Automatic Removal of Array Memory leaks in Java

A thesis
submitted as partial fulfillment
of the requirements towards the M.Sc. degree

by

Ran Shaham

under the supervision of

Dr. Mooly Sagiv and Dr. Elliot Kolodner

School of Mathematical Sciences
Raymond and Beverly Sackler Faculty of Exact Sciences
Tel-Aviv University
Tel-Aviv, Israel

September, 1999

To my parents, Rivka and Joshua
and to my wonderful wife, Yael

Acknowledgments

I would like to express my gratitude to Dr. Mooly Sagiv for his dedicated supervision, guidance, help and great patience throughout this work. I would like to thank Dr. Elliot Kolodner for his advices, ideas, help and great patience throughout this work.

I would like to thank Jong-Deok Choi, Nurit Dor, Sholmit Pinter, Noam Rinetskey for reading the drafts and for their helpful comments. Special thanks to Ayal Zaks for his helpful comments and ideas.

Many thanks to Binational Science Foundation, grant number 96-00337, for their financial support.

Many thanks to IBM Haifa Research Laboratory for their financial, technical and professional support.

Abstract

Automatic *garbage collection* (GC) reclaims memory that can no longer be used by the running program, and makes this memory available for reuse. Current 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. Some programmers try to circumvent these memory leaks by rewriting their source code, i.e., explicitly assigning null to object references that are guaranteed not to be used again. Such solutions may lead to erroneous (or slower) programs and may even be eliminated by optimizing compilers.

In this thesis, we present a practical algorithm for statically detecting such memory leaks occurring in arrays of objects. No previous algorithm exists. The algorithm is conservative, i.e., it never reports 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 GC, 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. The algorithm operates on Java Bytecode and can analyze one class at a time. We believe that this will allow our algorithm to scale for large programs.

Contents

1 Introduction

Programming languages such as Java relieve the programmer from the burden of explicit memory management through the use of an automatic garbage collection algorithm (GC) that is applied behind the scenes. This makes programming in these languages significantly easier than in *C* or *C++* since many run-time errors are avoided. Moreover, consider an object used by several modules. It should be deallocated only when all modules are no longer interested in that object. In explicit memory management model, the deallocation of such an object creates an inter-module dependency. In an automatic memory management model, modular programming is supported naturally since storage reclamation of the object is performed automatically.

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]The third top ranked bug by Java users in Sun's bug parade site as for August 1999 (`http://developer.java.sun.com/developer/bugStats/top25bugs.shtml`) regards Java's image memory leaks. Users report system crashes due to such memory

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 running example throughout the thesis[2].

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]`.

Such solutions is currently being employed in the JDK library, e.g., in the `jdk.util.Vector` class[3] 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 surely not a good programming practice. In fact, the whole idea of GC-aware programs defeats some of the purposes of automatic GC.
- Aiding the memory management task may require knowledge of the GC algorithm, which is implementation dependent. This may lead to programs that depend on a particular GC algorithm.
- 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.
- An optimizing compiler may deduce that `null` assignment statements have no effect, thus eliminating them!

---

leaks.

[2]We purposely do not check for overflow of the `stack` array in `push` and `pop`, as this leads to a more interesting analysis. Our algorithm would yield more precise results for a more robust class implementation.

[3]Null assignments in Vector are documented as "/* let GC do its work */"

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

(b)

Figure 1: (a) The Running Example Stack Class. (b) The Stack Class explicitly assigning `null` to prevent a memory leak.

Consider the `Vector` class in the `java.util` package, which implements a dynamic array of objects. Though it has already been instrumented[4] 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

[4]in Sun's implementation of the `Vector` class.

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

(a)

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

(b)

Figure 2: (a) `removeAllElments` method without the loop. Our algorithm detects that when calling `removeAllElements`, all `elementData` elements references are not subsequently used. (b) `removeAllElments` method of `java.util.Vector` class. The loop was added to allow GC to free the removed elements.

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 [AFM97, MBL97, CS98, BOSW98] being considered, and work showing how to reduce the cost of synchronization, e.g., [BKMS98], 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 Works

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 simple string search on Java program files and found some of the 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 savings of memory.

This research was inspired by work on liveness analysis for Java for local variables holding references [ADM98]. 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 would result in 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`. Moreover, this set of references is usually not known at compile-time; thus, the solution proposed in [KS98, SK98] which amounts to converting object arrays into multiple reference variables is not usually applicable.

Identifying liveness requires flow-sensitive analysis that may take superlinear 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, which operates on Java bytecode, can analyze one class at the time by conservatively approximating potential method invocations. We show that despite these conservative assumptions, such an algorithm is capable of finding memory leaks in many interesting cases, including the implementations 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 define 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. No similar algorithm exists. The compile-time cost of identifying dead regions of arrays is bigger than the cost of identifying dead reference variables but the potential run-time benefits seem larger, since a dead region of an array is equivalent to several dead reference variables. Comparing our empirical results to the results of [ADM98] show that this is the case for some array-based examples.

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., [Pug92, Wol95]). 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 bounds violations [Gup93, SPMS98, KW95]. One of the most precise methods was proposed by Cousot and Halbwachs [CH78]; it automatically identifies affine relationships between variables by scanning the control flow graph in a forward direction. We show 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 affine relationships between variables.

In this paper 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. The main idea is to determine the references and assignments to array elements, while exploiting

the forward information. Note that the algorithm is not bi-directional [Muc97]. There is a clear separation between the forward and backward phases. The latter uses the results of the forward phase, and has no effect on the forward phase results.

We present a variant of a forward direction algorithm, identifying relationships between variables, since it serves as an introduction to the backward phase. Both phases use the *constraint graph* described in [CLR94, Chapter 25.5, pp.539–543] as a simple representation of program variables relationships. The constraint graph allows us to efficiently represent a special case affine relationships of the form

$$x \leq y + c$$

where `x` and `y` are program variables or fields and c is an integer constant. Section 6 explains 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. A `null` assignment is added in program points where the live region of the analyzed array decreases. In the running example, our algorithm detects that before program point s, array elements `stack[0], stack[1] ...stack[top]` are live, while after s, array elements `stack[0], stack[1] ..., stack[top-1]` are live. 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 includes the full implementation of both phases of the algorithm, the forward phase and the backward phase. The prototype has no front-end, so the input to the prototype is the approximated supergraph of a class. The prototype hase a nice feature of producing LaTeX output. Tables 2, 3, 4 are automatically produced by the implementation.

Additionally, the extended version of the algorithm as described in Section 6.1 was implemented, and used to find dead array regions for java.util.Vector class.

Some programming languages, such as CLU [L+81, Lis93], provide built-in dynamic arrays that can be used to implement stacks and vectors. This would partially eliminate the need for our algorithm. 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.

Static analysis was also used for other kinds of allocation optimization. The following techniques are used in order to avoid GC, and inserting deallocation statements by the compiler.

Compile-Time garbage collection is an allocation optimization that statically determines and recycles garbage heap cells. Recycling is done through a special inserted code for handing the release of garbage heap

cells in two possible ways :

1. if the heap management has a free list then a released cell can simply be added to the free list.
2. if the release can be paired with a new allocation that will be executed in the subsequent code, then the cell need not be added to the free list: it can simply be reused directly.

Using compile-time garbage collection techniques in a run-time garbage collected environment aims to reduce the cost of the run-time GC, by recycling cells using cheaper specialized code instead of a general purpose GC code. Much work have been done on compile-time garbage collection for functional languages [Jon99, JM90, Moh90, Ham95]. Little work on compile-time garbage was done for object-oriented languages [Sei98]. Experimental results [Jon94] for functional languages point out a reduction in GC time, but not in total execution time.

Our technique and compile-time garbage collection techniques have a different nature. While our algorithm detects possible garbage memory ("may garbage"), compile-time garbage collection techniques detect memory which is definitely not used subsequently in the program ("must garbage"). To detect "must garbage" our algorithm is combined either with run-time GC, or alternatively with aliasing algorithm. The latter combination can be used as a compile-time garbage collection technique. However, aliasing algorithms are either time-consuming or imprecise. We choose to concentrate on the combination of our algorithm with run-time GC.

Escape analysis is a static analysis that determines whether the lifetime of data exceeds its static scope. Escape analysis was introduced for functional languages [PG92, Bla98a] and recently was also used to object-oriented languages [Bla98b, WR98]. For object-oriented languages, such as Java, escape analysis is used to determine whether an object o does not escape from method m; thus o may be stack allocated in m, instead of being heap allocated, and as a result the object is deallocated without GC. Experimental results in [Bla98b] show an average speedup of 21% using escape analysis to stack allocate objects. In some sense, escape analysis is a special case of compile-time garbage collection, since a stack-allocated object is detected as garbage and deallocated when the respective stack frame is popped. Escape analysis may be applicable in some cases to stack allocate an array of objects, however, current escape analysis techniques do not generally handle stack allocation of objects referenced by instance fields.

Region analysis is another allocation optimization where all objects are allocated in a stack of regions, where the size of a region is not necessarily known at compile-time. The allocation and deallocation of regions is determined statically, so GC is not needed in a regions based memory management model. Region analysis was introduced for functional languages [BTV96, TT94] and recently was also used to object-oriented languages [CV98]. Experiments in [CV98] compare C++ programs running times and memory reuse for C++

with GC (either mark and sweep or generational) and for C++ with regions (either bounded sized regions or unbounded sized regions). While the running times seem to be about the same in both cases, memory reuse results differ greatly in their example programs.

A region-based memory management model, replaces the garbage collection memory management model. So, as discussed above, our algorithm is not applicable by itself to a region-based memory management model. However, the combination of the algorithm and alias analysis can be used for region analysis.

Final note is that the aforementioned allocation optimization techniques do not, currently, handle arrays precisely.

2 Motivating Experiments

We conducted some simple experiments in order to determine the frequency of the array leak problem and in order to measure the potential benefits of its elimination.

2.1 Frequency of the Problem

The experiment to determine the frequency of occurrence was conducted before having an implementation of the algorithm. Instead, we used lexical scanning of Java source files. We searched for classes having an array of objects field, 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. We plan to use the implementation of our algorithm to do a more exact search that detects additional occurrences of array 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.

2.2 Potential Benefits

Another consideration is the potential benefit of detecting and eliminating array memory leaks. Notice that an object can be collected only after it is no longer reachable. Thus, detecting a *dead* array element reference, when another aliased *live* reference exists, may eliminate a future memory leak, but is not guaranteed to save space.

Benchmark Program	Spec JVM98 Input			Other Input		
	Time × Space w/out leak (M $Byte^2$)	Time × Space with leak (M $Byte^2$)	Ratio	Time × Space w/out leak (M $Byte^2$)	Time × Space with leak (M $Byte^2$)	Ratio
javac	1073.98	1085.13	0.9897	784.31	797.61	0.9833
db	465.67	465.67	1	740.07	1008.92	0.7335
GCTest				1.06	8.79	0.1206

Table 1: Heap Size results

We conducted an experiment similar to the one conducted in [ADM98] using a modified JVM[5]. After every 100KB of allocation, we invoke the *GC*, perform all possible finalizations, and perform *GC* again. We verify that *GC* is activated only upon the above request, by starting with a large initial heap size, and also by using a special flag. 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` (shown in Figure 4) and `removeAllElements` (shown in Figure 2) methods. Then we compared the allocation integral, calculated as the area under the heap size curve. The programs are taken from SPEC JVM98 [SPE98], a suite to measure the performance of Java platforms. Only the benchmark programs making use of `java.util.Vector` were considered. We measured these programs both with the original Spec JVM98 input and with other inputs that we constructed for testing purposes.

