing work [3] which does not compute information for transitive calls, but instead computes the escape information only for the objects which are created inside each method (excluding its transitive calls). We obtain strictly more accurate bounds as the following example illustrates.

EXAMPLE 4.4. *Applying Def. 4.3 to the total heap allocation inferred in Ex. 3.5, by using the escape information of Ex. 4.2, results in the escaped memory upper bounds:*

$$\begin{array}{ll} \check{m}(n) = (2^{\mathsf{nat}(n)} - 1) * size(\mathrm{Tree}^1) & \check{f}(n) = 0 \\ \check{f}_c(n,a) = \log(\mathsf{nat}(n-1)+1) * size(\mathrm{Integer}^3) & \check{g}(n) = size(\mathrm{Integer}^3) \\ \check{f}_d(i,a) = \log(\mathsf{nat}(i-1)+1) * size(\mathrm{Long}^4) & \check{h}(n) = size(\mathrm{Long}^4) \end{array}$$

We can see that the escaped memory upper bound for m *does not accumulate the allocations of* Long^4 *nor* $\mathrm{Integer}^2$ *and* $\mathrm{Integer}^3$ *objects because they do not escape from* f. *In [3], the allocations corresponding to* $\mathrm{Integer}^3$ *and* Long^4 *are accumulated because they escape from the method where these objects have been created. The problem is that in [3] they are accumulated in the CR and hence in all upper bounds for methods that transitively invoke* g *and* h. □

The following theorem states the soundness of our escaped memory upper bounds.

THEOREM 4.5 (soundness). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$*, then* $\check{p}(\bar{v}) \geq escaped(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$.

Proof.
(sketch) First, by Theorem 3.4, we have the soundness of the total allocation upper bound $p(\bar{v}) \geq total(t)$. Second, by the soundness of escape analysis [4], we know that $collectable(\mathrm{p})$ gives a safe approximation of the objects that escape from t. Now, by combining both parts, we have that $\check{p}(\bar{v}) \geq escaped(t)$ and, hence, the soundness of $\check{p}(\bar{v})$ follows. □

5. Live Heap Space Analysis

This section presents a novel live heap space analysis for garbage-collected languages which obtains precise upper bounds including logarithmic, exponential, etc. complexity classes. Achieving accuracy is crucial because live heap bounds represent the minimum amount of memory required to execute a program.

5.1 The Notion of Peak Consumption

Essentially, given a trace $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \leadsto^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$, the peak consumption can be defined as $peak(t) = max(size(h_2), \ldots, size(h_n)) - size(h_1)$. We decrement $size(h_1)$ because the objects created in an outer scope (i.e., those in h_1) cannot be collected during the execution t, as stated in condition (2) of scoped-GC in Sec. 4.

Let us illustrate this notion by means of this simple method "void r() {A; p(); B; q(); C;}" whose memory consumption is showed in Fig. 5. A, B and C are sequences of instructions that do not contain any method invocation. We use the notation $\hat{p}$ to the note the peak consumption of executing the method p. We can observe that the peak heap consumption $\hat{r}$ is the maximal of three possible scenarios: (1) In the leftmost column, we depict a scenario where we allocate A and then execute p, thus we add the peak heap consumption of p. (2) In the next alternative scenario, we still have A and then return from p's execution, thus we add the memory escaped upon return from p (i.e., $\check{p}$) and we continue until the execution of q. Hence we add B plus the peak of q. (3) In the next column, we have A, plus the memory escaped from p, plus B, plus the memory escaped from q, plus C. Observe that any of these scenarios may correspond to the actual peak and we need to infer upper bounds for all of them and then take the maximal. The rightmost column indicates the upper bound for total allocation which is clearly much less accurate.

