Memory Leaks Detection in Java by Bi-Abductive Inference

Dino Distefano and Ivana Filipović

Queen Mary University of London

Abstract. This paper describes a compositional analysis algorithm for statically detecting leaks in Java programs. The algorithm is based on separation logic and exploits the concept of bi-abductive inference for identifying the objects which are reachable but no longer used by the program.

1 Introduction

In garbage collected languages like Java the *unused memory* is claimed by the garbage collector, thus relieving the programmer of the burden of managing explicitly the use of dynamic memory. This claim is only partially correct: technically, the garbage collector reclaims only allocated portions of memory which have become unreachable from program variables, and often, this memory does not entirely correspond to the unused memory of the system. For instance, it is quite common that memory is allocated, used for a while, and then no longer needed nor used by the program. However, some of this memory cannot be freed by the garbage collector and will remain in the state of the program for longer than it needs to be, as there are still references to it from some program variables. Even though this phenomenon, typical of Java and other garbage collected languages like Python, defines a different form of "memory leakage" than in traditional languages like C, its results are equally catastrofic. If an application leaks memory, it first slows down the system in which it is running and eventually causes the system to run out of memory. Many memory-leak bugs have been reported (e.g., bug #4177795 in the Java Developer's Connection[13]) and experiments have shown that on average 39% of space could be saved by freeing reachable but unneeded objects [23, 21].

There are two main sources of memory leaks in Java code [12, 25, 17]:

- *Unknown or unwanted object references.* As commented above, this happens when some object is not used anymore, however the garbage collector cannot remove it because it is pointed to by some other object.
- *Long-living (static) objects.* These are objects that are allocated for the entire execution of the program.

These two possibilities appear in different forms. For example, a common simple error, such as forgetting to assign null to a live variable pointing to the object not needed anymore, leads to a memory leak. Such a leak can have serious consequences if the memory associated to it is substantial in size. Some more sophisticated examples discussed in literature are:

- *Singleton pattern, static references and Undounded caches.* The Singleton pattern, one of the most used object-oriented design patterns [6], ensures that a class has *only one* instance and provides a global access point to it. Once the singleton class is instantiated it remains in the program's memory until it is finished with its execution. However, the garbage collector will not be able to collect any of its referants, even though they might have a shorter lifetime than the singleton class [17]. Most caches are implemented using the Singleton pattern involving a static reference to a top level Cache class.
- *Lapsed listener methods.* Listeners are commonly used in Java programs in the Observer pattern [6]. Sometimes an object is added to the list of listeners, but it is not removed once it is no longer needed [12]. Here, the collection of listeners may grow unboundedly. The danger with such listener lists is that they may grow unboundendly causing the program to slow down since events are propagated to continuously growing set of listeners. Swing and AWT are very prone to this kind of problems.
- *Limbo.* Memory problems can arise also from objects that are not necessarily long-living but that occupy a consistent amount of memory. The problem occurs when the object is referenced by a long running method but it is not used. Until the method is completed, the garbage collector is not able to detect that the actual memory occupied by the object can be freed [7].

In this paper we propose a static analysis algorithm able to detect, at particular program points, the objects that are reachable from program variables but not further used by the program. This allows the possibility to free the unnecessary occupied memory. Our technique is based on the concept of *footprint*: that is, the part of memory that is actually used by a part of the program. Calculating the footprint of a piece of code singles out those allocated objects that are really needed from those that are not. The synthetization is done using *bi-abduction* [2], a recent static analysis technique which has been shown useful for calculating the footprint of large systems. Because it is based on bi-abduction our analysis is *compositional* (and therefore it has potential to scale for realistic size programs as shown in [2]) and it allows to reason about leaks for incomplete piece of code (e.g., a class or a method in isolation from others). This paper shows how bi-abduction is a valuable notion also in the context of garbage collection.

Throughout the paper we consider a running example given in Figure 1. The program uses a bag of integers and two observers for each bag, that register when an object is added to or removed from the bag, and consequently perform certain actions. The leaks here are due to live variables not being assigned null when they are no longer needed. Also, with the Observer pattern, a common mistake is not to remove the observers when they are no longer used. This is also illustrated by the example.