We also constructed an example program, `GCTest` to demonstrate the benefit of eliminating array memory leaks, shown in Figure 3. Our program creates a `vector` consisting of `vector` elements. Every element of the inner `vector` is initialized with a reference to a large newly allocated object; then this reference is immediately removed. Thus, running this example with the version of `vector` with leaks may lead to an OutOfMemoryError exception. We also include experimental results for the GCTest example.

Table 1 presents our experimental data obtained on a 400 MHz Intel Pentium-II CPU with 128MB of memory, running Windows NT 4.0. Results vary, from no change to dramatic change in heap size.

[5]We used Sun's JDK 1.2, Classic VM as the basis.

```
import java.util.Vector;


 void GCTest() {
      Vector v = new Vector();
       for (int i=0; i<100; i++) {
           Vector tmpVec = new Vector(1);
           Object[] tmpObjVec = new Object[1000000];
           tmpVec.addElement(tmpObjVec);
           tmpVec.removeAllElements();
           v.addElement(tmpVec);
      }
}
```

Figure 3: GCTest example.

3 Settings

3.1 Doing it in Java Bytecode

In general, our algorithm is applicable to a garbage collected programming environment, e.g., C with GC [BW88]. However, practical considerations, such as algorithm scalability and preciseness, suggest that the *Java* programming language is a natural choice.

Main reasons are:

- *Java* lacks of pointer arithmetic. This enables, in general, more accurate aliasing information, which is a prerequisite to a practical implementation of the algorithm.
- Data encapsulation is supported naturally in *Java* through the object-oriented model. In particular, data encapsulation enables the analysis of a single class file at a time.
- *Java* lacks explicit memory deallocation operation, as oppose to C/C++, and thus making GC a natural implementation choice[6] includes built-in automatic storage management system (typically a

[6]Formally, GC is not mandatory in Java Virtual Machine, although it is supported in all the Java Virtual Machines we know.

```
void removeElementAt(int index {
     if (index >= elementCount) {
        throw new ArrayIndexOutOfBoundsException (index + " >= " + elementCount);
     }
     else if (index < 0) {
        throw new ArrayIndexOutOfBoundsException(index);
     }

     int j=elementCount-index-1;
     if (j > 0) {
        // Copy elements one place to the ‘‘left’’
        System.arraycopy(elementData, index + 1, elementData, index, j);
     }
     elementCount--;
     elementData[elementCount]=null;
}
```

Figure 4: `removeElementAt` removes an element at a given index.

garbage collector), as oppose to C/C++.

In addition, applying the algorithm to *Java* bytecode is preferable than applying it to *Java* source code since:

- From the perspective of the compiler, a class file contains almost the same information as *Java* source.
- *Java* bytecode has already been used as a target language for various languages other than *Java* e.g. Ada95 [CDG97], Eiffel [C$^+$99], Scheme [Bot99].
- Many commercial libraries are distributed only as class file. By using class files we have access to the whole class files of a program even if the source is not available.
- Java is dynamic and not all information is available until runtime. Thus, the most natural place to implement is as part of a dynamic compiler for Java at the bytecode level, e.g., JIT .

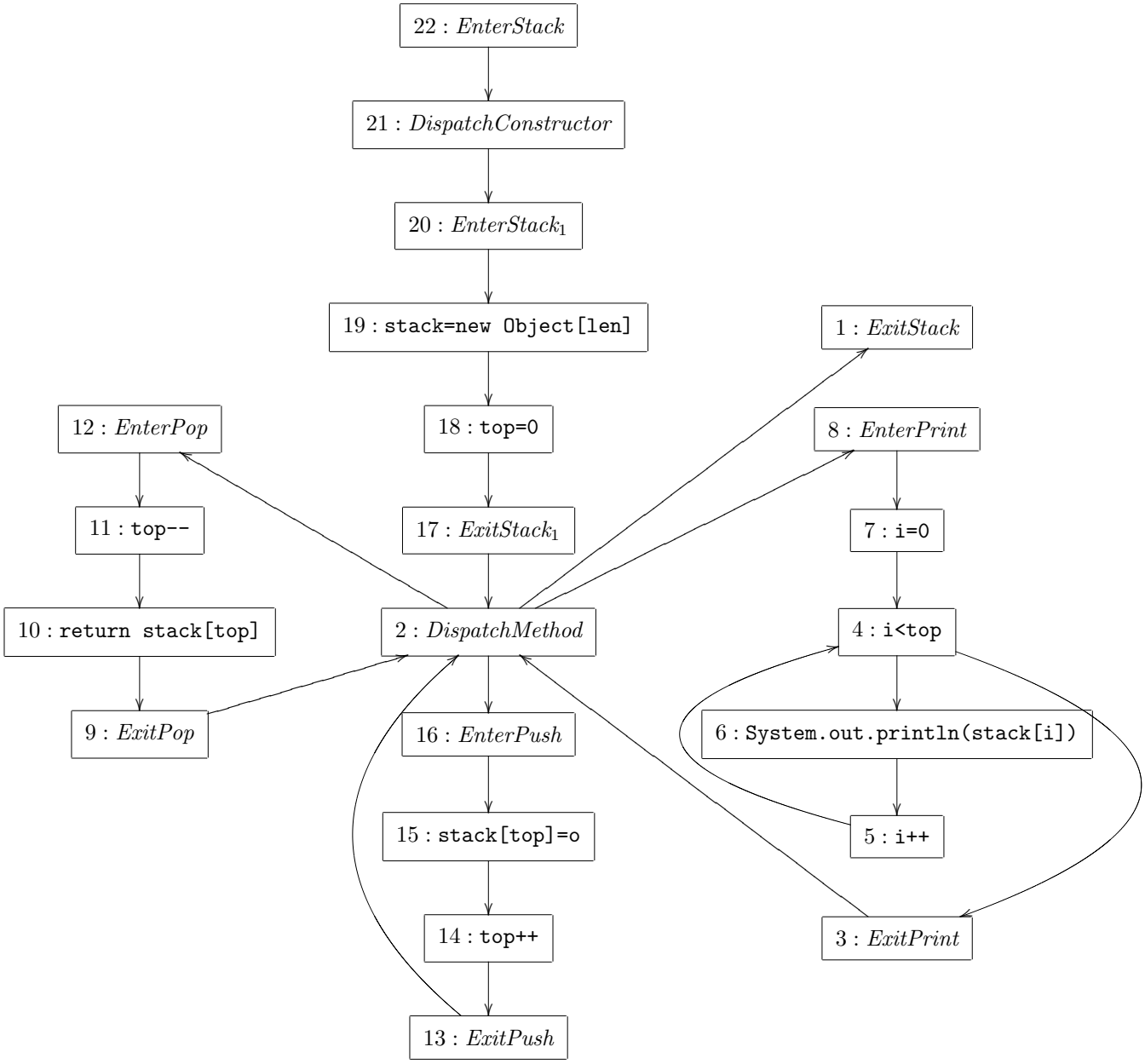


Figure 5: The approximated supergraph of the running example. Supergraph nodes are numbered according to post depth first search order. This order improves the efficiency of the iterative algorithm (see Section 4).

3.2 Class Level Analysis of Java Programs

In this section, we lay the groundwork for our simple class level analysis of *Java* (extensions are discussed in Section 6). Despite the fact that the algorithm is intended to analyze Java bytecode, we choose to explain the algorithm using the original Java statements from our running example.

The *program supergraph*(see [Mye81, SP81, RHS95]) 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 define *the approximated supergraph*. The approximated supergraph of the running example is shown in Figure 5.

The approximated supergraph of the class C is an approximation of supergraphs occurring at any instance of C in the following sense:

- The *EnterClass* node corresponds to program points where an instance C is allocated. Essentially, it is similar to the entry of a program.

- Following Enter<Class> there is a special supergraph node, *DispatchConstructor*, corresponding to program points where a constructor is invoked. There are arcs from this node to any constructor in C. Thus, in the approximated supergraph there is no information regarding the constructor that was applied. Furthermore, we assume that the values of actual parameters used in the constructors are not known.

  In the running example, there is one constructor with statements shown in supergraph nodes 19 and 18. The value of `len` is assumed to be unknown. Thus, the analysis in Section 4 cannot make any assumption on the size of the array `stack`.

- After applying the constructor we allow an arbitrary number of invocations of methods belonging to the class using the node *DispatchMethod*. The values of the parameters to the methods are assumed to be unknown. A *finalize* method can be treated similarly, except that it can only be executed once.

  In the running example, supergraph nodes 7, 4, 6 and 5 include the statements for `print`, supergraph nodes 11 and 10 include the statements for `push`, and supergraph nodes 15 and 14 include the statements for `pop`. At node 15, the value of the parameter `o`, pushed into the stack, is not known.

- When a class method invokes another method (possibly itself), we conservatively add an arc connecting the call node to the *DispatchMethod* node, and add arcs connecting *DispatchMethod* node and the successor nodes of the call node. This approximates a possible callee, which in turn invokes a method belonging to the class. Finer-grained approaches can be taken into consideration, e.g., considering non-virtual invocation. For example, for invocation of a *private* method M' belonging to the class, an arc can be added from the call node to *beginM'* node, and arcs can be added from *endM'* to the the successor nodes of the call node.

- There is no interleaving between execution paths on a single class instance, occuring in different program threads, due to the fact that all class methods are synchronized. Finer-grained approaches can be taken into consideration, e.g., when class instances are not shared among program threads, class methods

may be non-synchronized. The program call graph is considered in order to verify that class instances are not shared among program threads.

- We ignore invocations of subclass methods (until Section 6) since they cannot affect encapsulated information. Since we are interested in *safety* properties, i.e., properties that hold (or do not hold) every time the control reaches supergraph node, execution paths that do not use the encapsulated data are immaterial.

- The *Exit<Class>* supergraph node represents the program points where an instance of C is not subsequently used in the program.

Encapsulation at the class level is ensured by using *private* fields and local variables, and by not allowing objects referenced by these variables to "escape" outside the class level scope[7]. In the running example, `top` is a private field. `i` is local variable. `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.

Extensions to the above framework to deal with class variables, static methods, and static initializers are straightforward. More advanced analysis that relaxes some of the constraints on encapsulation, e.g., to allow protected instance variables, has also been considered; it requires analyzing subclasses as well, or a relaxed compilation model [BK97]. The extension to protected instance variables is discussed in Section 6.

Exception handling constructs are handled conservatively by converting try-catch to `if` statement. Other precise methods, using a compact representation of the control flow graph, for modeling the effect of exception handling constructs can be considered, e.g., [CGHS99].

4 The Algorithm

In this section, we give an efficient algorithm for computing *liveness* information for arrays. This section is organized as follows: In Section 4.1, we define the problem by extending the classical definition of *liveness* of scalar variables. In Section 4.2 we recall the definition of constraint graph from [CLR94, Chapter 25.5, pp.539–543]. The constraint graph provides an efficient representation for special form of inequalities between index variables. Then, in Section 4.3, we use the constraint graph to give an iterative forward algorithm, which computes inequalities between index variables at every supergraph node. These inequalities play an

[7]We make exceptions to the escape rule for frequently used methods whose effect we know, e.g., `System.out.println` and `System.arraycopy`.

important role in identifying live regions for arrays. Finally, in Section 4.4, we present the algorithm for identifying live regions for arrays, at every supergraph node. This algorithm also uses the constraint graph and the forward information, computed by the algorithm in Section 4.3, to obtain quite precise *liveness* information for arrays at every supergraph node.

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 p occurs before the use of `var` and `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 4.1 *An expression* `e` **is live before a program point** p*, if there exists an execution sequence,* $\pi_1.\pi_2$ *such that*

- *The path* π_1 *ends at program point* p*.*
- `e` *denotes a location (or reference)* l *at the end of* π_1*.*
- l *is used at the end of* π_2 *without prior assignment along* π_2

In the running example 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 the information that is important for *GC* (see Section 5).