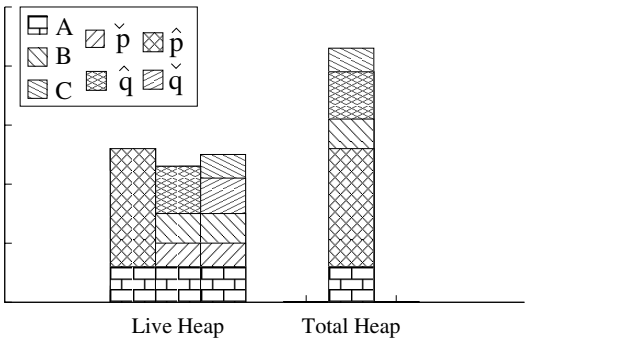

Figure 5. Memory Consumption of simple program

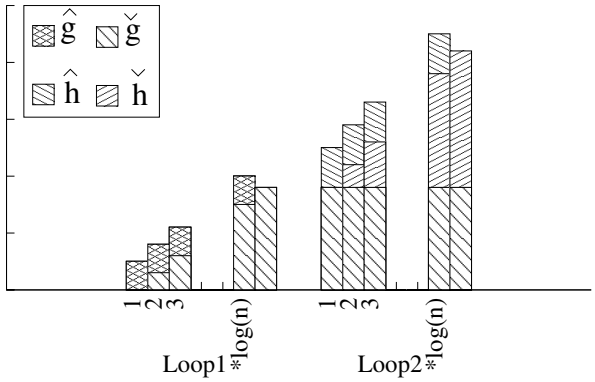

Figure 6. Memory Consumption of running example

In general the problem is more complicated, e.g., when method invocations occur within loops. Fig. 6 depicts the actual memory consumption of the execution of method f in our running example. Column 1 captures the heap allocation of executing g at the first iteration of the first loop (defined by procedure $\mathrm{f_c}$). Column 2 represents the escaped memory from g plus the next iteration of the loop where g allocates again $\hat{\mathrm{g}}$ memory and so on. As the loop in $\mathrm{f_c}$ is executed $\log(n)$ times we have all such possible scenarios over the tag Loop 1. Then, we start the execution of the second loop with an initial heap usage of $log(n)$ times the memory escaped from g. Similarly, at each iteration of the second loop, method h is invoked which allocates a maximal of memory $\hat{\mathrm{h}}$ and upon return, we need to consider the escaped memory from h plus the next execution. As the loop is executed $\log(n)$ times, we have all possible scenarios to the right grouped over the tag Loop 2. The peak heap allocation of executing f is the maximal of all such scenarios, namely the maximal between the two scenarios marked with $*$. The important point is that we need to infer upper bounds for $\hat{\mathrm{h}}$, $\hat{\mathrm{g}}$, $\check{\mathrm{h}}$, $\hat{\mathrm{g}}$ and generate as peak heap consumption the expression $\hat{\mathrm{f}} = \max(\hat{\mathrm{g}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{g}}, \hat{\mathrm{h}} + (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \log(\mathrm{n}) * \check{\mathrm{g}})$. Note that, in principle, it could happen that $\hat{\mathrm{g}} > (\log(\mathrm{n}) - 1) * \check{\mathrm{h}} + \hat{\mathrm{h}}$.

5.2 Peak Consumption Cost Relation

We now propose a novel approach for generating CR that, by relying on the escaped memory bounds, capture the peak heap consumption by considering all possible states of a program's execution. Our proposal is based on the following intuition: Let m_1 and m_2 be two methods, and let $\hat{m}_1(\bar{x}_1)$ and $\hat{m}_2(\bar{x}_2)$ be the peak heap consumption of executing m_1 and m_2 respectively, then the peak heap consumption of the two consecutive calls $\mathrm{m}_1;\mathrm{m}_2$ is $\max(\hat{m}_1(\bar{x}_1), \check{m}_1(\bar{x}_1) + \hat{m}_2(\bar{x}_2))$. The following definition generalizes this idea for an arbitrary sequence of statements.