Fig. 1. Running example - Driver.java

```
import java.io.*;
import java.util.Iterator;
import java.util.ArrayList;

public class Driver {
      public static void main( String [] args ) {
  1.          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  2.          System.out.print("Enter the numbers, when finished enter -1 ");
  3.          IntegerDataBag bag = new IntegerDataBag();
  4.          IntegerAdder adder = new IntegerAdder( bag );
  5.          IntegerPrinter printer = new IntegerPrinter( bag );
  6.          Integer number = -1;
  7.          try{
              number = Integer.parseInt(br.readLine());;
          }
          catch  (IOException ioe) {
                   System.out.println("IO error trying to read input!");
                   System.exit(1);
          }
  8.          while (number >= 0) {
              try {
                  bag.add(number);
                  number = Integer.parseInt(br.readLine());
               } catch (IOException ioe) {
                   System.out.println("IO error trying to read input!");
                   System.exit(1);
               }
          }
  9.          bag.printBag();
 10.          ArrayList rlist = new ArrayList();
 11.         rlist = bag.reverseList();
 12.           IntegerDataBag revbag = new IntegerDataBag();
 13.          IntegerAdder adderr = new IntegerAdder( revbag );
 14.          IntegerPrinter printerr = new IntegerPrinter( revbag );
 15.          Iterator i = rlist.iterator();
 16.          while (i.hasNext()){
              revbag.add((Integer) i.next());
          }
 17.          Integer s,m;
 18.          s=revbag.sum();
 19.          m=revbag.mult();
 20.          System.out.print("The sum and the product are: "+s+" "+m+"\n");
      }
}
```

2 Informal Description of the Algorithm for Discovering Memory Leaks

Our algorithm for memory leak detection is two-fold. It runs two shape analyses[1]: a *forward* symbolic execution of the program and a *backwards* precondition calculation. The memory leak at each program point is obtained by comparing the results of the two analyses. More precisely:

1. For each method of each class (apart from the main method) we calculate its specifications. The specifications describe the minimal state necessary to run the method safely (i.e., without `NullPointerException`).
2. Using the results obtained in the previous step, we calculate the precondition of each subprogram of the main method. Here, the subprogram is defined with respect to the sequential composition. The calculation of the precondition of each subprogram is done in a backwards manner, starting from the last statement in the program. The results are saved in a table as (program location, precondition) pairs.
3. Using the forwards symbolic execution, intermediate states at each program point are calculated and added to the results table computed in step 2.
4. The corresponding states obtained in steps 2 and 3 are compared, and as the preconditions obtained by the backwards analysis are sufficient for safe execution of the program, any excess state that appears in the corresponding precondition obtained by the forwards analysis, is considered a memory leak.

3 Basics

In this section, we lay out some basic concepts, such as programming language, storage model and underlying logic used in this paper.

3.1 Programming Language

The programming language we consider here is a while java-like language [4].

$s ::= x = E \mid x.\langle C :\ t\ f\rangle = E \mid x = E.\langle C :\ t\ f\rangle \mid x = \mathsf{new}C(v) \mid \mathsf{return}\ E \mid$
$\mathsf{invoke}\ x.\langle C :\ t\ m\rangle(v) \mid x = \mathsf{invoke}\ y.\langle C :\ t\ m\rangle(v) \mid \mathbf{if}\ B\ \mathbf{then}\ c \mid \mathbf{while}\ B\ \mathbf{do}\ c$
$c ::= s \mid c; c$

Let $\mathsf{FN}, \mathsf{CN}, \mathsf{TN}$ and MN be countable sets of field, class, type and method names respectively. A signature of an object field/method is a triple $\langle C : t\ f\rangle \in \mathsf{CN} \times \mathsf{TN} \times (\mathsf{FN} \cup \mathsf{MN})$ indicating that the field f in objects of class C has type t. We denote a set of all signatures by Sig. Here, $E \in \mathsf{Pvar} \cup \{\mathsf{nil}\}$ and Pvar is a countable set of program variables ranging over $x, y, \ldots$. Basic commands include assignement, mutation, lookup, allocation, return from a method and method invocation. A programs consist of basic commands, composed by the sequential composition.

[1] Shape analyses, introduced in [20], are program analyses that establish deep properties of the program heap such as a variable point to a cyclic/acyclic linked list.

3.2 Storage Model and Symbolic Heaps

We describe the storage model and a fragment of separation logic formulae, suitable for symbolic execution [1, 3], which plays an important role in our work.

