Automatic Removal of Array Memory Leaks in Java

Ran Shaham[1,2*], Elliot K. Kolodner[2], and Mooly Sagiv[1]

[1] Tel-Aviv University
{rans,sagiv}@math.tau.ac.il
[2] IBM Haifa Research Laboratory
kolodner@il.ibm.com

Abstract. Current garbage collection (GC) techniques do not (and in general cannot) collect all the garbage that a program produces. This may lead to a performance slowdown and to programs running out of memory space.
In this paper, we present a practical algorithm for statically detecting memory leaks occurring in arrays of objects in a garbage collected environment. No previous algorithm exists. The algorithm is conservative, i.e., it never detects a leak on a piece of memory that is subsequently used by the program, although it may fail to identify some leaks. The presence of the detected leaks is exposed to the garbage collector, thus allowing GC to collect more storage.
We have instrumented the Java virtual machine to measure the effect of memory leaks in arrays. Our initial experiments indicate that this problem occurs in many Java applications. Our measurements of heap size show improvement on some example programs.

1 Introduction

Java's run-time GC does not (and in general cannot) collect all the garbage that a program produces. GC typically collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again, even though they are reachable. This may lead to a performance slowdown and to programs running out of memory space. This may also have a negative effect on Java usability.

1.1 A Running Example

A standard Java implementation of a stack data structure is shown in Figure 1(a). After a successful `pop`, the current value of `stack[top]` is not subsequently used. Current garbage collection techniques fail to identify memory leaks of this sort; thus, storage allocated for elements popped from the stack may not be freed in a timely manner. This example class serves as the running example throughout this paper.

* Supported in part by the U.S.-Israel BSF under grant 96-00337

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 50–66, 2000.

1.2 Existing Solutions

A typical solution to avoid these memory leaks is to explicitly assign `null` to array elements that are no longer needed. For example, a stack implementation, which avoids these leaks is shown in Figure 1(b), where `null` is explicitly assigned to `stack[top]`.

```
public Class Stack {
  private Object stack[];
  private int top;
  public Stack(int len) {
    stack = new Object[len];
    top = 0;
  }
  public synchronized Object pop() {
    if (0 < top) {
        top--;
s:      return stack[top];
    }
    throw new ESExc();
  }
  public synchronized
              void push(Object o) {
    if (top < stack.length) {
s':     stack[top]=o;
        top++;
        return;
    }
    throw new FSExc();
  }
  public synchronized void print() {
    for (int i=0; i<top; i++) {
s'':  System.out.println(stack[i]);
    }
  }
}
```

(a)

```
public Class Stack {
  private Object stack[];
  private int top;
  public Stack(int len) {
    stack = new Object[len];
    top = 0;
  }
  public synchronized Object pop() {
    if (0 < top) {
        Object tmp;
        top--;
        tmp = stack[top];
        stack[top]=null;
        return tmp;
    }
    throw new ESExc();
  }
  public synchronized
              void push(Object o) {
    if (top < stack.length) {
        stack[top]=o;
        top++;
        return;
    }
    throw new FSExc();
  }
  public synchronized void print() {
    for (int i=0; i<top; i++) {
      System.out.println(stack[i]);
    }
  }
}
```

(b)

Fig. 1. (a) The runing example, Stack class. (b) With explicitly assigning `null`. (`ESExc` and `FSExc` are subclasses of `RuntimeException`)

Such solutions are currently being employed in the JDK library, e.g., in the `jdk.util.Vector` class and by some "GC-aware" programmers. These solutions have the following drawbacks:

- Explicit memory management complicates program logic and may lead to bugs; by trying to avoid memory leaks, a programmer may inadvertently free an object prematurely.
- GC considerations are not part of the program logic; thus, they are certainly not a good programming practice. In fact, the whole idea of GC aware programs defeats some of the purposes of automatic GC.
- The solution of explicitly assigning `null` may slow the program, since such `null` assignments are performed as part of the program flow. For example, consider the method `removeAllElements` of class `java.util.Vector` shown in Figure 2(b). The only reason for the loop is to allow GC to free the array elements. In contrast, our compile-time solution eliminates the need for such a loop. The method can be rewritten as shown in Figure 2(a); thus, at least `elementCount` instructions are saved. In Section 5 we give a potential interface to GC, which will allow unit-time operation in this case.

```
void removeAllElements() {
   elementCount=0; }
```

(a)

```
void removeAllElements() {
   for (int i=0; i < elementCount; i++)
      elementData[i]= null;
   elementCount=0; }
```

(b)

Fig. 2. (a) A "clean" implementation. (b) "GC-aware" implementation

Consider the `Vector` class in the `java.util` package, which implements a dynamic array of objects. Though it has already been instrumented (in Sun's implementation) with assignment to `null` in appropriate places in order to avoid leaks, it suffers from some of the limitations outlined above. Furthermore, our experimental results show that instead of using a "standard" implementation of such abstract data types (ADTs), programmers use a "tailored" implementation in many cases, due to considerations such as speed, or strong typing. Examples include rewriting a non-synchronized version of `Vector` or a well-typed version of `Vector`, maintaining only objects of a specific class. There are some Java language extensions for parameterized types, e.g., [3], being considered, and work showing how to reduce the cost of synchronization, e.g., [2], which may eliminate the need for some of these tailored implementations. Nevertheless, the above limitations lead us to conclude that programmers should be freed from dealing with these memory management considerations and that the leaks should be detected by automatic means, e.g., by compiler analyses.