DEFINITION 5.1 (peak consumption CR). *Consider a rule* $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ *and its corresponding size relations* φ_r. *Then, its* peak consumption *equation is* $\check{p}(\bar{x}) = \mathcal{T}(b_1, \ldots, b_n),\ \varphi_r$ *where* $\mathcal{T}$ *is defined as follows:*

$$\mathcal{T}(b_1,\ldots,b_n)::= \begin{cases} 0 & \text{if } n = 0 \\ \max(\hat{q}(\bar{x}_1), \check{q}(\bar{x}_1) + \mathcal{T}(b_2,\ldots,b_n)) & \text{if } b_1 = \mathrm{q}(\langle \bar{\mathrm{x}}_1 \rangle, \langle \bar{\mathrm{y}}_1 \rangle) \text{ is a call} \\ \mathcal{M}^t(b_1) + \mathcal{T}(b_2,\ldots,b_n) & \text{if } b_1 \text{ is an instruction} \end{cases}$$

Given a program P, *we denote by* $\hat{\mathcal{S}}_P$ *the* peak consumption cost relation *generated for each rule in* P. □

In the above definition, it can be observed that, in the second case, we generate two possible scenarios not only for methods, but also for intermediate procedures. These scenarios correspond to either the peak of the first procedure call or to the escaped memory from the first procedure call plus the peak of the rest of the instructions sequence. Considering the two scenarios at the level of procedures (no only of methods) allows us to gain further accuracy in situations, like in the method f, in which intermediate procedures correspond to loops which contain method invocations. The next example illustrates this point.

EXAMPLE 5.2. *The peak consumption* CR $\hat{\mathcal{S}}_P$ *of the rule-based program is different from the one in Fig. 4 in equations (1), (3), (4) and (6) which are now as follows:*

$$\begin{array}{lll} (1) & \hat{m}(n) = size(\mathrm{Tree}^1) + \max(\hat{m}(s_1), \check{m}(s_1) + \max(\hat{m}(s_2), \\ & & \check{m}(s_2) + \max(\hat{f}(n), \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3)))) \\ (3) & \hat{f}(n) = \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(i,a')) \\ (4) & \hat{f}_c(n,a) = \max(\hat{g}(n), \check{g}(n) + \hat{f}_c(n',a')) \\ (6) & \hat{f}_d(i,a) = \max(\hat{h}(i), \check{h}(i) + \hat{f}_d(i',a')) \end{array}$$

with the same constraints as those of Fig. 4. We can now replace the escaped memory upper bounds $\check{g}$, $\check{h}$, $\check{m}$ *and* $\check{f}$ *by the ones in Ex. 4.4. As an optimization, we do not apply the transformation to the last call in the rules, for instance, to the call to* $init$ *in equation (1), since trivially* $\hat{init} \geq \check{init}$. *Observe that in equation (3) we have applied also two possible scenarios to the intermediate procedure* f_c *which does not correspond to a method by introducing the max operator. This is essential to keep the two possible peaks (marked with "*" in the figure) separate instead of accumulating both of them, which would lead to a larger, less accurate upper bound. Besides, it is sound w.r.t. scoped-GC because the corresponding escaped memory bounds for* $\check{f}_c$ *and* $\check{f}_d$ *are obtained by considering that GC takes place upon method's return only.*

The most important point is that equation (4) accurately captures the memory consumption of all scenarios in Loop 1 of Fig. 6 and equation (6) captures those in Loop 2 to the right of the figure, as it will become clear after solving the equations in Ex. 5.3. □

An important feature of our CR $\hat{\mathcal{S}}_P$ is that they can still be solved by relying on a standard upper bound solver for CR produced by cost analysis like the one in [3]. The only adjustment is that our CR use the $\max$ operator which is frequently not supported. This is handled by a further preprocessing which transforms one equation that uses $\max$ into an equivalent set of equations that do not use $\max$ by creating nondeterministic equations whenever we have $\max$. In particular, an equation of the form $p(\bar{x}) = A + max(B, C), \varphi$ is translated into the two equations $p(\bar{x}) = A + B, \varphi$ and $p(\bar{x}) = A + C, \varphi$. Since an upper bound solver looks for an upper bound for all possible paths, it is guaranteed that this transformation simulates the effect of the $\max$ operator. Nested $\max$ are translated iteratively. For instance, the translation of equation (1) in Ex. 5.2, results in the following equations:

$$\begin{array}{l} \hat{m}(n) = size(\mathrm{Tree}^1) + \hat{m}(s_1), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \hat{m}(s_2), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \hat{f}(n), \varphi_1 \\ \hat{m}(n) = size(\mathrm{Tree}^1) + \check{m}(s_1) + \check{m}(s_2) + \check{f}(n) + \hat{init}(s_0,s_1,s_2,s_3), \varphi_1 \end{array}$$

EXAMPLE 5.3. *Solving the transformed equations results in the following closed-form upper bounds:*

$$\begin{array}{lll} \hat{m}(n) & = 2^{\mathsf{nat}(n)} * size(\mathrm{Tree}^1) + \hat{f}(n) \\ \hat{f}(n) & = \max(\hat{f}_c(n,a), \check{f}_c(n,a) + \hat{f}_d(n,a')) \\ \hat{f}_c(n,a) & = (\log(\mathsf{nat}(n-1)+1) + 1) * size(\mathrm{Integer}^3) + size(\mathrm{Integer}^2) \\ \hat{f}_d(i,a) & = (\log(\mathsf{nat}(i-1)+1) + 1) * size(\mathrm{Long}^4) \\ \hat{g}(n) & = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3) \\ \hat{h}(n) & = size(\mathrm{Long}^4) \end{array}$$

We can observe that the peak bound for f *accurately captures the maximal of the two scenarios in the figure: (1)* $\hat{f}_c(n,a)$ *corresponds to the leftmost column of Fig. 6 (since* $\check{g}$ *is* $size(\mathrm{Integer}^3)$ *which is accumulated* $log(n)-1$ *times and* $\hat{g}(n)$ *is* $size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3)$ *and (2)* $\check{f}_c(n,a) + \hat{f}_d(n,a')$ *corresponds to the rightmost column where, as expected, we accumulate* $log(n) - 1$ *times the escaped* $size(\mathrm{Long}^4)$ *object plus an additional one which is the peak consumption of* h *(and nothing escapes from* $\mathrm{f_c}$*).*

It is fundamental to observe the difference between the above live heap space bound for m *and the total allocation computed in Ex. 3.5. In our live bound, since the allocation required by* f *can be entirely garbage collected upon exit from* f*, the required heap is not proportional to the number of times that* f *is invoked (i.e., exponential on* n*) but rather the memory required for a single execution of* f. □

The following theorem states that the upper bounds computed by our analysis are *sound*, i.e., for any input values, they evaluate to a larger value than the actual peak consumption.

THEOREM 5.4 (soundness). *Given a procedure* p*, and a trace* $t \equiv \langle \mathrm{q}, \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) \cdot bc, tv_1 \rangle \cdot A; h_1 \rightsquigarrow^*_{gc} \langle \mathrm{q}, bc, tv_n \rangle \cdot A; h_n$ *then* $\hat{p}(\bar{v}) \geq peak(t)$ *where* $\bar{v} = \alpha(\bar{\mathrm{x}}, tv_1, h_1)$. □

6. Approximating the Ideal Garbage Collector

In this section, we show how the analysis of Sec. 5 can be refined to consider other GC schemes and, in particular, to get closer to the *ideal* GC manager where objects are collected as soon as they become unreachable. For instance, the peak consumption upper bound inferred in Ex. 5.3 for f is accurate when using a scope-GC scheme, since all objects created inside the loops are collected only upon exit from f. However, it is clearly inaccurate for an ideal GC scheme, since the lifetime of each object created in f is limited to one iteration of the corresponding loop, and therefore f can be executed in constant heap space.