Let *LVar* (ranged over by $x', y', z', \ldots$) be a set of logical variables, disjoint from program variables *PVar*, to be used in the assertion language. Let *Locs* be a countably infinite set of locations, and let *Vals* be a set of values that includes *Locs*. The storage model is given by:

$$\mathit{Heaps} \overset{\mathit{def}}{=} \mathit{Locs} \rightharpoonup_{\mathsf{fin}} \mathit{Vals} \qquad \mathit{Stacks} \overset{\mathit{def}}{=} (\mathit{PVar} \cup \mathit{LVar}) \to \mathit{Vals}$$
$$\mathit{States} \overset{\mathit{def}}{=} \mathit{Stacks} \times \mathit{Heaps}$$

Program states are symbolically represented by special separation logic formulae called *symbolic heaps.* They are defined as follows:

$$\begin{array}{lcll}
E & ::= & x \mid x' \mid \mathsf{nil} & \textit{Expressions} \\
\Pi & ::= & E{=}E \mid E{\neq}E \mid \mathsf{true} \mid p(\overline{E}) \mid \Pi \wedge \Pi & \textit{Pure formulae} \\
S & ::= & s(\overline{E}) & \textit{Basic spatial predicates} \\
\Sigma & ::= & S \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Spatial formulae} \\
H & ::= & \exists \boldsymbol{x'}.\, (\Pi \wedge \Sigma) & \textit{Symbolic heaps}
\end{array}$$

Expressions are program or logical variables x, x' or nil. Pure formulae are conjunctions of equalities and disequalities between expressions, and abstract pure predicates $p(\overline{E})$ and describe properties of variables. They are not concerned with heap allocated objects. Spatial formulae specify properties of the heap. The predicate emp holds only in the empty heap where nothing is allocated. The formula $\Sigma_1 * \Sigma_2$ uses the separating conjunction of separation logic and holds in a heap h which can be split into two *disjoint parts* H_1 and H_2 such that Σ_1 holds in H_1 and Σ_2 in H_2. In symbolic heaps some (not necessarily all) logical variables are existentially quantified. The set of all symbolic heaps is denoted by SH. In the following we also use a special state fault, different from all the symbolic heaps, to denote an error state. S is a set of basic spatial predicates. The spatial predicates can be arbitrary abstract predicates [18]. In this paper, we mostly use the following instantiations of the abstract predicates $x.\langle C :\ t\ f\rangle \mapsto E$, $\mathsf{ls}(E, E)$ and $\mathsf{lsn}(E, E, E)$. The *points-to* predicate $x.\langle C :\ t\ f\rangle \mapsto E$ states that the object denoted by x points to the value E by the field f. We often use the notation $x.f \mapsto E$ when the class C and type t are clear from the context. Also, if the object has only one field, we simplify notation by writing $x \mapsto \_$. Predicate $\mathsf{ls}(x, y)$ denotes a possibly empty list segment from x to y (not including y) and it is defined as:

$$\mathsf{ls}(x, y) \iff (x = y \wedge \mathsf{emp}) \ \vee\ (\exists x'.x \mapsto x' * \mathsf{ls}(x', y))$$

Predicate $\mathsf{lsn}(O, x, y)$ is similar to $\mathsf{ls}(x, y)$, but it also keeps track of all the elements kept in the list. This is done by maintaining a set O of all the values.

$$\mathsf{ls}(O, x, y) \iff (x = y \wedge \mathsf{emp} \wedge O = \emptyset) \vee$$
$$(\exists x', o', O'.\mathit{union}(o', O') = O \wedge x \mapsto o', x' * \mathsf{ls}(O', x', y))$$

Here *union* is an abstract predicate indicating the union of its arguments. In this paper we sometimes do not write the existential quantification explicitly, but in that case, we keep the convention that primed variables are implicitly existentially quantified. Also, we use a field splitting model, i.e., in our model, objects are considered to be a compound entities composed by fields which can be split by $*$[2]. Notice that if S_1 and S_2 describe the same field of an object than $S_1 * S_2$ implies false.