Notice that Definition 4.1 coincides with the classic *liveness* definition for 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 represents inequalities between program variables. The constraint graph is a practical implementation of a set of constraints of a special form. Operations on the set are equivalent to solving path problems on directed graphs.

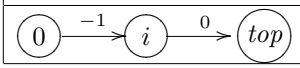
Constraint Graph	Inequalities Information
	$0 < i \leq top$

Figure 6: The constraint graph that represents the inequalities between integer variables after supergraph node 5.

Definition 4.2 *The* **constraint graph** *is a finite labeled directed graph* $G = (V, E, w)$*, with a vertex in* $v \in V$ *for every encapsulated integer variable or field and a special* $0 \in V$ *vertex. Every directed* $e \in E$ *is labeled by a* weight $w(e) \in Z$*. Such a directed graph represents the inequalities:*

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

The constraint graph, which represents the inequalities after supergraph node 5 (`i = i + 1`) of the running example, is shown in Figure 6. The -1 edge from 0 to i represents the inequality $0 \leq i + (-1)$, or $0 < i$.

The reader is referred to [CLR94, Chapter 25.5, pp.539–543] for explanations on the properties of constraint graphs. Every directed path in the graph induces an *implied constraint* between the source and the target of the path with a weight, which is the sum of the weights of the edges on that path. The shortest path between any two vertices corresponds to the *strongest implied constraint* between the source and target variables. Finally, the constraint graph *represents a contradiction* if there exists a negative directed cycle in the graph.

For a constraint graph $G(V, E, w)$ we denote by $TC(G)$ the constraint graph, whose edges are labeled with the strongest implied constraints. This means that for every pair of vertices x and y, an edge from x to y is set to the strongest implied constraint between x and y. We implemented $TC(G)$ using Floyd's all-pairs-shortest-path algorithm.

4.3 Forward Computation of Inequalities between Variables

The forward phase is an iterative algorithm for computing inequalities between integer variables and fields. The algorithm operates on the approximated supergraph. Inequalities are represented using constraint graphs. 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.

Supergraph nodes are visited in reverse post depth first order (e.g., see [Muc97] for such an algorithm). At every supergraph node the algorithm maintains a constraint graph. We have implemented such an algorithm

in Java and its output is displayed in Table 2. The iterations of the iterative backward algorithm on the running example are shown in Table 4. The final results of the iterative algorithm on the running example are shown in Table 3. A nice feature is that Tables 2, 3, 4 are automatically produced by the implementation.

The algorithm starts by assuming that all information is available and then propagates inequalities along supergraph paths. When several paths into a supergraph nodes exist, the algorithm conservatively assumes that only inequalities that hold on all these paths remain true[8].

As usual, our algorithm may be overly conservative, i.e., it may miss inequalities that always hold.

In particular, there are four types of inaccuracies:

- We do not yet take into account the fact that *Java* requires that array references be safe. For example, i must be non-negative after a use of `a[i]`. This extension is rather easy.
- We only interpret assignment statements of the form $i = j + c$ or $i = c$ where i, j are program variables and c is a constant; thus other assignment statements are interpreted conservatively by eliminating the inequalities regarding the assigned variable in the resulting set of constraints. Also, we only interpret conditions of the form $i \leq j + c$, $i \leq c$ or $c \leq i$, where i, j are program variables and c is a constant; other conditions are not interpreted.
- We use a single supergraph node to represent many program points and use extra supergraph arcs, leading to infeasible paths.
- The supergraph may contain "invalid" interprocedural paths that do not respect the call-return mechanism. There are two classical solutions to this problem due to Sharir and Pnueli [SP81]. We have not yet implemented either of these solutions.

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

Id	**Node**	**Statement**	**Constraint Graph**	**Inequalities Information**
1	22	*EnterStack*	$\perp$	$\perp$
2	21	*DispatchConstructor*	$\perp$	$\perp$

[8] One exception are program conditions which are partially interpreted, thus, eliminating some infeasible path.

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

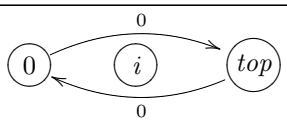
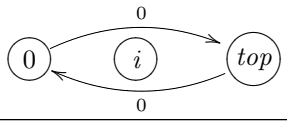
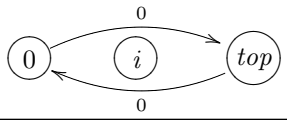
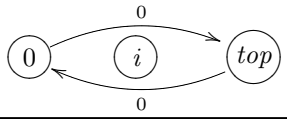
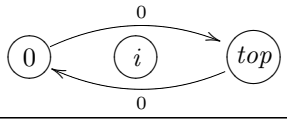
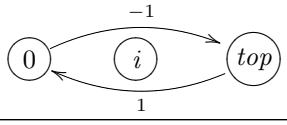
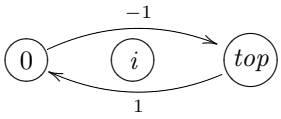
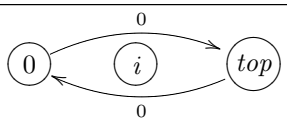
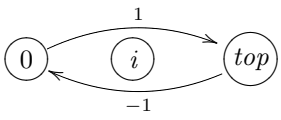
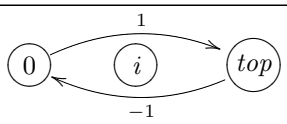
Id	Node	Statement	Constraint Graph	Inequalities Information
3	20	*EnterStack1*	$\bot$	$\bot$
4	19	`stack= new[...]`	$\bot$	$\bot$
5	18	`top = 0`		$\{0 \leq top, top \leq 0\}$
6	17	*ExitStack1*		$\{0 \leq top, top \leq 0\}$
7	2	*DispatchMethod*		$\{0 \leq top, top \leq 0\}$
8	16	*EnterPush*		$\{0 \leq top, top \leq 0\}$
9	15	`def stack[top]`		$\{0 \leq top, top \leq 0\}$
10	14	`top = top + 1`		$\{0 < top, top \leq 1\}$
11	13	*ExitPush*		$\{0 < top, top \leq 1\}$
12	12	*EnterPop*		$\{0 \leq top, top \leq 0\}$
13	11	`top = top - 1`		$\{-1 \leq top, top < 0\}$
14	10	`use stack[top]`		$\{-1 \leq top, top < 0\}$

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

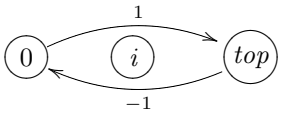
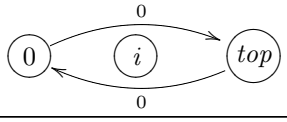
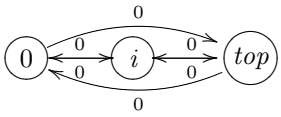
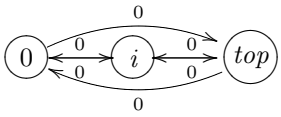
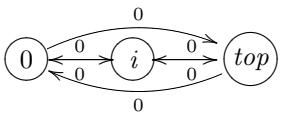
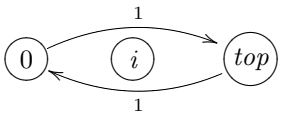
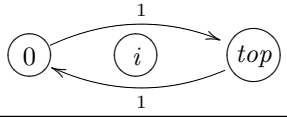
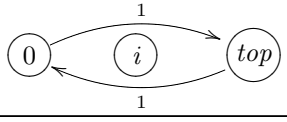
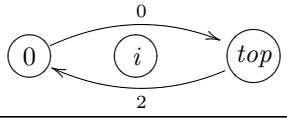
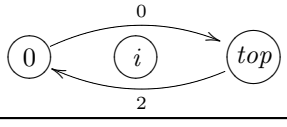
Id	Node	Statement	Constraint Graph	Inequalities Information
15	9	*ExitPop*		$\{-1 \leq top, top < 0\}$
16	8	*EnterPrint*		$\{0 \leq top, top \leq 0\}$
17	7	`i = 0`		$\{0 \leq i, 0 \leq top, i \leq 0, i \leq top, top \leq 0, top \leq i\}$
18	4	`i < top`		$\{0 \leq i, 0 \leq top, i \leq 0, i \leq top, top \leq 0, top \leq i\}$
19	6	`use stack[i]`	$\bot$	$\bot$
20	5	`i = i + 1`	$\bot$	$\bot$
21	3	*ExitPrint*		$\{0 \leq i, 0 \leq top, i \leq 0, i \leq top, top \leq 0, top \leq i\}$
22	2	*DispatchMethod*		$\{-1 \leq top, top \leq 1\}$
23	16	*EnterPush*		$\{-1 \leq top, top \leq 1\}$
24	15	`def stack[top]`		$\{-1 \leq top, top \leq 1\}$
25	14	`top = top + 1`		$\{0 \leq top, top \leq 2\}$
26	13	*ExitPush*		$\{0 \leq top, top \leq 2\}$

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

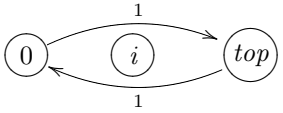
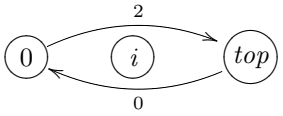
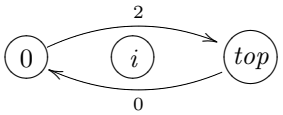
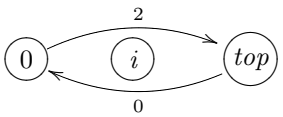
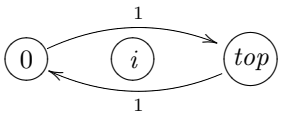
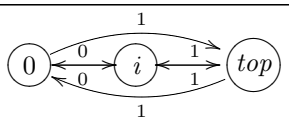
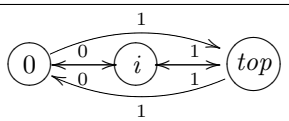
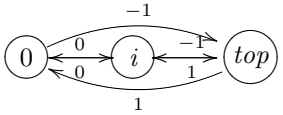
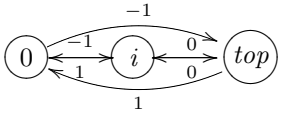
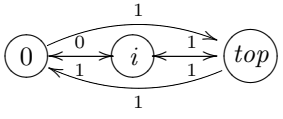
Id	Node	Statement	Constraint Graph	Inequalities Information
27	12	*EnterPop*		$\{-1 \leq top, top \leq 1\}$
28	11	`top = top - 1`		$\{-2 \leq top, top \leq 0\}$
29	10	`use stack[top]`		$\{-2 \leq top, top \leq 0\}$
30	9	*ExitPop*		$\{-2 \leq top, top \leq 0\}$
31	8	*EnterPrint*		$\{-1 \leq top, top \leq 1\}$
32	7	`i = 0`		$\{0 \leq i, -1 \leq top, i \leq 0, i \leq top + 1, top \leq 1, top \leq i + 1\}$
33	4	`i < top`		$\{0 \leq i, -1 \leq top, i \leq 0, i \leq top + 1, top \leq 1, top \leq i + 1\}$
34	6	`use stack[i]`		$\{0 \leq i, 0 < top, i \leq 0, i < top, top \leq 1, top \leq i + 1\}$
35	5	`i = i + 1`		$\{0 < i, 0 < top, i \leq 1, i \leq top, top \leq 1, top \leq i\}$
36	4'	`i < top`		$\{0 \leq i, -1 \leq top, i \leq 1, i \leq top + 1, top \leq 1, top \leq i + 1\}$