1.3 Main Results and Related Work

Section 2 presents our motivating experiments for showing that array memory leaks pose a real problem, and that the problem is worth solving, performance-wise. We performed a simple string search on Java source files and found some

occurrences of the array memory leak problem. For several programs we also measured the potential benefit of solving the problem and found that there are cases where there is a significant saving of memory.

This research was inspired by work on liveness analysis for Java for local variables holding references [1]. This liveness analysis leads to a reduced *root set*, enabling more memory to be reclaimed. However, such techniques are not applicable in general to arrays of objects. Treating an array as a single reference variable yields an overly conservative result; an array represents a set of references, where every array element is a potential reference, while a reference variable represents only one potential reference. For example, the field `stack` in the Stack class is *live* after `s`, but the location denoted by `stack[top]` is *dead* after `s`.

Identifying liveness requires flow sensitive analysis that may take non-linear time and could fail to scale for large programs. Moreover, due to the capability of Java to load classes in run-time, not all the code is necessarily available even when a program starts running. Therefore, our algorithm can operate on Java bytecode and analyze one class at a time by conservatively approximating potential method invocations. Despite these conservative assumptions, our algorithm is capable of finding memory leaks in many interesting cases, including the implementation of various array-based ADTs, e.g., dynamic arrays, stacks and cyclic queues. Indeed, we believe that this will allow our algorithm to scale for large programs, while locating most of the leaks in well written programs that make use of private or protected fields for encapsulation. In Section 3 we briefly discuss the *approximated supergraph* to allow a simple class level analysis of Java.

In Section 4, we give an algorithm for identifying live regions of arrays. Technically, identifying live array regions is more complex than the problem of identifying live scalars, since in many cases it is necessary to identify relationships between index variables. In the `print` method of the running example, knowing that `i` is less than `top` before s'' is important in order to determine that elements of stack beyond `top` cannot possibly be used in the `println` invocation. Relationships between variables have also been used to analyze array accesses for parallelizing compilers and in the context of other array reference analyses (e.g., [11,17]). These techniques can also be extended to detect the minimal and maximal values used as array indices; this allows the removal of checks for array bound violations [8,9]. One of the most precise methods was proposed by Cousot and Halbwachs [6]; it automatically identifies linear relationships between variables by scanning the control flow graph in a forward direction. In [13] it is shown that this general technique can be also used to analyze live array regions. Our chief observation is that live array regions can be also represented using linear relationships between variables.

We show how the result of a forward direction dataflow analysis, which identifies relationships between variables, is integrated into a backward analysis of the control flow, which determines the live regions of the array.

Both phases use the *constraint graph* suggested in [4, Chapter 25.5, pp.539–543] as a simple representation of program variable relationships. The constraint graph allows us to efficiently represent a special case linear relationship of the form $x \leq y+c$, where `x` and `y` are program variables and c is an integer constant. In [13] we explain how to handle more general sets of constraints and more interesting classes of programs.

In Section 5, we explain how a GC algorithm can exploit the results of our analysis algorithm. Our algorithm can also be applied to a Java program with potential leaks in order to determine the necessary `null` assignments. `null` assignment statements are added at program points where the array elements become dead. In the running example, our algorithm detects that before program point s, array element `stack[top]` is live, while after s, array element `stack[top]` is dead. Thus `stack[top]` can be assigned to `null`, as shown in Figure 1(b).

A prototype of the algorithm was implemented in Java, and used to find dead array regions for the running example in 0.21 CPU seconds. The prototype has no front-end, so the input to the prototype is the approximated supergraph of a class. The extended version of the algorithm as described in [13] was also implemented, and used to find live array regions for `java.util.Vector` class. Interestingly, the analysis located a bug in the method `lastIndexOf(Object elem, int index)` (to be fixed by Sun Microsystems). We noticed the bug due to the overly imprecise live array regions.

Some programming languages, such as CLU [10], provide built-in dynamic arrays that can be used to implement stacks and vectors. However, our algorithm can handle cases beyond dynamic arrays such as cyclic queues where the regions of live array elements are not necessarily continuous. Furthermore, our algorithm does not require extensions to the Java language.

2 Motivating Experiments

Frequency of the Problem The experiment to determine the frequency of occurrence was conducted before implementing the algorithm. Instead, we used lexical scanning of Java source files. We searched for classes having a field, which is an array of objects, and integer field(s), preferably containing the string "count" in their names. Also, we looked for methods containing the string "remove". The motivation for such searches, was to find re-implementations of the `java.util.Vector` class, keeping in mind that the methods like `removeAllElements` and `removeElementAt` use explicit assignments to `null` to prevent memory leaks.