Here, it is worth to mention a fundamental rule which gives the bases of local reasoning in separation logic:

$$\frac{\{H_1\}\ C\ \{H_2\}}{\{H_1 * H\}\ C\ \{H_2 * H\}}\ \text{Frame Rule}$$

where C does not assign to H's free variables [15]. The frame rule allows us to circumscribe the region of the heap which is touched by C, (in this case H_1), perform local surgery, and combine the result with the frame, i.e. the part of the heap not affected by the command C (in this case H).

3.3 Bi-abduction

The notion of *bi-abduction* was recently introduced in [2]. It is the combination of two dual notions that extend the entailment problem: *frame inference* and *abduction.*

Frame inference [1] is the problem of determining a formula $\mathfrak{F}$ (called the *frame*) which we need to add to the conclusions of an entailment in order to make it valid. More formally,

Definition 1 (Frame inference). *Given two heaps H and H' find a frame $\mathfrak{F}$ such that $H \vdash H' * \mathfrak{F}$.*

In other words, solving a frame inference problem means to find a description of the extra parts of heap described by H and not by H'.

Abduction is dual to frame inference. It consists of determining a formula $\mathfrak{A}$ (called the *anti-frame*) describing the pieces of heap missing in the hypothesis and needed to make an entailment $H * \mathfrak{A} \vdash H'$ valid. Abduction was introduced in the context of scientific process as a means to distinguish the process of hypothesis formation from deductive and inductive inference [19]. In this paper we use abduction in the very specific context of separation logic.

Bi-abduction is the combination of frame inference and abduction. It consists in deriving at the same time interdependent frames and anti-frames.

Definition 2 (Bi-Abduction). *Given two heaps H and H' find a frame $\mathfrak{F}$ and an anti-frame $\mathfrak{A}$ such that $H * \mathfrak{A} \vdash H' * \mathfrak{F}$*

[2] An alternative model would consider the granularity of $*$ at the level of objects. In that case, objects cannot be split by $*$ since they are the smallest unit in the heap.

Table 1. Algorithm 1 LeakDetectionAlgorithm(Prg)

```
$Plocs := LabelPgm(1, Prg)$;
$Mspecs := CompSpecs()$;
$LocPre := ForwardAnalysis(Mspecs)$;
$LocFp := BackwardAnalysis(Mspecs)$;
forall $loc \in Plocs$ do
    $Pre := LocPre(loc)$;
    $Fp := LocFp(loc)$;
    $MLeak(loc) := \{R \mid H_1 \vdash H_2 * R \wedge H_1 \in Pre \wedge H_2 \in Fp\}$
end for
```

Many solutions are possible for $\mathfrak{A}$ and $\mathfrak{F}$. A criterion to judge the quality of solutions as well as a bi-abductive prover were defined in [2]. In this paper we use bi-abduction to find memory leaks in Java programs.

Throughout the paper we will write the frame and anti-frame to be determined in the bi-abduction problem in "frak" fonts (e.g., $\mathfrak{A}, \mathfrak{F}, \mathfrak{B} \dots$) in order to distinguish them from the known parts of the entailment.

4 Detecting Memory Leaks

Algortihm 1 computes allocated objects that can be considered memory leaks, at particular program points.

Firstly, the program is labelled using the *LabelPgm*() function. The labelling function is described in more details in Section 4. Secondly, the specs of all the methods in the program are computed using the function *CompSpecs*(). Using these specs, a symbolic execution (forward analysis) of the program is performed *ForwardAnalysis*() (see Section 4.1). The result of the analysis are assertions, obtained by symbolically executing the program which represent an over-approximation of all the possible states the program can be at a location.

These assertions, together with the program locations to which they correspond, are recorded in an array *LocPre*. Next, a backward analysis *BackwardAnalysis*() is performed, again using the calculated specs of the methods. At each program point an assertion is obtained, that represents a preconditions for a subprogram starting at that program location. These results are written in an array *LocFp* indexed by the locations. Finally, for each program point, the results, i.e. the preconditions obtained in these two ways are compared by solving a frame inference problem. The solution frame corresponds to the memory leaked at that location.

Labelling program points The program is labeled only at *essential* program points. A program point is considered essential only if

- it is a basic command not enclosed within a **while** or **if** statement,
- or, if it is the outer-most **while**-statement or the outer-most **if**-statement.