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

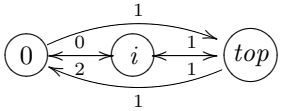
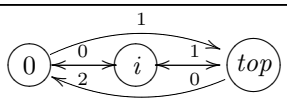
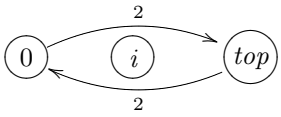
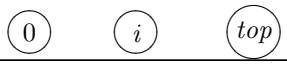
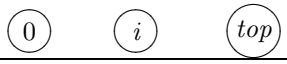
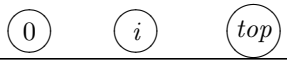
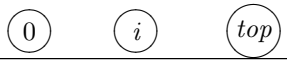
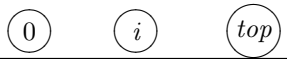
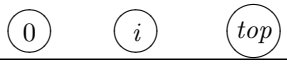
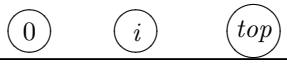
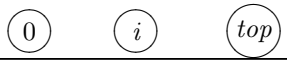
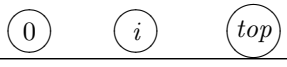
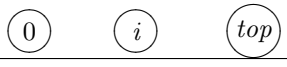
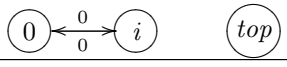
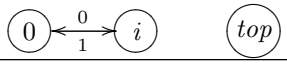
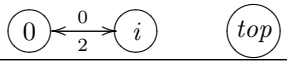
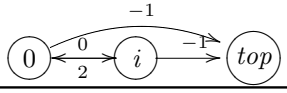
Id	Node	Statement	Constraint Graph	Inequalities Information
37	4	`i < top`		$\{0 \leq i, -1 \leq top, i \leq 2, i \leq top + 1, top \leq 1, top \leq i + 1\}$
38	3	*ExitPrint*		$\{0 \leq i, -1 \leq top, i \leq 2, i \leq top + 1, top \leq 1, top \leq i\}$
39	2'	*DispatchMethod*		$\{-2 \leq top, top \leq 2\}$
40	2	*DispatchMethod*		$\top$
41	16	*EnterPush*		$\top$
42	15	`def stack[top]`		$\top$
43	14	`top = top + 1`		$\top$
44	13	*ExitPush*		$\top$
45	12	*EnterPop*		$\top$
46	11	`top = top - 1`		$\top$
47	10	`use stack[top]`		$\top$
48	9	*ExitPop*		$\top$
49	8	*EnterPrint*		$\top$
50	7	`i = 0`		$\{0 \leq i, i \leq 0\}$
51	4'	`i < top`		$\{0 \leq i, i \leq 1\}$
52	4	`i < top`		$\{0 \leq i, i \leq 2\}$
53	6	`use stack[i]`		$\{0 \leq i, 0 < top, i \leq 2, i < top\}$

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

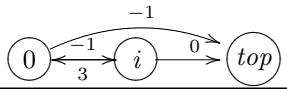
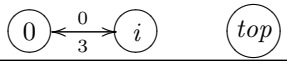
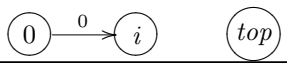
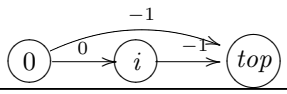
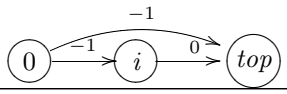
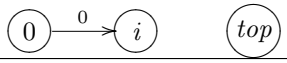
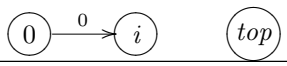
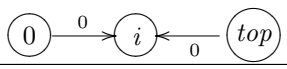
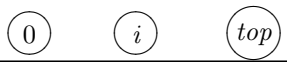
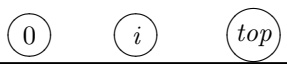
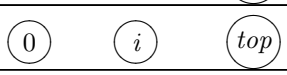
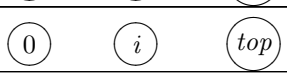
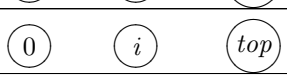
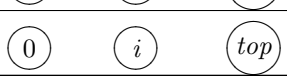
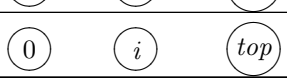
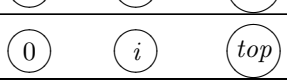
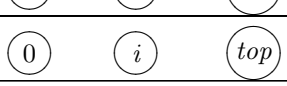
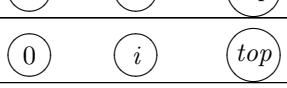
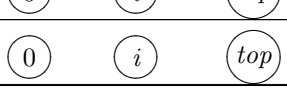
Id	Node	Statement	Constraint Graph	Inequalities Information
54	5	`i = i + 1`		$\{0 < i, 0 < top, i \leq 3, i \leq top\}$
55	4'	`i < top`		$\{0 \leq i, i \leq 3\}$
56	4	`i < top`		$\{0 \leq i\}$
57	6	`use stack[i]`		$\{0 \leq i, 0 < top, i < top\}$
58	5	`i = i + 1`		$\{0 < i, 0 < top, i \leq top\}$
59	4'	`i < top`		$\{0 \leq i\}$
60	4	`i < top`		$\{0 \leq i\}$
61	3	*ExitPrint*		$\{0 \leq i, top \leq i\}$
62	2'	*DispatchMethod*		$\top$
63	2	*DispatchMethod*		$\top$
64	16	*EnterPush*		$\top$
65	15	`def stack[top]`		$\top$
66	14	`top = top + 1`		$\top$
67	13	*ExitPush*		$\top$
68	12	*EnterPop*		$\top$
69	11	`top = top - 1`		$\top$
70	10	`use stack[top]`		$\top$
71	9	*ExitPop*		$\top$
72	8	*EnterPrint*		$\top$

Table 2: The iterations of our worklist algorithm for identifying inequalities information in the running example. We show the supergraph node, the statement, and constraint graph after this statement and the inequalities represented. Tagged nodes represent information before a widening is performed.

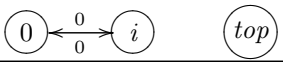
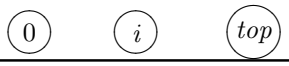
Id	Node	Statement	Constraint Graph	Inequalities Information
73	7	`i = 0`		$\{0 \leq i, i \leq 0\}$
74	1	*ExitClass*		$\top$

Table 3: The final output of the worklist algorithm on the running example. We show the supergraph node, the statement, and constraint before this statement and the inequalities represented. $\perp$ is an artificial constraint graph and inequalities where all the encapsulated variables are not initialized $\top$ represent an empty constraint graph, i.e., unknown constraints on encapsulated varaibles.

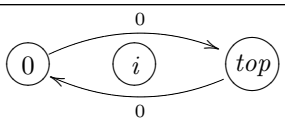
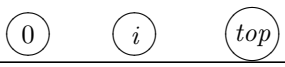
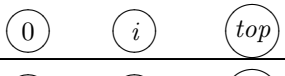
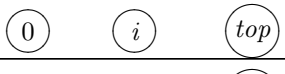
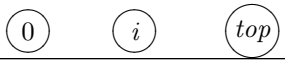
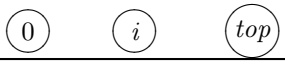
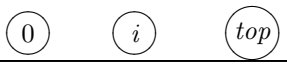
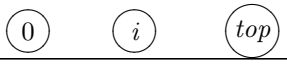
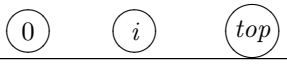
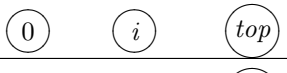
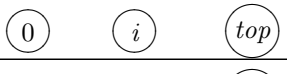
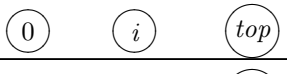
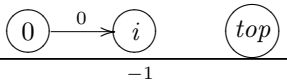
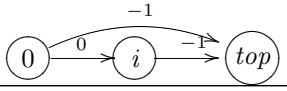
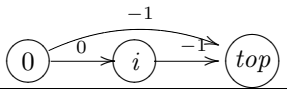
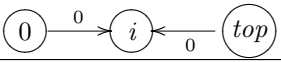
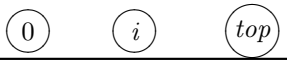
Id	Node	Statement	Constraint Graph	Inequalities Information
1	22	*EnterStack*	$\perp$	$\perp$
2	21	*DispatchConstructor*	$\perp$	$\perp$
3	20	*EnterStack1*	$\perp$	$\perp$
4	19	`stack= new[...]`	$\perp$	$\perp$
5	18	`top = 0`	$\perp$	$\perp$
6	17	*ExitStack1*		$\{0 \leq top, top \leq 0\}$
7	2	*DispatchMethod*		$\top$
8	12	*EnterPop*		$\top$
9	11	`top = top - 1`		$\top$
10	10	`use stack[top]`		$\top$
11	9	*ExitPop*		$\top$

Table 3: The final output of the worklist algorithm on the running example. We show the supergraph node, the statement, and constraint before this statement and the inequalities represented. $\perp$ is an artificial constraint graph and inequalities where all the encapsulated variables are not initialized $\top$ represent an empty constraint graph, i.e., unknown constraints on encapsulated varaibles.

Id	Node	Statement	Constraint Graph	Inequalities Information
12	16	*EnterPush*		$\top$
13	15	`def stack[top]`		$\top$
14	14	`top = top + 1`		$\top$
15	13	*ExitPush*		$\top$
16	8	*EnterPrint*		$\top$
17	7	`i = 0`		$\top$
18	4	`i < top`		$\{0 \leq i\}$
19	6	`use stack[i]`		$\{0 \leq i, 0 < top, i < top\}$
20	5	`i = i + 1`		$\{0 \leq i, 0 < top, i < top\}$
21	3	*ExitPrint*		$\{0 \leq i, top \leq i\}$
22	1	*ExitClass*		$\top$

In the following subsections, we briefly explain the iterations of our worklist algorithm:

4.3.1 Iterations

The algorithm starts with the *EnterClass* supergraph node; we conservatively assume that no information is available on non-encapsulated variables and optimistically assume that all information is available on encapsulated variables. This special constraint graph is denoted by $\perp$.

The algorithm continues to visit supergraph nodes until no more changes occur. When a supergraph node changes, its successors are visited. When a node is processed before all its predecessors are known, the constraint graphs at the unknown predecessors are ignored.

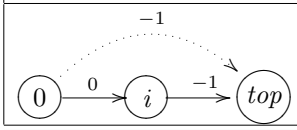
Constraint Graph	Inequalities Information
	$0 \leq i < top$

Figure 7: The constraint graph representing the inequalities information before supergraph node 5.

After the effect of a supergraph node is taken into account by the algorithm, we add the strongest implied constraints as defined in Section 4.2. In Figure 7, the dotted edge between vertex 0 and vertex *top* corresponds to the strongest implied constraint between 0 and *top*. One can "lazily" add strongest implied constraints, as supergraph nodes are being processed.

4.3.2 Skip Supergraph Nodes

The algorithm skips nodes, which do not effect inequalities information. For these skipped nodes, it simply copies the constraint graph before the node to the constraint graph after the node. These skipped nodes include:

- Nodes for statements, which do not assign to encapsulated variables.
- Special supergraph nodes, e.g., *EnterClass* and *DispatchMethod*.

In Table 2, the constraint graph after supergraph node 22 *EnterStack* at row 1 is obtained by copying $\perp$.

4.3.3 Join

In this section we define the *join* operation. Join is used when the supergraph flow merges. Join is the intersection of the (strongest implied) constraints occurring on all merging supergraph paths. This guarantees that every inequality that the algorithm obtains indeed holds every time the control reaches this point.

In the constraint graph representation, for a pair of vertices x, y the new constraint is the weakest constraint of the strongest implied constraints between x and y in the constraint graphs being joined, (i.e., the maximum of the shortest-path from x to y in the constraint graphs being joined).