Luckily, we can take advantage of scopes in the rule-based representation in order to infer accurate upper bounds for such GC schemes without modifying our analysis. In Def. 4.1 the effect of GC is considered only on exit from procedures that correspond to methods, this is essential in order to obtain safe upper bounds for scoped-GC, since in the original language GC is assumed to be applied upon exit from method scopes. However, the rule-based language distinguishes scopes that correspond to code fragments (in

the original program) smaller than methods, e.g., $\mathrm{f_c}$ and $\mathrm{f_d}$ respectively correspond to the first and second loop of f. Considering the effect of GC on exit from these (non-method) smaller scopes corresponds to applying more often GC than in the original language, and therefore getting closer to the ideal GC. In order to support this, we need to compute the set of collectable objects for blocks exactly as we do for methods in Def. 4.1. Let us see an example.

EXAMPLE 6.1. *If we apply GC upon exit from* $\mathrm{f_c}$*, then the collectable objects are* $collectable(\mathrm{f_c}) = \{\mathrm{Integer}^2, \mathrm{Integer}^3\}$*, and hence* $\hat{f}_c(n,a) = 0$*. Observe that in Ex. 4.2* $collectable(\mathrm{f_c})$ *contains only* $\mathrm{Integer}^3$*. This in turn improves the peak consumption for* f *to* $\hat{f}(n) = \max(\hat{f}_c(n,a), \hat{f}_d(n,a'))$*, which is clearly more precise than the one in Ex. 5.3.* □

Interestingly, the above upper-bound can be even further improved in order to obtain one which is as close as possible to the ideal behavior. Consider Rule (6) in Fig. 2 which corresponds to the second loop in f. The object created in h, and escaped to the calling context, becomes unreachable immediately after executing $\mathrm{intValue}_2$. Thus, if we separate the loop's body into a separate procedure $\mathrm{f'_d}$, we make this behavior observable to our analysis. This can be done by transforming the rules associated to the loops as follows:

(4) $\mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\quad \mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$
$\quad \mathrm{f_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle).$

$\mathrm{f'_c}(\langle \mathrm{n,a}\rangle, \langle \mathrm{n,a}\rangle) ::=$
$\quad \mathbf{n > 1},$
$\quad \mathrm{g}(\langle \mathrm{n}\rangle, \langle \mathrm{s_0}\rangle),$
$\quad \mathrm{intValue_1}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\quad \mathrm{a} := \mathrm{a} + \mathrm{s_0},$
$\quad \mathrm{n} := \mathrm{n}/2.$

(6) $\mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\quad \mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$
$\quad \mathrm{f_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle).$

$\mathrm{f'_d}(\langle \mathrm{i,a}\rangle, \langle \mathrm{i,a}\rangle) ::=$
$\quad \mathbf{i > 1},$
$\quad \mathrm{h}(\langle \mathrm{i}\rangle, \langle \mathrm{s_0}\rangle),$
$\quad \mathrm{intValue_2}(\langle \mathrm{s_0}\rangle, \langle \mathrm{s_0}\rangle)$
$\quad \mathrm{a} := \mathrm{a} * \mathrm{s_0},$
$\quad \mathrm{i} := \mathrm{i}/2.$

Now the peak consumption equations for $\mathrm{f_c}$ and $\mathrm{f_d}$ are:

$$\begin{array}{ll}
\hat{f}_c(n,a) = \max(\hat{f}'_c(n,a), \check{f}'_c(n,a) + \hat{f}_c(n',a')) & \{n{>}1, n'{=}n/2\}\\
\hat{f}_c(n,a) = 0 & \{n{\leq}1\}\\
\hat{f}_d(i,a) = \max(\hat{f}'_d(i,a), \check{f}'_d(i,a) + \hat{f}_d(i',a')) & \{i{>}1, i'{=}i/2\}\\
\hat{f}_d(i,a) = 0 & \{i{\leq}1\}\\
\hat{f}'_c(n,a) = size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3) & \\
\hat{f}'_d(i,a) = size(\mathrm{Long}^4) &
\end{array}$$

and, since $\check{f}'_c(n,a){=}\check{f}'_d(i,a){=}0$, solving them results in

$$\begin{array}{l}
\hat{f}_c(n,a){=}size(\mathrm{Integer}^2){+}size(\mathrm{Integer}^3)\\
\hat{f}_d(i,a){=}size(\mathrm{Long}^4)
\end{array}$$

which in turn improves the upper bound of f to

$$\hat{f}(n) = max(size(\mathrm{Integer}^2) + size(\mathrm{Integer}^3), size(\mathrm{Long}^4))$$

which is indeed the minimal amount of memory required in order to execute f in the presence of an ideal GC.