About 5600 Java source files were scanned, including the *Java Development Kit* version 1.1.6 source files. In 1600 files, an array of objects is defined. In 20 files the problem was detected in 25 statements, i.e., several files contained more than one instance. Out of the 25 statements, 13 did not have the desired `null` assignment, i.e., they contained a potential memory leak.

Potential Benefits We conducted an experiment similar to that conducted in [1] using a modified JVM, in order to evaluate the potential benefits of removing leaks. We used Sun's JDK 1.2, Classic VM, as the basis. After every 100KB of allocation, we invoke the GC, perform all possible finalizations, and perform GC again. We calculate the heap size as a function of bytes allocated, sampled every 100KB. To simulate a potential memory leak, two versions of `java.util.Vector` class are used, the original one, and a version with leaks, i.e., without the explicit assignments to `null` in the `removeElementAt` and `removeAllElements` methods. Then we compared the allocation integral, calculated as the area under the heap size curve.

We measured the allocation integral in two programs from Spec JVM98 [15], *javac*, the Java compiler, and *db*, a benchmark simulating a database, on the original and modified Spec inputs. These are the only programs in the Spec suite using vectors. The measurements were done on a 400 MHz Intel Pentium-II CPU with 128MB of memory, running Windows NT 4.0.

In *javac* we obtain 1.35% average allocation integral improvement. The original input to *db* yields no improvement. However, using the modified input to *db*, which contains many delete commands, the improvement is 26.65%, concluding (as in [1]) that the expected main benefit is preventing bad surprises .

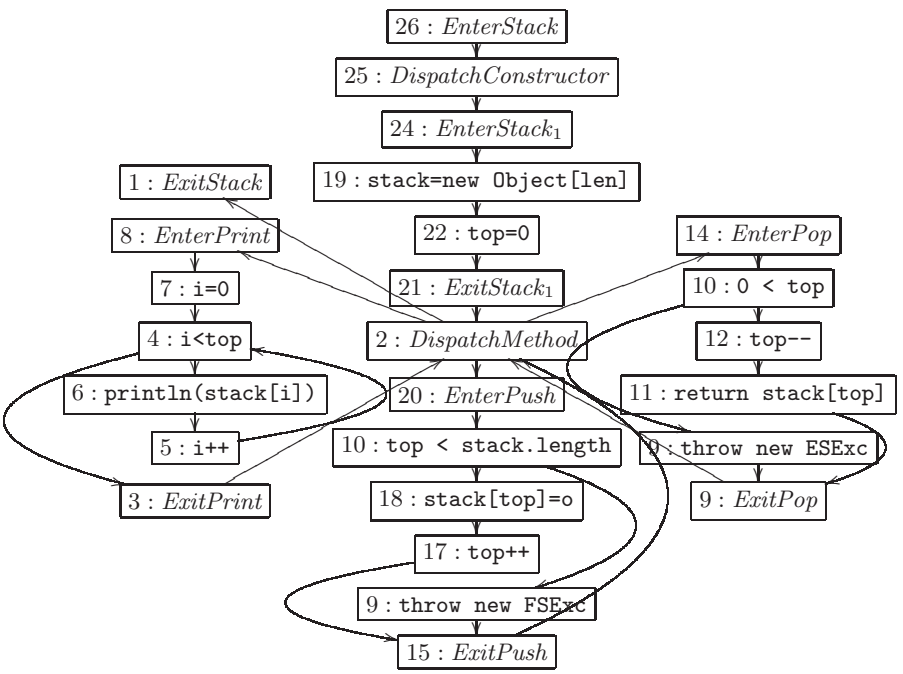


Fig. 3. The approximated supergraph of the running example.

3 Class Level Analysis of Java Programs

In this section we briefly discuss the *approximated supergraph* of a class. The main idea is to approximate the execution paths on all the class instances, while allowing the derivation of interesting information on objects encapsulated at the class level. Thus, paths that do not modify or use encapsulated data are not included in the approximated supergraph.

The *program supergraph*(see [14,12]) integrates the program call graph and the individual control flow graphs of each of the procedures in the program. To allow interprocedural class level analysis, we use the *approximated supergraph*. The approximated supergraph of the class C is an approximation of supergraphs occurring at any instance of C. Since the idea of approximated supergraph is not a core part of this work and due to space limitations, we choose not to elaborate on the subject. Details regarding the approximated supergraph including extensions to include the full spectrum of Java constructs are found in [13]. The approximated supergraph of the running example is shown in Figure 3.

Encapsulation at the class level is ensured by using *private* fields, local variables and method parameters, and by not allowing objects referenced by these variables to "escape" outside the class level scope. In the running example, `top` is a private field, `i` is local variable, `len` is a method parameter, `stack` is a private field, and in addition is not passed as a parameter or returned as a result, thus its referenced array can not escape outside the class level scope. Therefore, they are all encapsulated in `Stack`, and the analysis of `Stack` class using the approximated supergraph is conservative.