In Table 2, the constraint graph before supergraph node 4, `i < top` at row 36, , is obtained by joining the constraint graphs after node `i = 0` at row 32, the constraint graph after `i = i + 1` at row 35. The edge connecting vertex 0 to vertex i is labeled by 0 as in the constraint graph after node `i = 0`. However, the edge connecting vertex 0 to vertex i is labeled by 1 as in the constraint graph after node `i = i + 1`. (Note that node 4 is a skip node, so although the constraint graph after node 4 is presented at row 36, it is equal to the constraint graph before node 4)

4.3.4 Assignment Statements

We consider three kinds of assignment statements:

- Statement of the form `i = c` where `i` is an encapsulated variable. This statement adds the constraints $i \leq 0 + c$ and $0 \leq i + (-c)$. Therefore, the constraint graph after this statement is obtained from the constraint graph before the statement by removing all edges leading to vertex i and emanating from vertex i, and then adding a $-c$ edge from vertex 0 to vertex i, and a c edge from vertex i to vertex 0 .

  In Table 2, the constraint graph after statement `top = 0` at row 5 is obtained from the constraint graph at row 4, $\perp$, by adding the two extra edges, 0 and -0.

- Statement of the form `i = i + c` where `i` is an encapsulated variable. This statement adds a constraint $i \leq j + (c' + c)$ for every constraint $i \leq j + c'$, in the constraint graph before the statement, and a constraint $j \leq i + (c' - c)$ for every constraint $j \leq i + c'$, in the constraint graph before the statement. Therefore, the constraint graph after this statement is obtained from the constraint graph before the statement by incrementing the c' edges from vertex i to a vertex j by c, and decrementing the c' edges from a vertex j to vertex i by c.

  In Table 2, the constraint graph after statement `top = top + 1` at row 10 is obtained from the constraint graph at row 9, and then incrementing the 0 edge from vertex *top* to vertex 0 by 1, and decrementing the -0 edge from vertex 0 to vertex *top* by 1.

- Statement of the form `i = e` where `i` is encapsulated variable and e is an expression other than c or $j + c$ for an encapsulated variable j. Such a statement is interpreted conservatively by eliminating edges leading to/emanating from vertex i.

4.3.5 Widening

Widening [CC79] accelerates the termination of the algorithm, by deliberately losing some information. In our case widening takes the strongest implied constraints from the former visit of a supergraph node (which is the target of a backedge) that remain true in the current visit of the supergraph node.

In Table 2, the constraint graph after supergraph node 2 statement `DispatchMethod` at row 40 is obtained by taking only strongest implied constraint from the constraint graph at row 22 (the former visit of node 2) that remain true in the constraint graph at row 39 (the current visit of node 2). The result it $\top$, since the constraints at row 22 are all weakened at row 39, thus all constraints are removed.

4.3.6 Conditions

Interestingly, the inequality information allows us to partially interpret program conditions in many interesting cases[9].

Conditions of the form $i \leq j + c$ are handled by propagating the constraint graph before the condition node strengthened with a c edge from vertex i to vertex j along the true edge, and by propagating the constraint graph before the condition node strengthened with a $-(c+1)$ edge from vertex j to vertex i along the false edge.

The condition `i` $\leq$ `c` is handled similarly where vertex 0 plays the role of vertex j. The condition `c` $\leq$ `i` is handled similarly where vertex 0 plays the role of vertex i, vertex i plays the role of vertex j, and $-c$ plays the role of c (since $c \leq i$ can be written as $0 \leq i + (-c)$).

In Table 2, the constraint graph along the true edge of node 4, which is supergraph node 6 statement `return stack[top]` at row 19 is obtained by strengthening the 0 edge from vertex i to vertex *top* at row 18 to be -1 (representing the fact the it is known that along the true edge the constraint $i < top$ holds). The constraints at row 18 are $top = i = 0$, so the path along the true edge is leading to node 6 is infeasible, since $i < top$ does not hold when $top = i = 0$, so $\perp$ result is expected at row 19. Indeed, the above strengthening leads to a negative cycle (consider the -1 cycle $i \xrightarrow{-1} top \xrightarrow{0} i$), i.e., contradiction, thus the resulting constraint graph is $\perp$. The constraint graph along the false edge of node 4, which is the constraint graph after supergraph node 3 at row 21 is obtained by strengthening the 0 edge from vertex *top* to vertex i to be 0, i.e., the constraint between *top* and i was already "strong" and was not strengthened by the fact that along the false edge $top \leq i$ holds.

4.4 Backward Computation of Live Regions

In this section, we sketch an iterative algorithm for computing live regions in arrays. For the purpose of the description, we assume one particular encapsulated array `A`. The algorithm operates on the approximated supergraph. Our chief insight is that live regions can also be represented using constraint graphs, with one additional designated node, denoted by \$. This node represents constraints on the live array indices of array `A`.

Although integer variables and \$ are different semantic objects, the representation of (conditional) program variables relations is syntactically the same as the representation of liveness constraints, thus it is possible to represent both in the same constraint graph, but with different meaning defined below.

For example, the constraint graph in Figure 8 corresponds to the liveness information before supergraph

[9] It resembles Floyd's strongest postconditions [Flo67]. This analogy is beyond the scope of this paper

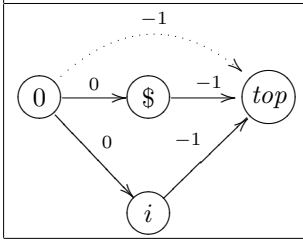
Constraint Graph	**Liveness Information**
	$\{stack[\$] \mid 0 \leq \$ < top, 0 \leq i < top\}$

Figure 8: The constraint graph representing the liveness information before supergraph node 4.

node 4. Since $0 \leq \$ < top$, this constraint graph represents the fact that array elements $\texttt{stack}[0], \texttt{stack}[1], \ldots, \texttt{stack}[\texttt{top} - 1]$ are live. In addition, the live region is conditional because of the -1 edge connecting vertex i to vertex *top*, i.e., the `stack` elements may be referenced only if the print loop is entered; thus, `i` must be less than `top`. In other words, if $i \geq top$ then none of the array elements is alive. Notice that the inequality $i < top$ may be violated at some paths leading to supergraph node 4.

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

$$\{A[\$] \mid \bigwedge_{\langle x,y \rangle \in E} x \leq y + w(\langle x, y \rangle)\}$$

where `A` is the encapsulated array; `A`[$] denotes the live array elements, and live regions come into play when either x or y are \$. The constraint graph in Figure 8 represents the liveness information:

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

In addition, we also allow a special constraint graph, denoted by *dead*(`A`), representing the fact that all array elements are dead.

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 `A`$[i]$ is live before p then i satisfies all the constraints in the constraint graph that the algorithm yields at n.

Supergraph nodes are visited in post depth first order (e.g., see [Muc97] for such an algorithm). At every supergraph node the algorithm maintains a constraint graph. We have implemented such an algorithm in Java.

The iterations of the iterative backward algorithm on the running example are shown in Table 4.

The algorithm starts by assuming that all information is available, and the array is dead. 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.

When several paths from a supergraph nodes exist, the algorithm conservatively assumes that liveness information includes the liveness information along all outgoing paths[10].

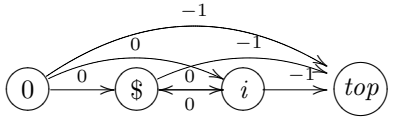
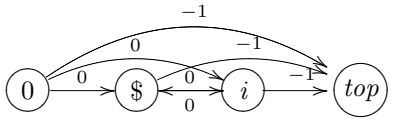
As usual, our algorithm may be overly conservative, i.e., at program point p it may assume some array elements as live, although there is no execution path from p to a use of these array elements prior to redefinition.

In particular, there are five types of inaccuracies, four of them are the aforementioned inaccuracies of the forward phase. In addition:

- We only interpret assignment to array elements and uses of array elements of the form $A[i+c]$, where `A` is the analyzed array, i is a program variable and c is a constant; thus other assignments to array elements are interpreted conservatively by not shrinking the liveness information. Other uses of array elements are interpreted conservatively by assuming that all array elements are live.

Despite of these inaccuracies, our analysis is capable of computing quite precise results for the `Stack` class. Moreover, modifying `Stack` code in `push, pop` methods to check overflow, yields even more precise results.

Table 4: The iterations of our worklist algorithm for identifying *liveness* regions on the running example. Tagged nodes represent information before a widening is performed.

Id	Node	Statement	Constraint Graph	Liveness Information
1	1	*ExitClass*	$dead(stack)$	$dead(stack)$
2	2	*DispatchMethod*	$dead(stack)$	$dead(stack)$
3	3	*ExitPrint*	$dead(stack)$	$dead(stack)$
4	4	`i < top`	$dead(stack)$	$dead(stack)$
5	5	`i = i + 1`	$dead(stack)$	$dead(stack)$
6	6	`use stack[i]`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ \leq i, \$ < top, i \leq \$, i < top\}$
7	4	`i < top`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ \leq i, \$ < top, i \leq \$, i < top\}$

[10] One exception are program conditions which are partially interpreted, thus, eliminating some infeasible paths.

Table 4: The iterations of our worklist algorithm for identifying *liveness* regions on the running example. Tagged nodes represent information before a widening is performed.

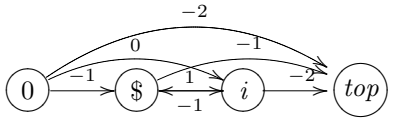
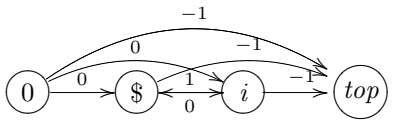
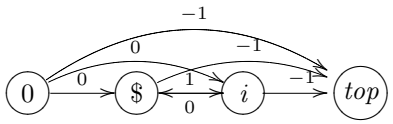
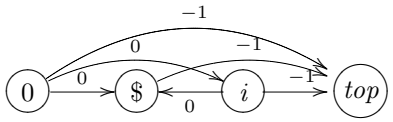
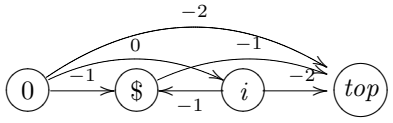
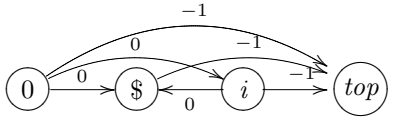
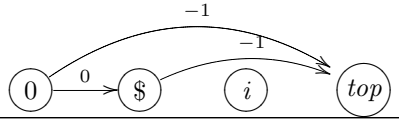
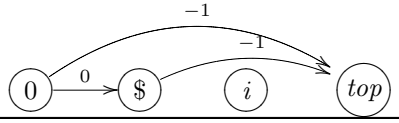
Id	Node	Statement	Constraint Graph	Liveness Information
8	5	`i = i + 1`		$\{stack[\$] \mid 0 < \$, 0 \leq i, 2 \leq top, \$ \leq i+1, \$ < top, i < \$, i \leq top-2\}$
9	6	`use stack[i]`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ \leq i+1, \$ < top, i \leq \$, i < top\}$
10	4'	`i < top`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ \leq i+1, \$ < top, i \leq \$, i < top\}$
11	4	`i < top`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top, i \leq \$, i < top\}$
12	5	`i = i + 1`		$\{stack[\$] \mid 0 < \$, 0 \leq i, 2 \leq top, \$ < top, i < \$, i \leq top-2\}$
13	6	`use stack[i]`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top, i \leq \$, i < top\}$
14	7	`i = 0`		$\{stack[\$] \mid 0 \leq \$, 0 < top, \$ < top\}$
15	8	*EnterPrint*		$\{stack[\$] \mid 0 \leq \$, 0 < top, \$ < top\}$

Table 4: The iterations of our worklist algorithm for identifying *liveness* regions on the running example. Tagged nodes represent information before a widening is performed.