In order to support such transformations, one should guide the transformation from the bytecode to the rule-based program by the information previously computed on the lifetime of the different objects. Such analysis should give us indications about when it is profitable to make smaller scopes. Currently, we do this transformation only for scopes that correspond to loops. Also, it should be noted that there is an efficiency versus accuracy trade-off here, as we generate more equations in this case which thus will be more expensive to solve. Note that the same ideas are useful for supporting region-based memory management. The idea is to infer regions and use this information to separate the scopes, such that the exit from scopes coincides with the removal of the corresponding region.

7. Experiments

In this section, we assess the practicality of our proposal on realistic programs, the standardized set of benchmarks in the JOlden suite [12]. This benchmark suite was first used by [7] in the context of memory usage verification for a different purpose, namely for checking memory adequacy w.r.t. given specifications, but there is no inference of upper bounds as our analysis does. It has been also used by [5] for our same purpose, i.e., the inference of peak consumption. However, since [5] does not deal with memory-consuming recursive methods, the process is not fully automatic in their case and they have to provide manual annotations. Also, they require invariants which sometimes have to be manually provided. In contrast, our tool is able to infer accurate live heap upper bounds in a fully automatic way, including logarithmic and exponential complexities.

The first column of Table 1 contains the name of the benchmark. For most examples, we analyze the method `main` which transitively requires the analysis of the majority of the methods in the package. Only in those benchmarks whose name appears in two different rows, we do not analyze the `main` but rather all those methods invoked within the `main` that we succeed to analyze. In particular, benchmarks Health(cV), Health(gR), Bh(cTD), Bh(eB), Voronoi(cP), and Voronoi(b) correspond, respectively, to methods `createVertex`, `getResults`, `createTreeData`, `expandBox`, `createPoints`, and `buildDelaunayTriangulation` in the corresponding packages. In benchmark Bh, we cannot obtain an upper bound for the method `stepSystem` which is invoked within `main`. The reason is that this method contains a loop whose termination condition does not depend on the size of the data structure, but rather on the particular value stored at certain locations within the data structure. In general, it is complicated to bound the number of iterations of this kind of loops. Basically, the same situation happens in the method `simulate` of benchmark Health. In Voronoi, we are able to analyze all methods when they are not connected together. Unfortunately, we cannot analyze the `main` which, first invokes the method `createPoints` which returns an object `point` and then invokes the method `point.buildDelaunayTriangulation` on such object. The problem is that the upper bound of `buildDelaunayTriangulation` depends on the size of the object `point` returned by `createPoints` and the size analysis is not able to propagate such relation. It should be noted that, in these three cases, the limitations are not related to our proposal in this paper but to external components which can be independently improved.

The second and third columns in the table show, respectively, the upper bounds for total allocation and for live heap space usage. Note that the cost model we use for the experiments substitutes the symbolic expressions $size(Obj)$ by their corresponding numerical values, so that the system can perform mathematical simplifications. In particular, the size of primitive types is $1, 2, 4$, etc. bytes respectively for byte, char, int, etc.; the size of a reference is set to 4 bytes; and the size of an object is the sum of the sizes of all its fields (including those inherited).[1]