4 The Algorithm

In this section, we give an efficient algorithm for computing *liveness* information for arrays. In Section 4.1, we define the problem by extending the classical definition of liveness of scalar variables. In Section 4.2 we define constraint graphs that provide an efficient representation for special form of inequalities between index variables. In addition, we show that constraint graphs can represent liveness information. In Section 4.3 an iterative algorithm for identifying live regions for arrays at every supergraph node is given. This algorithm uses the constraint graph and the previously computed forward information, to obtain quite precise liveness information. This section is concluded in Section 4.4, in which we briefly describe the iterative forward algorithm, which computes inequalities between index variables at every supergraph node.

For expository purposes, we assume that the program supergraph contains one designated encapsulated array `A` of type `T[]`.

4.1 The Liveness Problem for Arrays

Recall that a scalar variable `var` is *live* before a program point p, if there exists an execution sequence in the program including p and a use of `var` such that

(i) p occurs before the use of `var` and (ii) `var` is not assigned between p and the use.

We now generalize this definition for arbitrary program expressions that evaluate to a location or reference (or equally have a defined L-value).

Definition 1. *An expression* `e` **is live before a program point** p*, if there exists an execution sequence,* $\pi_1.\pi_2$ *such that (i) the path* π_1 *ends at program point* p*, (ii)* `e` *denotes a location (or reference)* l *at the end of* π_1*, and (iii)* l *is used at the end of* π_2 *without prior assignment along* π_2*.*

In the running example (see Figure 1), the location denoted by `stack[top]` in s is live before s, but not before any other point in the class. For example, it is not live before the end of the method `pop` since on any sequence from that point to a usage of a location denoted by `stack[top]` in s, this location must be assigned a new value at *s'*. Indeed, the main idea in this definition is to allow the expression `e` to denote more than one location for different execution paths. In the running example, `stack[i]` is live before *s"* for all $0 \leq i < top$. This is the kind of information important for GC (see Section 5).

Notice that Definition 1 coincides with the classic liveness definition for a scalar variable and in this case l is the (activation record) location of the scalar.

4.2 The Constraint Graph

We now define the constraint graph, which efficiently represents inequalities between program variables. Operations on the inequalities are implemented by path calculations on the constraint graph.

Definition 2. *The* **constraint graph** *is a finite labeled directed graph with a set of vertices* V *of the encapsulated integer variable or field, including a special vertex* 0*, and in addition another special vertex, denoted* \$*, for representing liveness constraints to be discussed later. The constraint graph is captured by a* weight *function* $w: V \times V \rightarrow \mathbb{Z} \cup \{-\infty, \infty\}$*. Pictorially, we draw an edge from* v_1 *to* v_2 *if* $w(v_1, v2) < \infty$*.*

Such a directed graph w *represents the inequalities:*

$$\bigwedge_{x,y \in V} x \leq y + w(x, y) \tag{1}$$

This interpretation of w is used in the forward phase of the algorithm. The constraint graph, which represents the inequalities after supergraph node 5 (`i = i + 1`) of the running example, is shown in Figure 4. The -1 edge from 0 to i represents the inequality $0 \leq i + (-1)$, or $0 < i$. Usually, isolated vertices are omitted from the figures. An edge, which its weight is implied by the sum of the weights along a directed path in the graph connecting the source and the target vertices of that edge, is also not included in the figures.

The reader is referred to [4, Chapter 25.5, pp.539–543] for explanations on the properties of constraint graphs.

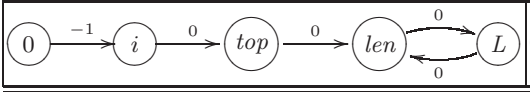
Constraint Graph	Inequalities Information
	$0 < i \leq top \leq len = L$

Fig. 4. The constraint graph after supergraph node 5. L stands for `stack.length`

Our chief insight is that live regions can also be represented using constraint graphs, with one additional designated vertex, denoted by \$. This node represents constraints on the live array indices of the array. For example, the constraint graph in the first row of Figure 5 corresponds to the liveness information $0 \leq \$ < top$. This constraint graph represents the fact that array elements `stack`$[0], \texttt{stack}[1], \ldots, \texttt{stack}[\texttt{top} - 1]$ are live. Another example is the constraint graph corresponding to a conditional live region which is presented in the second row of Figure 5. It signifies that array elements `stack`$[0], \texttt{stack}[1], \ldots,$ `stack`$[\texttt{top} - 1]$ are live, and in addition the live region is conditional because of the -1 edge connecting vertex i to vertex top. In other words, if $i \geq top$ then none of the array elements is alive. This can happen when the stack is empty and `top = 0`.