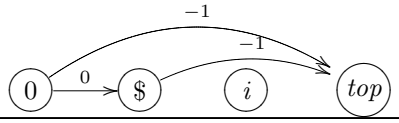
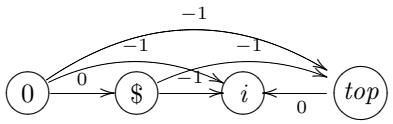
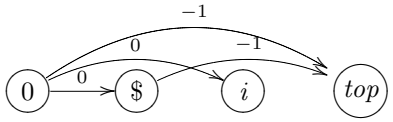
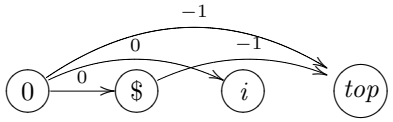
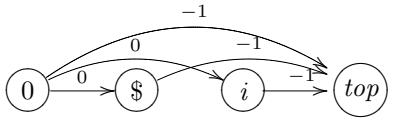
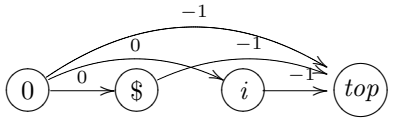
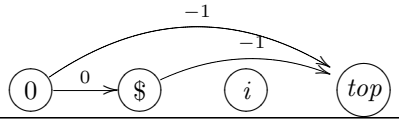
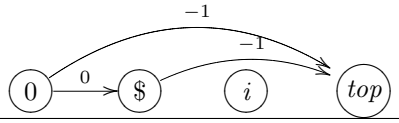
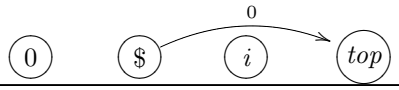
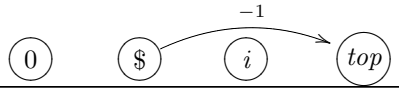
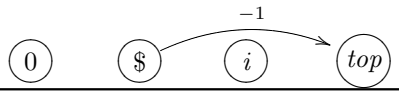
Id	Node	Statement	Constraint Graph	Liveness Information
16	2	*DispatchMethod*		$\{stack[\$] \mid 0 \leq \$, 0 < top, \$ < top\}$
17	3	*ExitPrint*		$\{stack[\$] \mid 0 \leq \$, 0 < i, 0 < top, \$ < i, \$ < top, top \leq i\}$
18	4'	`i < top`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top\}$
19	4	`i < top`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top\}$
20	5	`i = i + 1`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top, i < top\}$
21	6	`use stack[i]`		$\{stack[\$] \mid 0 \leq \$, 0 \leq i, 0 < top, \$ < top, i < top\}$
22	7	`i = 0`		$\{stack[\$] \mid 0 \leq \$, 0 < top, \$ < top\}$
23	9	*ExitPop*		$\{stack[\$] \mid 0 \leq \$, 0 < top, \$ < top\}$
24	10	`use stack[top]`		$\{stack[\$] \mid \$ \leq top\}$
25	11	`top = top - 1`		$\{stack[\$] \mid \$ < top\}$
26	12	*EnterPop*		$\{stack[\$] \mid \$ < top\}$

Table 4: The iterations of our worklist algorithm for identifying *liveness* regions on the running example. Tagged nodes represent information before a widening is performed.

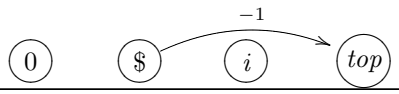
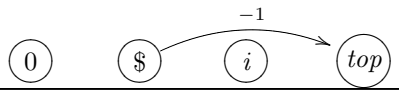
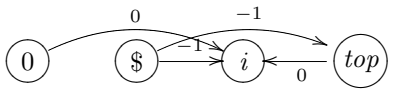
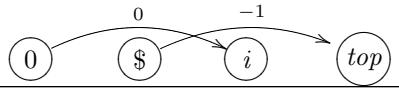
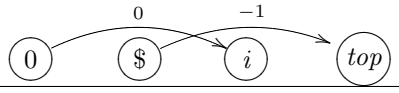
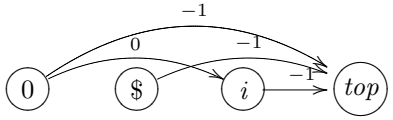
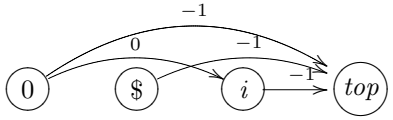
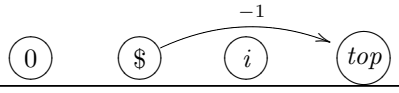
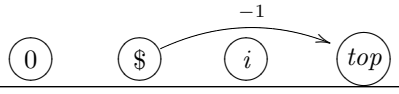
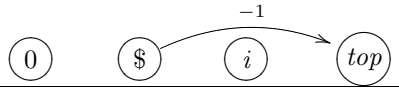
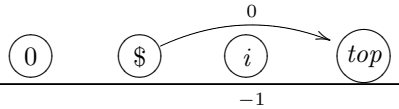
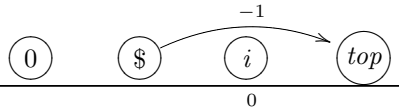
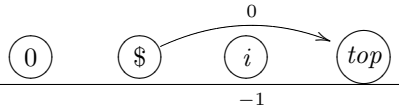
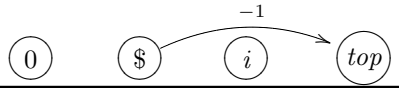
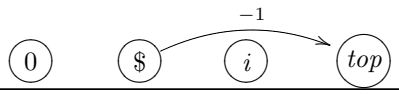
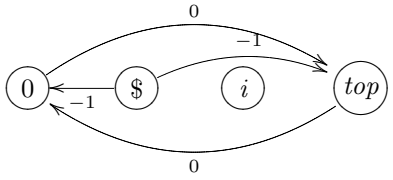
Id	**Node**	**Statement**	**Constraint Graph**	**Liveness Information**
27	2'	*DispatchMethod*		$\{stack[\$] \mid \$ < top\}$
28	2	*DispatchMethod*		$\{stack[\$] \mid \$ < top\}$
29	3	*ExitPrint*		$\{stack[\$] \mid 0 \leq i, \$ < i, \$ < top, top \leq i\}$
30	4'	`i < top`		$\{stack[\$] \mid 0 \leq i, \$ < top\}$
31	4	`i < top`		$\{stack[\$] \mid 0 \leq i, \$ < top\}$
32	5	`i = i + 1`		$\{stack[\$] \mid 0 \leq i, 0 < top, \$ < top, i < top\}$
33	6	`use stack[i]`		$\{stack[\$] \mid 0 \leq i, 0 < top, \$ < top, i < top\}$
34	7	`i = 0`		$\{stack[\$] \mid \$ < top\}$
35	8	*EnterPrint*		$\{stack[\$] \mid \$ < top\}$
36	9	*ExitPop*		$\{stack[\$] \mid \$ < top\}$
37	10	`use stack[top]`		$\{stack[\$] \mid \$ \leq top\}$
38	13	*ExitPush*		$\{stack[\$] \mid \$ < top\}$
39	14	`top = top + 1`		$\{stack[\$] \mid \$ \leq top\}$
40	15	`def stack[top]`		$\{stack[\$] \mid \$ < top\}$

Table 4: The iterations of our worklist algorithm for identifying *liveness* regions on the running example. Tagged nodes represent information before a widening is performed.

Id	Node	Statement	Constraint Graph	Liveness Information
41	16	*EnterPush*		$\{stack[\$] \mid \$ < top\}$
42	17	*ExitStack1*		$\{stack[\$] \mid 0 \leq top, \$ < 0, \$ < top, top \leq 0\}$
43	18	`top = 0`		$\{stack[\$] \mid \$ < 0\}$
44	19	`stack= new[...]`	*dead*(*stack*)	*dead*(*stack*)
45	20	*EnterStack1*	*dead*(*stack*)	*dead*(*stack*)
46	21	*DispatchConstructor*	*dead*(*stack*)	*dead*(*stack*)
47	22	*EnterStack*	*dead*(*stack*)	*dead*(*stack*)

In the following subsections, we briefly explain the iterations of our worklist algorithm. Some of the algorithm steps resemble the respective steps of the forward phase.

4.4.1 Iterations

The algorithm starts with the *Exit*<*Class*> supergraph node; here we know that none of the array elements are live. It continues to visit supergraph nodes until no more changes occur. When a supergraph node changes its predecessors are visited. When a node is processed before all its successors are known, the constraint graphs at the unknown successors are ignored.

After the effect of a supergraph node is taken into account by the algorithm, we add the strongest implied constraints as defined in Section 4.2. In Figure 8, the dotted edge between vertex 0 and vertex *top* corresponds to the strongest implied constraint between 0 and *top*.

4.4.2 Skip Supergraph Nodes

The algorithm skips nodes, which do not affect liveness information. For these skipped nodes, it simply copies the constraint graph after the node to the constraint graph before the node. These skipped nodes include:

- Nodes for statements, which do not assign to encapsulated variables and do not use or assign the elements of `A`.
- Special supergraph nodes, e.g., *EnterClass* and *DispatchMethod*.

In Table 4, the constraint graph before supergraph node 1 *ExitClass* at row 1 is obtained by copying *dead(stack)*.

4.4.3 Join

The join operation is defined exactly as in Section 4.3.3. However, the inclusion of $ has the effect that the join may enlarge the live region in the array. Also, in the backward phase the join is used to combine live regions along several outgoing paths in the approximated supergraph.

In Table 4, the constraint graph after supergraph node 2, *DispatchMethod*, at row 27 is obtained by joining the constraint graphs before node *EnterPrint* at row 15 and node EnterPop at row 26. The edge connecting vertex 0 to vertex $ and the edge connecting vertex 0 to vertex *top*are not included, since it appears only in the constraint graph from *EnterPrint* and not on the constraint graph from *EnterPop*.(Note that node 2 is a skip node, so although the constraint graph before node 2 is presented at row 37, it is equal to the constraint graph after node 2)

4.4.4 Integrating Forward Information

The liveness of an expression, $\texttt{A}[i]$, before supergraph node n, depends on two things (see Definition 4.1):

1. The value of i on supergraph paths from node *EnterClass* leading to node n, which determines the location l denoted by $\texttt{A}[i]$.
2. The usage of location l on supergraph paths emanating from node n.

Therefore, integrating the forward information regarding the value of i and the backward information regarding the liveness of $\texttt{A}[i]$ can have a dramatic impact on the precision of the analysis.

Despite the different nature of the forward and backward information, they can be integrated simply since both are represented by constraint graphs. The forward information is denoted by G_f, the backward information by G_b. We define their integration, $\mathit{integrate}(G_f, G_b)$, according to the following two rules:

1. $G_b = \mathit{dead(A)}$. In this case the forward information is irrelevant, since it does not provide any constraint on the usage of the array; thus, $\mathit{integrate}(G_f, \mathit{dead(A)})$ yields *dead(A)*.
2. Otherwise, for every pair of vertices x, y, $\mathit{integrate}(G_f, G_b)$ includes the strongest constraint of the strongest implied constraints between x and y in G_b and G_f.

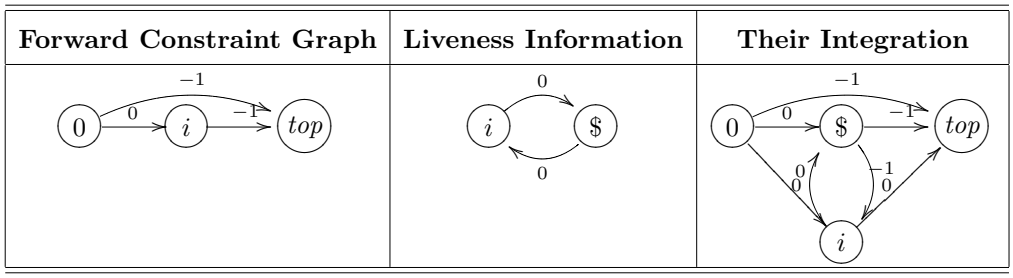


Figure 9: The integrated constraint graph.