This means that we do not consider essential those statements within the body of **while** and **if** statements, either basic or compound. Function *LabelPgm*,

$$\mathit{LabelPgm}(i,s) = (i:\ s) \qquad \mathit{LabelPgm}(i,s;c) = (i:\ s); \mathit{LabelPgm}(i+1,c)$$

takes a program and an integer, and returns a labelled program. We labelled our running example (Table 1) according to the labelling algorithm. Memory leaks are sought for only at the essential program locations. The rationale behind this choice can be understood as follows. If a new unnamed cell is assigned in each iteration to a variable then the garbage collector can claim the object before the iteration during the execution of the loop (if there are no references to it). For example this is the case of the `Integer.parseInt(br.readLine())` in the body of the while loop at location 8 in Fig. 1. The other possibility is when objects used in the body of the **while**-loop are potentially used in each iteration and could become a memory leak only upon the exit from the loop; for example a data structure is created, traversed or manipulated during the execution of the loop. Such structure is not a leak as long as the loop is executing (for example the `bag` in the body of the loop at location 8). Only if the structure is not used anymore after the loop has terminated, but the variable holding the structure is not set to null, then it is considered to be a leak and should be detected.

4.1 Forward and Backward Shape Analyses

Our algorithm is based on the use of two existing shape analyses [3, 2] for which we provide brief and rather informal summary.

Forward Shape analysis. The forward shape analylsis consists of three main steps: symbolic execution, heap abstraction and heap rearrangement. Symbolic execution implements a function

$$\mathsf{exec} : \mathit{Stmts} \times \mathsf{SH} \to \mathcal{P}(\mathsf{SH}) \cup \{\mathsf{fault}\}.$$

It takes a statement and a heap and returns a set of resulting heaps after the execution of the statement or the special element fault indicating that there is a possible error. For example, the result of the execution of a statement $x.\langle C\colon t\ f\rangle = E_2$, which assigns value E_2 to the field f of object x, in a heap $\mathsf{H} * x.\langle C\colon t\ f\rangle \mapsto E_1$ is $\mathsf{H} * x.\langle C\colon t\ f\rangle \mapsto E_2$.

Abstraction is done by rewriting rules, also called *abstraction rules* which implement the function

$$\mathsf{abs} : \mathsf{SH} \to \mathsf{SH}$$

The abstraction rules are applied after the execution of any command, which helps to keep the state space small.

The rules of symbolic execution work at the level of the object fields which is the most basic entity considered in the analysis. In other words, the rules manipulate only points to predicate $\mapsto$, but they cannot be applied to composite abstract predicates or inductive predicate like $\mathsf{ls}(x,y)$. In case the field object that need to be accessed by symbolic execution is hidden inside one of these composite/inductive predicates, rearrangement is used to expose this field. Rearrangement implements function

$$\mathsf{rearr} : \mathit{Heaps} \times \mathit{Vars} \times \mathit{Sig} \to \mathcal{P}(\mathsf{SH})$$

Forward shape analysis can be defined as the composition of rearrangement, symbolic execution and abstraction

$$\mathcal{F} = \mathsf{abs} \circ \mathsf{exec} \circ \mathsf{rearr}.$$

The forward analysis is *sound* since it computes, at any program point, an over-approximation of the set of all states in which the program can be in any possible run [3]. Complete formal description of the forward shape analysis used here, as well as the tool jStar implementing it, can be found in [3, 4].

Compositional backward shape analysis. Backward shape analysis is achieved using bi-abduction which allows to construct the analysis in a compositional fashion. Such analysis can be seen as the attempt to build proofs for Hoare triples of a program. More precisely, given a class composed of methods $m_1(\boldsymbol{x_1}), \ldots, m_n(\boldsymbol{x_n})$ the proof search automatically synthesizes preconditions $P_1, \ldots, P_n$ and postcondition $Q_1, \ldots, Q_n$ such that the following are valid Hoare triples:

$$\{P_1\}\, m_1(\boldsymbol{x_1})\, \{Q_1\}, \ldots, \{P_n\}\, m_n(\boldsymbol{x_n})\, \{Q_n\}$$