Let us first explain the examples Tsp, Bisort, Health, TreeAdd, Perimeter and Voronoi which follow a similar pattern. Basically, they contain methods (in rule-based form) which have this shape $p(X) ::= alloc(k), p(Y_1), \ldots, p(Y_n)$, i.e., a certain allocation k is accumulated by several recursive calls to the method. The size of the arguments in the recursive calls decrease by half in examples Tsp, Bisort and Voronoi and there are two recursive calls. Thus, their resulting upper bounds are linear. In benchmarks Health, Perimeter and TreeAdd, the size of the argument decreases by a constant; the first two examples contain 4 recursive calls and the

[1] This is just an estimation. The sizes depend on the particular JVM

Bench	Total Allocation Upper Bounds	Live Heap Space Upper Bounds
Mst	$\texttt{nat(A+1)*nat}(\frac{\texttt{A}}{4}) + \texttt{33*nat(A+1)} + 8$	$\texttt{nat(A+1)} + 8 + \max(\texttt{nat(A+1)}^2 + \texttt{18*nat(A+1)} + \texttt{nat}(\frac{\texttt{A}}{4}) + 72, \texttt{nat(A+1)*nat}(\frac{\texttt{A}}{4}) + \texttt{25*nat(A+1)} + \texttt{2*nat}(\frac{\texttt{A}}{4}) + 48)$
Em3d	2*nat(D-1)*(32+nat(B)) + 2*nat(B) + 16*nat(C) + 2*nat(D) + 89	max(4*nat(B) + nat(C) + 2*nat(D) + 2*nat(D-1) + 153, 4*nat(B) + max(16,nat(C)) + 2*nat(D) + 2*nat(D-1) + 153), (34 + nat(B))*nat(D-1) + 6*nat(B) + 3*nat(D) + 313)
Bisort	4*nat(A) + 12*nat(B-1) + 52	max(4*nat(A),12*nat(B-1) + 36)
Tsp	46*nat(2*B-1) + 138	28*nat(2*B-1) + 84
Power	258544	5992
Health(cV)	$104*4^{\texttt{nat(A)}} + 416$	$104*4^{\texttt{nat(A)}} + 416$
Health(gR)	$28*4^{\texttt{nat(A-1)}} + 36$	$28*4^{\texttt{nat(A-1)}} + 36$
TreeAdd	$40*2^{\texttt{nat(B-1)}} + 4*\texttt{nat(A)} + 76$	$24*2^{\texttt{nat(B-1)}} + 60$
Bh(cTD)	96*nat(B) + 128	92*nat(B) + nat(B-1) + 308
Bh(eB)	96	92
Perimeter	$56*4^{\texttt{nat(B)}} + 4*\texttt{nat(A)} + 128$	$56*4^{\texttt{nat(B)}} + 112$
Voronoi(cP)	20*nat(2*A-1) + 60	20*nat(2*A-1) + 60
Voronoi(b)	$88*2^{\texttt{nat(A-1)}} + 8$	$88*2^{\texttt{nat(A-1)}} + 8$

Table 1. Upper bounds for Total Allocation and Live Heap Usage

latter one 2 recursive calls. Thus, their resulting upper bounds are exponential. The upper bounds for live heap and total heap for the methods in Health and Voronoi are the same. This happens because the analyzed methods are encharged of creating the data structures and there is no memory that can be garbage collected. In the remaining examples, the method `main` first calls the method `parseCmdLine` which creates a (linear) number of objects that do not escape to the `main` and, then, calls other methods that construct (and modify) a data structure which escapes to the `main`. The fact that some memory can be garbage collected explains that the live heap bounds are smaller than the total allocation. Tsp is interesting because some auxiliary `Double` objects are created during the execution of the methods `uniform` and `median` which do not escape from such methods and hence the difference between the live bound and the total allocation is bigger.

Benchmark Power has a constant memory consumption. Its live bound is much smaller than the total allocation because many objects are created by the constructor of `Lateral` which become unreachable and hence can be garbage collected. In the examples Mst and Em3d, most of the memory is allocated during the construction of a graph and all such memory cannot be garbage collected. As before, the live bound is slightly smaller because of the memory created by `parseCmdLine` which can be entirely garbage collected. Finally, the methods analysed for the benchmark Bh also create a number of auxiliary objects that can be garbage collected and the live heap bounds become tighter than the total allocation.