In Table 4, the constraint graph before supergraph node 3, *ExitPrint*, at row 29 is obtained by integrating the liveness constraint graphs after node 3 (which is obtained by copying the constraint graph before node 2 *DispatchMethod* at row 28) and the resulting (i.e., after reaching a fixed-point) inequality constraint graph before node 3 at row 21 in Table 3. Three extra edges are added to the liveness constraint graph after the integration. Two edges, the 0 edge from vertex 0 to vertex i and the 0 edge from vertex *top* to vertex i are simply taken from the inequality constraint graph. The third -1 edge connection vertex \$ and vertex i is interesting, since a new liveness constraint (a constraint on \$) was added by integrating forward information.

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

In the current implementation the forward information (shown in in Table 3 is integrated with the resulting backward information before every supergraph node.

4.4.5 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(stack)`. The resulting constraint graph is shown in In Table 4 at row 6.

4.4.6 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 $\texttt{A}[i+c]$. However, constraint graphs cannot explicitly represent negations of equalities, and therefore we cannot

always exclude the liveness of `A`$[i+c]$. Hence, only constraints of the form $\$ \leq i + c$ or $i \leq \$ + (-c)$ can be strengthened to $\$ \leq i + (c-1)$ or $i \leq \$ + (-c-1)$, respectively.

In the constraint graph this corresponds to decrementing the c edge from vertex 0 to vertex i by 1 and incrementing the $-c$ edge from vertex i to vertex 0 by 1.

In Table 4, the constraint graph before supergraph node 15 `stack[top] = o` at row 40 is obtained by decrementing 1 from the 0 edge connecting $ vertex and *top* vertex in the constraint graph after node 15 (which is simply a copy of the constraint graph before node 14 at row 39).

A simpler case is the assignment to the whole array, e.g., `stack=new Object[len]`. In this case the resulting constraint graph is simply `dead(stack)`.

In Table 4, the constraint graph before supergraph node 19 `stack = new Stack[len]` at row 44 is set to `dead(stack)`.

4.4.7 Assignment Statements

For the statements `i = j + c` or `i = c`, the liveness information is obtained by substituting occurrences of i with $j + c$ or c, respectively. 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 the constraint graph, consider the statement `i = j + c`. First, edges leading to vertex i are incremented by c, and edges emanating from vertex i are decremented by c. Then, the edges leading to vertex i are removed and set to lead vertex j, unless $j \neq i$ and the edge already exists with a smaller weight. Similarly, edges emanating from vertex i are removed and set to emanate from vertex j, unless $j \neq i$ and the edge already exists with a smaller weight.

In Table 4, the constraint graph before supergraph node 14 `top = top + 1` at row 39 is obtained by adding 1 to the -1 edge connecting $ vertex and *top*vertex in the constraint graph after node 14 (which is simply a copy of the constraint graph before node 13 at row 38).

The statement `i = c` is handled similarly where vertex 0 plays the role of vertex j.

4.4.8 Widening

Widening is performed exactly as in the forward phase.

In Table 4, the constraint graph before supergraph node 4, `i < top` at row 11 is obtained by taking only strongest implied constraint from the constraint graph at row 7 (the former visit of node 4) that remain true in the constraint graph at row 10 (the current visit of node 4). The edge connecting vertex $ and vertex i is removed, since the constraint $\$ \leq i$ occurring at row 7 does not remain true (in the current visit it is replaced by a weaker constraint $\$ \leq i + 1$.

4.4.9 Conditions

Interestingly, 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[11].

Conditions of the form $i \leq j + c$ are handled simply by first creating a graph having one c edge from vertex i to vertex j, and then integrating it with the liveness information along the true edge. Similarly, creating a graph having one $-(c+1)$ edge from vertex j to vertex i, and then integrating it with the liveness information along the false edge. Finally, the two resulting constraint graphs are joined.

The condition `i` $\leq$ `c` is handled similarly where vertex 0 plays the role of vertex j. The condition `c` $\leq$ `i` is handled similarly where vertex 0 plays the role of vertex i, vertex i plays the role of vertex j, and $-c$ plays the role of c (since $c \leq i$ can be written as $0 \leq i + (-c)$).

In Table 4, the constraint graph before supergraph node 4, `i < top` at row 18 is obtained by integrating the liveness constraint graph along the true edge (node 6) at row 13 with the constraint $i \leq top + (-1)$, by integrating liveness constraint graph along the false edge (node 3) at row 17 with the constraint $top \leq i + 0$, and joining the two resulting constraint graphs. Note that the constraint $i \leq top + (-1)$ is already included in the constraint graph at row 13, and that the constraint $top \leq i + 0$ is already included in the constraint graph at row 17. Thus the resulting graph at row 18 is simply the join of the constraint graphs at row 13 and row 17.

5 *GC* Interface to Exploit Algorithmic Results

The analyses for array memory leaks can be exploited in two ways:

1. By instrumenting the program with assignments to `null` at the appropriate places, as discussed in section 1.

2. By providing information to *GC* so that it can determine the parts of an array that are alive.

We describe the second way below. It should provide better performance as no extra code is required during the execution of the program.

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

[11] It resembles Dijkstra's weakest preconditions [Dij76]. This analogy is beyond the scope of this paper

of a type-accurate collector [ADM98, DMH92]. 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`. If we were to allow a gc-point in this interval, then we would have to use `0` through `top` as the constraint; this would be less exact.

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 it may also have constraints for each array. Thus, 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 [Wil92] (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.

Notice that there are cases where an array maybe encountered before its encapsulating object, e.g., the `stack` array before its encapsulating `Stack` object. In this case the GC will trace the array without knowledge of its associated constraints; thus, the benefits of the analysis will be lost. However, given the assumptions of the analysis (e.g., that the array is encapsulated by the class and cannot be passed as a parameter to a method of another class), this could occur only if the thread is stopped for garbage collection in the middle of the execution of a method of the class so that a pointer to the array is on the thread's stack. Thus, we do not expect this to happen very often. If this turns out to be a problem in practice, we have an alternative solution where we annotate an array object with bit flag, indicating that it is to be traced according to a constraint, and a back-pointer to its encapsulating instance object.

6 Extensions

6.1 Disjunctive Completion

Sometimes, it occurs that joining constraint graphs (equivalently constraints sets), may result in an over conservative result. To overcome this, an immediate solution is to maintain sets of constraint graphs in every program point. In other examples that we have tested, it is sometimes necessary to use two sets of graphs per program points (but not more than two). This is a well-known tradeoff between better time and better space complexity oppose to getting a more accurate result [CC79].

```
void insertElementAt(Object obj, int i) {
   int newcount = count + 1;
   if (index >= newcount) {
      throw new ArrayIndexOutOfBoundsException(index
                                                + " > " + count);
   }
   if (newcount > elementData.length) {
      ensureCapacityHelper(newcount);
   }
   System.arraycopy(elementData, i, elementData, i + 1, count - i);
   elementData[i] = obj;
   count++;
}
```

Figure 10: `insertElementAt` method of `java.util.Vector` class.

For example, consider the method `insertElementAt` of class `java.util.Vector` shown in Figure 10. Figure 11 demonstrates the first visit of statement `elementData[i] = obj` during the backward phase. While in Figure 11(a) the constraint graph after the statement is conservatively copied (by ignoring the statement) to the constraint graph before the statement, using two constraint graphs, as in Figure 11(b) leads to a more accurate result. In particular, using one constraint graph per supergraph node, for analyzing the `Vector` class leads to an overly conservative result where all array elements are considered live before *DispachMethod* node, while using at most two sets of constraint graphs per supergraph node leads to an accurate, yet conservative, result so that the liveness region before *DispachMethod* node is $\{elementData[\$] | \$ < count\}$.

We have implemented this extension of the algorithm, maintaining sets of constraint in every supergraph node. The implementation was used to analyze dead array regions for `elementData` array in `java.util.Vector` class. It turned out that maintaining only one set of constraints for every supergraph node for the forward phase, and maintaining at most two sets of constraints for every supergraph node for the backward phase suffice to get a precise result.

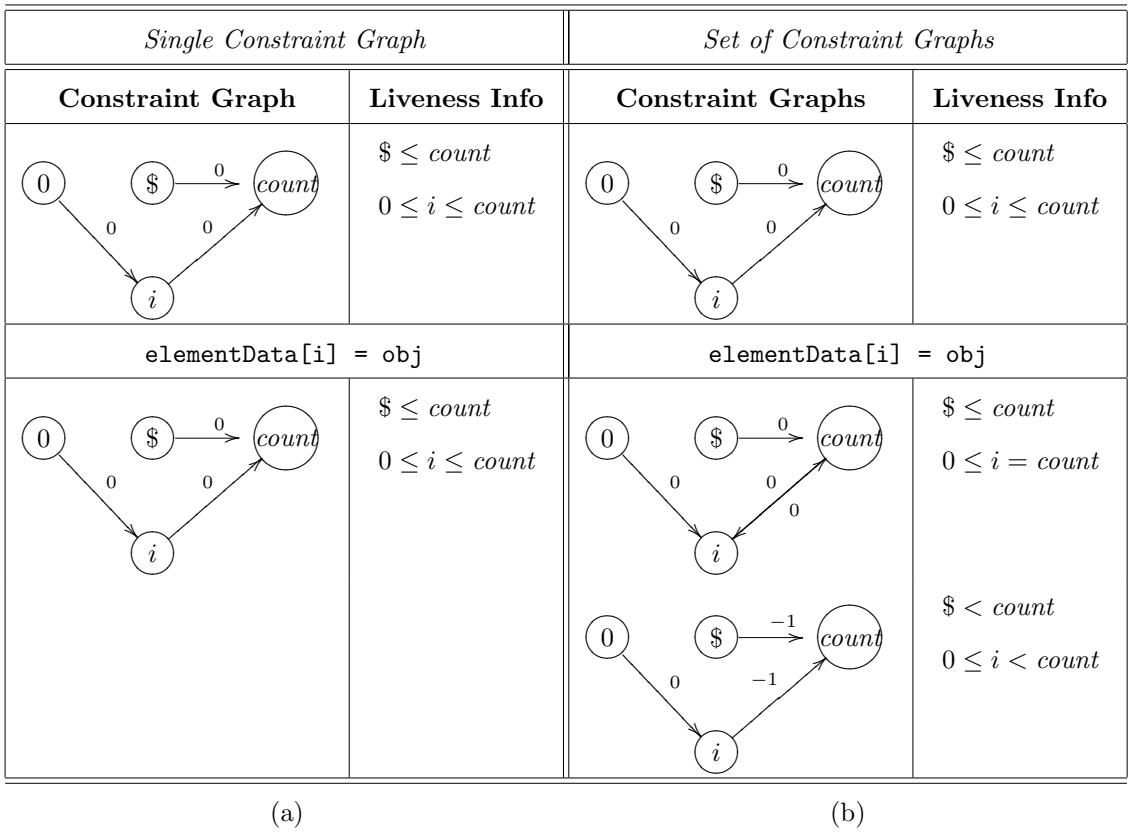


Figure 11: (a) The constraint graph before the statement `elementData[i]=obj` is a copy of the constraint graph after the statement. (b) Having several constraints graphs at a supergraph node, allows maintaining preciser liveness information before the statement `elementData[i]=obj`.