The triples are constructed by symbolically executing the program and by composing existing triples. The composition (and therefore the construction of the proof) is done in a bottom-up fashion starting from the leaves of the call-graph and then using their triples to build other proofs for methods which are on a higher-level in the call-graph. To achieve that, the following rule for sequential composition —called the Bi-Abductive Sequencing Rule— is used [2]:

$$\frac{\{P_1\}\, C_1\, \{Q_1\} \qquad \{P_2\}\, C_2\, \{Q_2\}}{\{P_1 * \mathfrak{A}\}\, C_1; C_2\, \{Q_2 * \mathfrak{F}\}}\; Q_1 * \mathfrak{A} \vdash P_2 * \mathfrak{F} \qquad (\mathsf{BA\text{-}seq})$$

This rule is also used to construct a proof (triple) of a method body in compositional way. In that case the specifications that are used refer to commands (e.g., statements) or (previously proved) methods in case of a method call. BA-seq can be used to analyze the program either composing specification "going forward" or "going backward". In our case, we use it as a core rule for the definition of our backward analysis.[3] A tool implementing bi-abductive analysis exists and it is described in [2].

[3] In the special case of while-loop the rule is used in a forward way combined with the abstraction mechanism which ensure convergence of the analysis [2].

Forward and Backward analyses in action. In this section we exemplify forward and backward analysis by applying them to an example. Let us consider a program consisting of three labelled commands.

```
1:  OneFieldClass x = new OneFieldClass();
2:  OneFieldClass x = new OneFieldClass();
3:  x.update(val).
```

For succinctness, let us denote the statements above as c_1, c_2 and c_3. The specifications of the statements are given bellow.

$$\{\mathsf{emp}\}c_1\{x \mapsto \_\} \qquad \{\mathsf{emp}\}c_2\{y \mapsto \_\} \qquad \{x \mapsto \_\}c_3\{x \mapsto val\}$$

In forward analysis, the program is executed symbolically, starting from an empty state. During the execution the memory is accumulated in the program state and a post-state of each statement is a pre-state of the following statement. Let us first consider what assertions at each program point we get by executing the forward analysis.

$$\{\mathsf{emp}\}c_1\{x \mapsto \_\}c_2\{x \mapsto \_ * y \mapsto \_\}c_3\{x \mapsto val * y \mapsto \_\}$$

We observe that the preconditions for the corresponding program points are as following:

$$1 : \mathsf{emp} \qquad 2 : x \mapsto \_ \qquad 3 : x \mapsto \_ * y \mapsto \_.$$

Let us now consider what happens when we combine the triples using the Bi-abductive sequencing rule in a backwards manner. Firstly, the triples of the last two labelled statements in the program are combined, and a new triple for the subprogram consisting of these two statements is obtained. That triple is used further to be combined with the previous statement in the program, and so on, until the beginning of the program is reached. If we apply the rule to specifications for c_2 and c_3, we get

$$\frac{\{\mathsf{emp}\}\, c_2\, \{y \mapsto \_\} \qquad \{x \mapsto \_\}\, c_3\, \{x \mapsto val\}}{\{x \mapsto \_\}\, c_2; c_3\, \{x \mapsto val * y \mapsto \_\}}\; y \mapsto \_ * x \mapsto \_ \vdash x \mapsto \_ * y \mapsto \_ .$$

Here, $\mathfrak{A} = x \mapsto \_$ and $\mathfrak{F} = y \mapsto \_$. Now, we combine the obtained triple for $c_2; c_3$ with the triple for c_1.

$$\frac{\{\mathsf{emp}\}\, c_1\, \{x \mapsto \_\} \qquad \{x \mapsto \_\}\, c_2; c_3\, \{x \mapsto val * y \mapsto \_\}}{\{\mathsf{emp}\}\, c_1; c_2; c_3\, \{x \mapsto val * y \mapsto \_\}}\; \mathsf{emp} * \mathsf{emp} \vdash x \mapsto \_ * y \mapsto \_ * \mathsf{emp} .$$

Here, $\mathfrak{A} = \mathsf{emp}$ and $\mathfrak{F} = \mathsf{emp}$. In this case, the preconditions for the corresponding program points are

$$1 : \mathsf{emp} \qquad 2 : x \mapsto \_ \qquad 3 : x \mapsto \_$$

Note that in the backward analysis state is accumulated in the postcondition. However, this does not pose the problem as it is the precondition that describes what state is necessary for safely running the program, while the postcondition describes what is accumulated after the execution is finished (when starting from the inferred precondition).