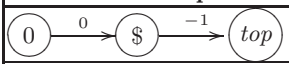
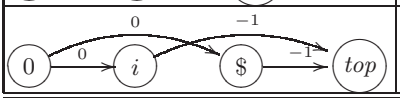
Constraint Graph	Liveness Information
	$\{stack[\$] \mid 0 \leq \$ < top\}$
	$\{stack[\$] \mid 0 \leq \$ < top, 0 \leq i < top\}$

Fig. 5. Constraint graphs representing liveness information

In general a constraint graph w represents the following liveness information:

$$\{A[\$] \mid \bigwedge_{x,y \in V} x \leq y + w(x,y)\} \tag{2}$$

where live regions come into play when either x or y are \$. This interpretation of w is used in the backward phase of the algorithm. The constraint graph in Figure 5 represents the liveness information:

$$\{stack[\$] \mid 0 \leq \$ + 0 \wedge \$ \leq top + (-1) \wedge 0 \leq i + 0 \wedge i \leq top + (-1)\} \tag{3}$$

The operations on constraint graphs are defined in Figure 6. The most basic operation is $TC(w)$, which yields a constraint graph whose edges are labeled with the shortest path (the least weight) between any two vertices. $TC(w)(x, y)$ corresponds to *strongest implied constraint* between x and y. We implemented $TC(G)$ using Floyd's all-pairs-shortest-path algorithm.

The constraint graph *represents a contradiction* if there exists a negative directed cycle in the graph. A contradiction represents either an infeasible execution path, or the fact that there are no future uses of the array. We denote these two cases by $\perp$, *dead(A)* respectively. The case of $w =$ *dead(A)* can only occur when w signifies liveness constraints. *eliminateNegativeCycles* detects contradicted constraint graphs, and $TC_{\perp}(w)$ extends $TC(w)$ by taking care of these special cases.

We define an order on constraint graphs to respect information order, i.e., $w_1 \sqsubseteq w2$ if $w_1 = \perp$, or $w_1 =$ *dead(A)* and $w_2 \neq \perp$ or for every $x, y \in V$, $w_1(x, y) \leq w_2(x, y)$.

For conviniency, our algorithm only operates on *closed* constraint graphs w, i.e., $TC_{\perp}(w) = w$. In this case, the order on closed constraint graphs is a *partial order* with *join*, $\sqcup$, (for merging information along different control flow paths), *meet*, $\sqcap$ (for partially interpreting program conditions) and *widening*, ∇ (for accelerating the termination of the iterative algorithm) operators shown in Figure 6. The operations in Figure 6 are exemplified in Section 4.3.

4.3 Backward Computation of Live Regions

In this section, we give an iterative algorithm for computing live regions in arrays. The algorithm operates on the approximated supergraph. The algorithm is *conservative*, i.e., the identified live regions must include "actual" live regions. When the iterative algorithm terminates, for every supergraph node n, and for every program point p that corresponds to n, if $\mathtt{A}[i]$ is live before p then i satisfies all the constraints in the constraint graph that the algorithm yields at n.

The algorithm starts by assuming that the array is dead at *Exit*$\langle$*Class*$\rangle$ supergraph node (*ExitStack* in the running example). Then it backward propagates liveness information along supergraph paths. The fact that the algorithm scans the supergraphs nodes in a backward direction may not come as a surprise, since the algorithm is an extension of the scalar variables liveness algorithm. Indeed, liveness information captures information about future usages.

Formally, the backward phase is an iterative procedure that computes the constraint graph $w_b[n]$ at every supergraph node n, as the least solution to the following system of equations:

$$\begin{aligned} w_b[n] &= \begin{cases} \textit{dead(A)} & n = \textit{Exit}\langle \textit{Class} \rangle \\ w_b[n] \nabla (w_f[n] \sqcap \bigsqcup_{m \in succ(n)} w_b[n, m]) & \text{otherwise} \end{cases} \\ w_b[n, m] &= [\![st(\langle n, m \rangle)]\!]^{\sharp}_{b_{\perp}} (w_b[m], w_f[n, m]) \end{aligned} \tag{4}$$

$$TC(w)(x,y) \stackrel{\text{def}}{=} \begin{cases} s & \text{the shortest path from } x \text{ to } y \text{ in } w \text{ is } s \\ \infty & \text{otherwise} \end{cases}$$

$$eliminateNegativeCycles(w) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{there exists a negative cycle in w - \{\$\}} \\ dead(A) & \text{there exists a negative cycle in w} \\ w & \text{otherwise} \end{cases}$$

$$TC_{\bot}(w) \stackrel{\text{def}}{=} \begin{cases} w & w \sqsubseteq dead(A) \\ eliminateNegativeCycles(TC(w)) & \text{otherwise} \end{cases}$$

$$w_1 \sqcup w_2 \stackrel{\text{def}}{=} \begin{cases} w_2 & w_1 \sqsubseteq w_2 \\ w_1 & w_2 \sqsubseteq w_1 \\ TC_{\bot}(\max(w_1, w_2)) & \text{otherwise} \end{cases}$$

$$w_1 \sqcap w_2 \stackrel{\text{def}}{=} \begin{cases} w_1 & w_1 \sqsubseteq w_2 \\ w_2 & w_2 \sqsubseteq w_1 \\ TC_{\bot}(\min(w_1, w_2)) & \text{otherwise} \end{cases}$$

$$w_{old} \nabla w_{new} \stackrel{\text{def}}{=} \begin{cases} w_{new} & w_{old} \sqsubseteq dead(A) \\ w' & \text{otherwise} \end{cases}$$

$$w'(x,y) \stackrel{\text{def}}{=} \begin{cases} w_{old}(x,y) & w_{old}(x,y) = w_{new}(x,y) \\ \infty & \text{otherwise} \end{cases}$$

$$w_S(x,y) \stackrel{\text{def}}{=} \begin{cases} \infty & S = \emptyset \\ c & S = \{x' \le y' + c\}, x = x', y = y' \\ \infty & S = \{x' \le y' + c\}, x \neq x' \vee y \neq y' \\ w_{S_1} \sqcap w_{S_2}(x,y) & S = S_1 \cup S_2 \end{cases}$$