6.2 Reversing Cousot and Halbwachs Forward Analysis

Interestingly, it is shown here that a forward analysis for detecting affine relationships between program variables can be adopted to the backward phase of the algorithm, i.e. for analyzing the liveness regions of arrays. We will employ the same idea used in Section 4.4 by considering \$ as a new program variable (a new instance variable, for class-level analysis), and modeling the effect of a use of an array element and an assignment to an array element in terms of program variables affine relationships. The following is the extension the algorithm to handle affine functions of program variables. It is based on Cousot and Halbwachs [CH78] forward algorithm to automatically identify affine relationships between variables.

The extension is applicable, for example, for array-based abstract data types where the relations among program variables, used as array indices, are affine functions. An example is an array-based binary tree implementation, where for a node located in array element $\mathtt{A}[i]$, its left son and right son are located in array elements $\mathtt{A}[2i], \mathtt{A}[2i+1]$, respectively. The algorithm in Section 4, can not precisely express expressions such

as $2i$; thus a use of array element `A`$[2i]$ implies an overly conservative approximation of all array elements are live. Using the extension for affine functions enables handling such cases more precisely.

6.2.1 Brief Description of Cousot and Halbwachs algorithm framework

A system of affine inequalities of the form

$$\sum_{i=1}^{n}(a_{i,j}x_i) = c_j : j = 1 \ldots m$$

where $a_{i,j}, c_j \in Z$, and x_i are integer program variables, is maintained in every program point.

The set of solutions to a finite system of affine inequalities can be interpreted geometrically as the *closed convex polyhedron* of R^n defined by the intersection of the closed *halfspaces* corresponding to each inequality.

In [CH78] it is described how to convert from the representation of a polyhedron to a system of affine relationships and vice-verca.

The operations in a fixed point computing algorithm for finding the affine relationships among program variables are described in the abstract domain, either in terms of system of affine inequalities, or in terms of operations on polyhedrons, e.g., the *convex-hull* of two polyhedrons.

The forward phase of our algorithm is now exactly applying the algorithm of Cousot and Halbwachs. The following subsections describe how to extend the operations in the backward phase of our algorithm in terms of this framework. Note that an extra program variable, denoted \$, is now being considered. This implies that the system of affine inequalities maintained in every supergraph node in the forward phase, represent a polyhedron in R^n, where n is the number of integer program variables, and the system of affine inequalities maintained in every supergraph node in the backward phase, represent a polyhedron in R^{n+1}.

In addition, we also allow a special element, denoted by *dead*(`A`), for an analyzed array `A`.

Lastly, the extension allows more precise interpretation expressions of the form $\sum_{i=1}^{n}(a_i x_i) + c$; thus statements of the following form are considered :

- Assignment statements of the form $x_j = \sum_{i=1}^{n}(a_i x_i) + c$
- Conditions of the form $0 \leq \sum_{i=1}^{n}(a_i x_i) + c$
- Use of array elements of the form `A`$[\sum_{i=1}^{n}(a_i x_i) + c]$
- Assignment to an array element of the form `A`$[\sum_{i=1}^{n}(a_i x_i) + c]$

6.2.2 Skip Supergraph Nodes

The algorithm skips nodes, which do not effect liveness information. For these skipped nodes, it simply copies the system of affine inequalities after the node to the system of affine inequalities before the node.

6.2.3 Join

In the convex polyhedron representation, the join of convex polyhedrons $P_1, \ldots, P_n$ is a convex polyhedron P including $P_1, \ldots, P_n$. Join operation is described in [CH78, section 4.4].

6.2.4 Integrating Forward Information

In the system of inequalities representation, denote the forward information by S_f, and the backward information by S_b.

We define their integration, *integrate*(S_f, S_b), according to the following two rules:

1. S_b = *dead(A)*. In this case the forward information is irrelevant, since it does not provide any constraints on the usage of the array; thus, *integrate*$(S_f$, *dead(A)*$)$ yields *dead(A)*.

2. Otherwise, it is simply $S_f \wedge S_b$, i.e. the set of affine inequalities satisfying both systems of affine inequalities.

6.2.5 Use of an Array Element

For a statement using $\mathtt{A}[\sum_{i=1}^{n}(a_i x_i)+c]$, the algorithm enlarges the live region to include the current (forward) value of $\sum_{i=1}^{n}(a_i x_i) + c$. This means that the constraints on \$ are relaxed such that $\$ = \sum_{i=1}^{n}(a_i x_i) + c$ is satisfiable. First, we integrate the forward information, denoted by S_f, and the fact that $\mathtt{A}[\sum_{i=1}^{n}(a_i x_i) + c]$ is live. This is done by integrating the system of affine inequalities :

$$\$ - \sum_{i=1}^{n}(a_i x_i) \leq c$$
$$-\$ + \sum_{i=1}^{n}(a_i x_i) \leq -c$$

with S_f. Then, the resulting system of affine inequalities, converted to convex polyhedron is joined with the convex polyhedron after the statement to obtain the convex polyhedron (equivalently system of affine inequalities) before the statement.

6.2.6 Assignment to an Array Element

For a statement assigning to $\mathtt{A}[\sum_{i=1}^{n}(a_i x_i)+c]$, the algorithm can shrink the live region to exclude the current (forward) value of $\sum_{i=1}^{n}(a_i x_i) + c$. This means that the constraints on \$ can be made stronger to exclude the liveness of $\mathtt{A}[\sum_{i=1}^{n}(a_i x_i) + c]$.

A naive approach is constructing two systems of affine inequalities from S, the system of affine inequalities after the statement. The first system, S_1 is $S \wedge \{\$ - \sum_{i=1}^{n}(a_i x_i) \leq (c-1)\}$, and S_2 is $S \wedge \{-\$ + \sum_{i=1}^{n}(a_i x_i) \leq$

$-(c+1)\}$. Then, S_1 and S_2 are joined to obtain the resulting system of affine inequalities before the statement.

6.2.7 Assignment Statements

For the statements $i = \sum_{i=1}^{n}(a_i x_i) + c$, the liveness information is obtained by substituting occurrences of i with $\sum_{i=1}^{n}(a_i x_i) + c$ in the system of affine inequalities. The system of affine inequalities is then normalized.

6.2.8 Widening

Widening operation is described in [CH78, section 4.5]. For polyhedrons P_{k-1}, P_k, $P = P_{k-1} \bigtriangledown P_k$ is the convex polyhedron consisting in the affine inequalities of P_{k-1} verified by every element of P_k.

6.2.9 Conditions

Conditions of the form $0 \leq \sum_{i=1}^{n}(a_i x_i) + c$ are handled by propagating the system of affine inequalities after the condition node strengthened with $-\sum_{i=1}^{n}(a_i x_i) \leq c$ along the true edge, and by propagating the system of affine inequalities after the condition node strengthened with $\sum_{i=1}^{n}(a_i x_i) \leq 1 - c$ along the false edge.

6.3 Handling Protected Fields

The following more relaxed assumptions will allow our algorithm to analyze one class at a time under a relaxed compilation model, proposed in [BK97]. Under the relaxed compilation model it is assumed that the compiler generates two copies of the code, one under the assumption that derived classes will be presented later, and a second with the assumption that the generated class will not be used for inheritance. In the latter case our algorithm can be applied conservatively with the following relaxed assumptions :

- `A` and its class fields aliases are `private` or `protected` class fields.
- `A` and its class aliases (class fields or class method local variables) are not passed as a parameter to a method, unless the method is a method of the declaring class.

In general, for a well-written class file, the relaxed assumptions should hold.

7 Conclusion

Automatic memory management through a garbage collector makes programming significantly easier. However, memory leaks still exist in a garbage collected memory model. We have introduced compile time analysis detecting memory leaks occurring in arrays of objects, and showed how to expose the detected

memory leaks to the garbage collector, thus making a more exact GC, reducing the number of potential memory leaks.

Memory leaks in a garbage collected environment tend to have a negative effect on usability of such a memory management model. Further investigation is needed in order to statically detect more kinds of memory leaks occurring in practice.

References

[ADM98] 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.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Atlanta, October 1997.

[BK97] Z. Budimlic and K. Kennedy. Optimizing java: theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, june 1997.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherwight Sinchronization for Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 1998.

[Bla98a] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, San Diego, California, 19–21 January 1998.

[Bla98b] Bruno Blanchet. Escape analysis for object oriented languages. application to Java$^{\text{tm}}$. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Denver, 1998.

[BOSW98] 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.

[Bot99] Per Bothner. Kawa, the java-based scheme system. http://www.gnu.org/software/kawa/, 1999.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg Beach, Florida, 21–24 January 1996.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw.—Practice and Experience.*, 18(9):807–820, September 1988.

[C+99] Dominique Colnet et al. Smalleiffel, the gnu eiffel compiler. http://SmallEiffel.loria.fr/, 1999.

[CC79] 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.

[CDG97] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. Targeting gnat to the java virtual machine. In *TRI-ADA.*, pages 149–161, 1997.

[CGHS99] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Workshop on Prog. Analysis for Softw. Tools and Eng.*, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84– 96, Tucson, Arizona, January 1978.

[CLR94] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* M.I.T. Press, 1994.

[CS98] Robert Cartwright and Guy Steele. Compatible genericity with run-time types for the java programming language. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Vancouver, B.C., 1998.

[CV98] Morten V. Christiansen and Per Velschow. Region-based memory management in java. Master's thesis, University of Copenhagen, may 1998.

[Dij76] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[DMH92] 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.

[Flo67] R.W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Math.*, volume 19, pages 19–32, Providence R.I., 1967. AMS.

[Gup93] Rajiv Gupta. Optimizing array bound checks using flow analysis. *Let. on Prog. Lang. and Syst.*, 2(1–4):135–150, March–December 1993.

[Ham95] G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Memory Management, International Workshop IWMM 95*, 1995.

[JM90] Tomas P. Jensen and Torben Mogensen. A backward analysis for compile-time garbage collection. In *European Symp. on Prog.*, pages 227–239, 1990.

[Jon94] Simon B. Jones. An experiment in compile-time garbage collection. Technical report, University of Stirling, Scotland, 1994.

[Jon99] Richard Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1999.

[KS98] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *Symp. on Princ. of Prog. Lang.*, pages 107–120, 1998.

[KW95] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 270–278, 1995.

[L+81] Barbara Liskov et al. CLU reference manual. In *Lec. Notes in Comp. Sci.*, volume 114. Springer-Verlag, Berlin, 1981.

[Lis93] Barbara Liskov. A history of CLU. In *Hist of Prog. Lang. Preprints*, pages 133–147, 1993.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *Symp. on Princ. of Prog. Lang.*, pages 132–145, 1997.

[Moh90] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. In *plilp*, 1990.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Mye81] E.W. Myers. A precise inter-procedural data flow algorithm. In *Symp. on Princ. of Prog. Lang.*, pages 219–230, New York, NY, 1981. ACM Press.

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1992.

[Pug92] William Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Communications of the ACM*, August 1992.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang.*, pages 49–61, New York, NY, 1995. ACM Press. Available at "http://www.cs.wisc.edu/wpis/papers/popl95.ps".

[Sei98] Helmut Seidl. Compile-time garbage collection for object-oriented languages. Draft Manusctipt, 1998.

[SK98] Vivek Sarkar and Kathleen Knobe. Enabling sparse constant propagation of array elements via array ssa form. In *SAS'98, Static Analysis Symposium*, 1998.

[SP81] 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.

[SPE98] SPEC JVM98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[SPMS98] José E. Moreira Samuel P. Midkiff and Marc Snir. Optimizing array reference checking in java programs. *IBM Systems Journal*, 37(3), 1998.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[Web99] Adam Brooks Webber. Class-invariant binary relations in java. Draft Manusctipt, June 1999.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jaques Cohen, editors, *Memory Management, International Workshop IWMM 92*, volume 637 of *Lec. Notes in Comp. Sci.*, pages 1–42, St. Malo, France, September 1992. Springer-Verlag, Berlin.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[WR98] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Denver, 1998.