Soundness of the algorithm. We now show that our algorithm is sound in the sense that it only classify as leaks a subset of those parts of memory which are allocated but not used anymore.

Let c be a command and H symbolic heap. We say that c is $s$afe at H (written $\mathsf{safe}(c, H)$) when the following holds

$$\mathsf{safe}(c, H) \text{ iff } \mathsf{fault} \notin \mathcal{F}(c, \{H\}).$$

Intuitively, c is safe at H when running the symbolic execution using H as precondition does not reach the faulting state.

Theorem 1. *The LeakDetectionAlgorithm only identifies real leaks.*

Proof. (Sketch) Let $R \in MLeak(loc)$, then there exists $H_1 \in LocPre(loc)$ and $H_2 \in LocFp(loc)$ such that $H_1 \vdash H_2 * R$. By definition H_2 is the precondition of a Hoare triple constructed by the backwards analysis. First of all, note that the backward analysis constructs only true triples [2]. Secondly, because of the tight interpretation of triples in separation logic[15], we have that a triple ensures that the program starting from the precondition cannot fault. Therefore, this implies that H_2, being a precondition of a true triple, must be safe. This in turn implies that the program starting at loc only accesses memory cells described by H_2 and it does not access any memory defined by R (otherwise H_2 could not be safe). Hence R is a real leak. □

5 Examples

In this section we illustrate how our algorithm works on several examples. Firstly, we revisit our running example given in Table 1 and show in detail how our algorithm operates on actual code. Then, we examine two more examples that reflect other causes of memory leaks discussed in introduction.

5.1 Running example

Our algorithm first applies the two analyses to our example. The results of the analyses is given in App. A. All the necessary specification of the underlying classes in our example are given in App. B. Here, we compare these results and infer which portion of the program state can be considered a memory leak.

At label 1 of the program, the precondition obtained in both forward and backward analysis is emp, and so there is no memory leak before the execution of the program has started, as expected. In fact, class Driver does not leak any memory upto the label 9.

Let us now consider what happens at label 9 of the program. Forward analysis finds that symbolic state

$$\begin{aligned}\exists O.\ & br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * \\ & (\forall * o \in O.o.bag \mapsto bag)\end{aligned}$$

describes a precondition at label 9. This precondition is a result of the symbolic execution of the program upto label 9, and so, it reflects the actual program state, i.e. this precondition contains all the memory allocated and reachable in the execution of the program so far.

Backward analysis, on the other hand, calculates that the precondition at this point is

$$bag.list \mapsto x' * ls(x', nil).$$

Backward analysis, as already discussed, pinpoints the exact memory necessary for safe execution of the program. So the subprogram starting at label 9 needs nothing more and nothing less than this precondition in order to execute safely (without crashing).

Our algorithm now uses frame inference to compares these two preconditions and concludes that the state

$$\exists O.\ br \mapsto \_ * bag.observers \mapsto y' * ls(O, y', nil) * (\forall * o \in O.o.bag \mapsto bag)$$

is not necessary for the execution of the rest of program, and hence, it is a leak.

During the execution of the program memory accumulates unless it is explicitelly freed, by say, setting certain variables to null and waiting for the garbage collector to reclaim the objects that are no longer refered to by variables. In our running example, no memory is freed, and the most dramatic memory leak appears towards the end of the program. At label 18 of our program, forward analysis produces symbolic state

$$\begin{aligned}\exists O, O'.\ & br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * \\ & (\forall * o \in O.o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * \\ & revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'.o.bag \mapsto bag)\end{aligned}$$

as a precondition. However, the backward analysis finds the precondition corresponding to the same label to be

$$revbag.list \mapsto x' * ls(x', nil).$$

This leaves a substantial ammount of memory to lie around in the program state, while it is not needed by the program:

$$\begin{aligned}\exists O, O'.\ & br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * \\ & (\forall * o \in O.o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) \\ & * revbag.observers \mapsto v' * ls(O', v', nil) * (\forall * o \in O'.o.bag \mapsto bag).\end{aligned}$$

5.2 Examples on other sources of leaks

We now point out examples that demonstrate some of the possible causes of memory leaks discussed in the introduction.