Fig. 6. The utility functions used in the forward and backward analysis algorithms. w_S constructs a constraint graph corresponding to the strongest implied constraints in S

where $st(\langle n, m\rangle)$ is either the statement at supergraph node n or the condition holding along the arc $\langle n, m\rangle$, $[\![st(\langle n, m\rangle)]\!]^{\sharp}_{b_{\bot}}$ is defined in Figure 7, and $w_f[n], w_f[n, m]$ are given in Equation (5).

In the following subsections, we briefly explain the operators and the transfer functions used in the analysis:

Join Join is used when the supergraph flow splits (see Equation (4)). $\sqcup$ shown in Figure 6 yields the intersection of the (strongest implied) constraints occurring on all splitting supergraph paths, i.e., the maximal weight.

In Figure 8, the constraint graph after supergraph node 4, `i < top` is obtained by joining the constraint graphs before node 6, `println(stack[i])` and node 3, *ExitPrint*. The edge connecting vertex i to vertex \$ is not included, since it appears only in the constraint graph from `println(stack[i])` and not in the constraint graph from *ExitPrint*.

Integrating Forward Information The motivation to integrate forward information into the backward information comes from the observation that the

statement	$[\![\text{statment}]\!]^\sharp_{\mathbf{f}}(\mathbf{w})$	$[\![\text{statment}]\!]^\sharp_{\mathbf{b}}(\mathbf{w}, \mathbf{w_f})$
`i = j + c`	$w \sqcap w_{\{i \le j+c, j \le i+(-c)\}}$	$\begin{cases} dead(A) & w = dead(A) \\ TC_\perp(w') & \text{otherwise} \end{cases}$ $w'(x,y) = \begin{cases} \min(w(j,y), w(i,y)-c) & x = j \\ \min(w(x,j), w(x,i)+c) & y = j \\ \infty & x = i \vee y = i \\ w(e) & \text{otherwise} \end{cases}$
`A = new T[i + c]`	$[\![A.length = i + c]\!]^\sharp_f(w)$	$dead(A)$
`use A[i + c]`	w	$w \sqcup (w_f \sqcap w_{\{\$ \le i+c, i \le \$+(-c)\}})$
`use A[exp]`	w	$w_\emptyset$
`def A[i + c]`	w	$\begin{cases} dead(A) & w = dead(A) \\ TC_\perp(w') & \text{otherwise} \end{cases}$ $w'(x,y) = \begin{cases} c-1 & x = \$, y = i, w(x,y) = c \\ -c-1 & x = i, y = \$, w(x,y) = -c \\ w(x,y) & \text{otherwise} \end{cases}$
`i ≤ j + c`	(true arc) $w \sqcap w_{\{i \le j+c\}}$ (false arc) $w \sqcap w_{\{j \le i+(-1-c)\}}$	
Transfer Function Extension		
$[\![st(\langle n,m\rangle)]\!]^\sharp_{f_\perp}(w)$	$\stackrel{\text{def}}{=} \begin{cases} \perp & w = \perp \\ [\![st(\langle n,m\rangle)]\!]^\sharp_f(w) & \text{otherwise} \end{cases}$	
$[\![st(\langle n,m\rangle)]\!]^\sharp_{b_\perp}(w, w_f)$	$\stackrel{\text{def}}{=} \begin{cases} \perp & w = \perp \vee w_f = \perp \\ [\![st(\langle n,m\rangle)]\!]^\sharp_b(w, w_f) & \text{otherwise} \end{cases}$	

Fig. 7. The forward and backward transfer functions for simple statements. w_f is the forward information constraint graph occurring in the same supergraph node of w. $w_{\{\text{set of constraints}\}}$ is defined in Figure 6. `exp` is an arbitrary expression other than `i+c`. A condition node is interpreted along its true and false arcs. Transfer functions are extended to handle the case when w is $\perp$. Statement, which involves only constant expressions, can be handled simply by using a designated zero variable. Other statements are handled conservatively

liveness of an expression, `A`$[i]$, before supergraph node n, depends on two sources of information (see Definition 1): (i) The value of i on supergraph paths from node $Enter\langle Class\rangle$ leading to node n, which determines the location l denoted by `A`$[i]$, and (ii) The usage of location l on supergraph paths emanating from node n.

Therefore, integrating the (previously computed) forward information regarding the value of i and the backward information regarding the liveness of `A`$[i]$ can have a dramatic impact on the precision of the analysis.

Figure 9 shows the integration of the forward and backward information before node 6. Using the forward phase information, $0 \le i < top \le len = L$, leads to a precise liveness information, $\{stack[\$] | 0 \le \$ < top, 0 \le i < top\}$.