It is not easy to compare our upper bounds with those obtained by [5] since the cost models are different (we count sizes of objects as explained above while they count number of objects), they consider a region-based memory model while our analysis is developed for a scope-based model and, besides, for recursive methods (which occur in most benchmarks) [5] requires manual annotations that are not shown in their paper. In spite of these differences, as expected, our upper bounds coincide with those of [5] asymptotically (i.e., by ignoring the coefficients and constants).

An interesting experimentation that we plan to do for future work is to compare our upper bounds with actual observed values. This is however a rather complicated task. Note that it would require choosing particular inputs, and the memory consumption of the program could highly vary depending on such choice. We are confident about the positive results since, as we saw above, our UBs are coherent with those in [5], which in turn have already been compared to actual observed values.

8. Related Work

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [22]). Analysis of live heap space is different because it involves explicit analysis of all program states. Most of the work of memory estimation has been studied for functional languages. The work in [11] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager. The techniques proposed in [18, 17] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form upper bounds, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations.

For imperative object-oriented languages, related techniques have been recently proposed. Previous work on heap space analysis [3] cannot be used to infer upper bounds on the maximum live memory as their cost relation systems are generated to accumulate cost, as explained in Sec. 3. Their refinement to infer escaped memory bounds is strictly less precise than ours as explained in Sec. 4, besides, there is no solution there to infer peak consumption. Later work improves [3] by taking garbage collection into account. In particular, for an assembly language, [8] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided. For a Java-like language, the approach of [5] infers upper bounds of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach is more flexible as regards its adaptation to other GC schemes (see Sec. 6). We believe that

our system is the first one to infer upper bounds on the live heap consumption which are not restricted to simple complexity classes.

9. Conclusions and Future Work

We have presented a general approach to automatic and accurate live heap space analysis for garbage-collected languages. As a first contribution, we propose how to obtain accurate bounds on the memory *escaped* from a method's execution by combining the total allocation performed by the method together with information obtained by means of escape analysis. Then, we introduce a novel form of *peak consumption cost relation* which uses the computed escaped memory bounds and precisely captures the actual heap consumption of programs' execution for garbage-collected languages. Such cost relations can be converted into closed-form upper bounds by relying on standard upper bound solvers. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection. We first develop our analysis under a scoped-memory management which reclaims memory on method's return. The amount of memory required to run a method under such model can be used as an over-approximation of the amount required to run it in the context of an ideal garbage collection which frees objects as soon as they become dead. We also show how to approximate such ideal behavior with our analysis. For future work, we also plan to consider how to adapt our techniques to region based memory management [16, 6].

Finally, the idea developed in Sec. 5 can be used to estimate other (non accumulative) resources which require to consider the maximal consumption of several execution paths. For example, it can be used to estimate the maximal height of the frames stack as follows. Given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_n$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \max(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_1}(\bar{x}_{i_k}))\ \ \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replace their corresponding 1's by 0's. This is a subject for future work.

Acknowledgments

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* ,TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

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

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Post-proceedings of Formal Methods for Components and Objects (FMCO'07)*, number 5382 in LNCS, pages 113–133. Springer-Verlag, October 2008.

[3] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

[4] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[5] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[6] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In David F. Bacon and Amer Diwan, editors, *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 85–96. ACM, 2004.

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

[8] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *Proceedings of the International Symposium on Memory management (ISMM)*, New York, NY, USA, 2008. ACM.

[9] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM Press, 1978.

[10] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[11] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[12] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

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

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

[15] Y. G. Park and B. Goldberg. Escape analysis on lists. In *PLDI*, pages 116–127, 1992.

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[17] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. of LCTES/OM*, pages 102–111. ACM, 2001.

[18] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[19] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[20] B. Wegbreit. Mechanical Program Analysis. *Comm. of the ACM*, 18(9), 1975.

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

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.