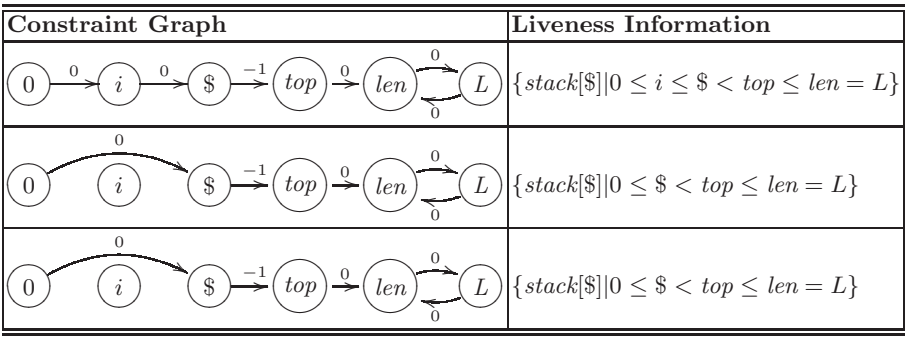


Fig. 8. The constraint graph before `println(stack[i])`, *ExitPrint*, and their join

The integration of forward and backward information is captured in Equation (4) by the $\sqcap$ operation.

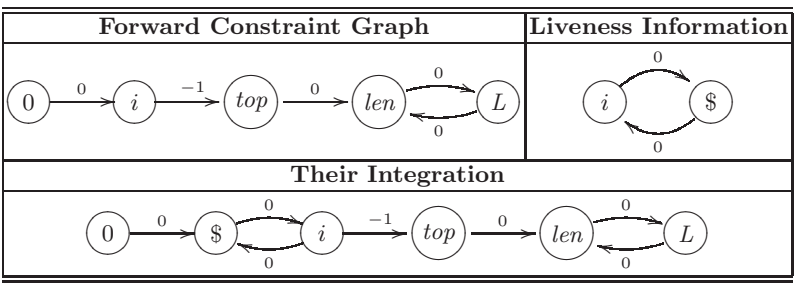


Fig. 9. The integrated constraint graph

Use of an Array Element For a statement using `A[i+c]`, the algorithm enlarges the live region to include the current (forward) value of $i + c$. This means that the constraints on $ are relaxed such that $\$ = i + c$ is satisfiable. First, we integrate the forward information and the fact that $\texttt{A}[i + c]$ is live. Then, the resulting constraint graph is joined with the constraint graph after the statement to obtain the constraint graph before the statement.

Figure 9 corresponds to integration of the forward and backward information before node 6, occurring in the first visit of that node. Then we join it with the current liveness information after node 6, which is *dead(A)*.

Assignment to an Array Element For a statement assigning to `A[i+c]`, the algorithm can shrink the live region to exclude the current (forward) value of $i + c$. This means that the constraints on $ can be made stronger to exclude the liveness of $\texttt{A}[i + c]$.

In the constraint graph this corresponds to decrementing the c edge from vertex \$ to vertex i by 1 and decrementing the $-c$ edge from vertex i to vertex \$ by 1. In Figure 10 the constraint graph before supergraph node 18, `stack[top]=o`, by shrinking the live region $\$ \leq top$ to $\$ \leq top + (-1)$.

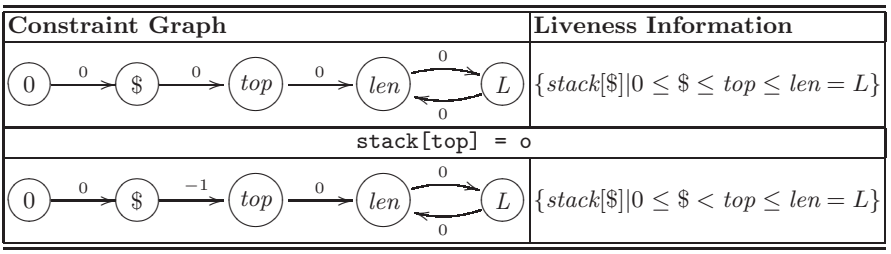


Fig. 10. The constraint graph before an assignment statement to an array element

Assignment Statements For the statement `i = j + c`, the liveness information is obtained by substituting occurrences of i with $j + c$. If i occurs in the left side of a constraint, then the constraint is normalized. For example, for the constraint $i \leq j' + c'$, after substituting $j + c$ for i, the normal form becomes $j \leq j' + (c' - c)$.

In Figure 11 the constraint graph before supergraph node 12, `top = top + (-1)`, is obtained by incrementing 0 edge from vertex \$ to vertex *top* by -1, and decrementing -1 edge from vertex *top* to vertex *len* by -1.

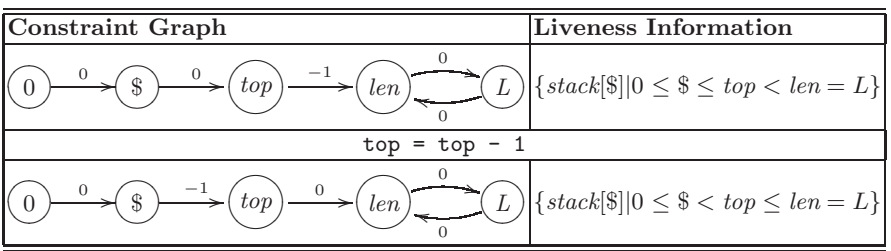


Fig. 11. The constraint graph before an assignment statement

Widening ∇ (see [5]) shown in Figure 6 accelerates the termination of the algorithm by taking the strongest implied constraints from the former visit of a supergraph node that remain true in the current visit of supergraph node.

Conditions The liveness information allows us to partially interpret program conditions in many interesting cases. This is a bit tricky, since supergraph nodes are visited in a backward direction.

The conditions of the form $i \leq j + c$ are handled by creating a constraint graph having one c edge from vertex i to vertex j, and then integrating it with the liveness information along the true edge (see Figure 7).

4.4 Forward Computation of Inequalities between Variables

The forward phase is an iterative algorithm for computing inequalities between integer variables and fields. The algorithm is *conservative*, i.e., every detected inequality at a supergraph node must hold on every execution through a program point represented by that node.

Formally, the forward phase is an iterative procedure that computes the constraint graph $w_f[n]$ at every supergraph node n, as the least solution to the following system of equations:

$$\begin{array}{l} w_f[n] = \begin{cases} w_\emptyset & n = Enter\langle Class\rangle \\ w_f[n] \nabla \bigsqcup_{m \in pred(n)} w_f[m,n] & \text{otherwise} \end{cases} \\ w_f[m,n] = [\![st(\langle m,n\rangle)]\!]^\sharp_f(w_f[m]) \end{array} \tag{5}$$

where $[\![st(\langle m,n\rangle)]\!]^\sharp_f$ is shown in Figure 7.

Since the forward phase is not new (e.g., a more precise version is given in [6]), in this paper, we only explain the backward phase.

5 GC Interface to Exploit Algorithmic Results

The output of the algorithm is a set of constraints associated with each program point that describe what sections of an array are alive at that point. A constraint may depend on the instance and local variables of the class and may include simple functions on those variables, e.g., `top - 1` for the `Stack` class. We choose to exploit instance variables constraints that hold at all program points at which a thread can be stopped for garbage collection. The points at which a thread can be stopped are precisely the gc-points of a type-accurate collector [1,7]. We judiciously choose where to put gc-points so that the "best" constraint holds. For example, the constraint for the `Stack` class is that the elements of the `stack` array from `0` through `top - 1` are alive, provided that there is no gc-point between the beginning of `pop` and statement `s`.

The chosen constraints are information that is logically associated with a specific class. Thus, it makes sense to store the constraints in the class data structure (or class object) together with the other information specific to the class, e.g., method table and description of fields. Notice that if a class has more than one array as an instance variable, then a set of constraints can be associated with each array field. A class-wide flag is also set in the class structure to indicate that it has at least one such array field.

When a tracing GC [16] (either mark-sweep or copying) encounters an object during its trace, it checks the class-wide flag in the object's class structure. If the flag is set, the collector traces the arrays reachable from the object, limiting its trace of the arrays according to their associated constraints.

6 Conclusion

We have presented a practical algorithm for determining the live regions of an array. The information produced by the algorithm can be used by GC in order to limit its trace of arrays, thereby leading to the collection of more garbage. Our experiments show a potential improvement of memory consumption.

The algorithm can determine a precise liveness information for some array based implementation of ADTs, namely stack and double stack (a stack growing in both directions). In addition, the extended version of the algorithm (maintaining at most two sets of constraint graphs per supergraph node) , described in [13] handles precisely dynamic vector. Further work is needed to analyze more Java programs and to detect more kinds of leaks occurring in arrays.

References

1. Ole Agesen, David Detlefs, and Elliot Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 1998. 53, 55, 64
2. David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 1998. 52
3. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Vancouver, B.C., 1998. 52
4. Cormen, Leiserson, and Rivest. *Algorithms.* MIT Press and McGraw-Hill, 1994. 54, 57
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press. 63
6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symp. on Princ. of Prog. Lang.*, January 1978. 53, 64
7. Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 273–282, San Francisco, CA, June 1992. 64
8. Rajiv Gupta. Optimizing array bound checks using flow analysis. *Let. on Prog. Lang. and Syst.*, 2(1–4):135–150, March–December 1993. 53
9. Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1995. 53
10. Barbara Liskov et al. CLU reference manual. In *Lec. Notes in Comp. Sci.*, volume 114. Springer-Verlag, Berlin, 1981. 54
11. William Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Communications of the ACM*, August 1992. 53

12. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, New York, NY, 1995. 56
13. Ran Shaham. Automatic removal of array memory leaks in Java. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, September 1999. Available at "http://www.math.tau.ac.il/∼ rans/thesis.zip". 53, 54, 56, 65
14. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981. 56
15. SPEC JVM98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/. 55
16. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management, International Workshop IWMM*, September 1992. 65
17. Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995. 53