The following example illustrates a memory leak caused by a static reference. Here, we have a huge static object LinkedList which is allocated when the program starts executing. Even though it is not used anymore after a certain point in the program, because it is not explicitely set to null, the garbage collector will not be able to reclaim its memory, as it is referenced by a static variable.

```
public class Myclass {
  static LinkedList myContainer = new LinkedList();
  public void leak(int numObjects) {
    for (int i = 0; i < numObjects; ++i) {
      String leakingUnit = new String("this is leaking object: " + i);
      myContainer.add(leakingUnit);}
    }
  public static void main(String[] args) throws Exception {
    {
      Myclass myObj = new Myclass();
      myObj.leak(100000); // One hundred thousand
    }
    System.gc();
    // do some other computation not involving myObj}
  }
```

Specifications of the methods are as follows.

$$\begin{array}{ll}
\{\mathsf{emp}\} & \{this.myContainer \mapsto x' * ls(x', nil)\} \\
myClass(); & leak(i) \\
\{this.myContainer \mapsto x'\} & \{this.myContainer \mapsto x' * ls(x', nil)\}
\end{array}$$

The forward analysis applied to the main() method, yields

$$\begin{array}{l}
\{MyClass.myContainer \mapsto x' \ \wedge \ x' = nil\} \\
MyClass\ myObj = new\ Myclass(); \\
\{myObj.myContainer \mapsto x' \ \wedge \ x' = nil\} \\
myObj.leak(100000); \\
\{MyClass.myContainer \mapsto x' * ls(x', nil)\} \\
\{MyClass.myContainer \mapsto x' * ls(x', nil)\} \\
System.gc(); \\
\{MyClass.myContainer \mapsto x' * ls(x', nil)\} \\
//do\_some\_other\_computation\_not\_involving\_myContainer \\
\{p * MyClass.myContainer \mapsto x' * ls(x', nil)\}
\end{array}$$

Here, p denotes some predicate that describes the postcondition of the program and does not mention any memory described by $MyClass.myContainer \mapsto x' * ls(x', nil)$. On the other hand, since the program does not use any memory referenced by MyContainer upon the exit from the local block, the backward analysis will find that MyContainer is last used inside this local block, and so our algorithm will discover that at the end of the local block memory referenced by MyContainer is leaked.

The last example we consider illustrates the phenomenon of Limbo, discussed in the introduction. The program first allocates a very big list and does some computation over the elements of the list. Then, it starts handling some input, which might last for very long (possibly forever). At the end of the main() method, memory referenced by the list would be garbage collected, but as the input handling might last very long, this could lead to running out of memory.

```
public static voin main(String args[]){
  int big_list = new LinkedList();
  //populate the list
  populate(big_list);
  // Do something with big_list
  int result=compute(big_list);
  //big_array is no longer needed but it cannot be garbage collected
  //we would need to set its reference to null explicitely
  for (;;) handle_input(result);
}
```

The forward analysis of the main() returns the following assertions, assuming that for handling input, no memory is needed.

$$
\begin{array}{l}
\{\mathsf{emp}\}\\
int\ big\_list = new\ LinkedList();\\
\{big\_list \mapsto x' \ \wedge x' = nil\}\\
populate(big\_list);\\
\{big\_list \mapsto x' * ls(x', nil)\}\\
intresult = compute(big\_list);\\
\{big\_list \mapsto x' * ls(x', nil)\}\\
for(;;)handle_{i}nput(result);\\
\{big\_list \mapsto x' * ls(x', nil)\}
\end{array}
$$

Our backward analysis discovers that the last point `big_list` is used is in a `int result=compute(big_list);` statement, and that is where our algorithm discovers that a program leaks memory referenced by this variable.

6 Related Work

The paper [16] introduces a backwards static analysis which tries to disprove the assumption that the last statement has introduced a leak. If a contradiction is found, then the original assumption of the leak was wrong. Otherwise, the analysis reports a program trace that leads to the assumed error. Like ours, this analysis allows to check incomplete code. However, it can only detect memory objects that are not referenced anymore, therefore this analysis is not suitable for detecting the kind of leaks (Java leaks) we are concerned with in this paper. The same limitation applies to the techniques described in [11, 8]. Similarly, the static analyses described in [5, 26] aims at detecting leaks caused by objects not reachable from program variables. Therefore they cannot detect the kind of leaks we aim at with our analysis.

The paper [21] introduces a static analysis for finding memory leaks in Java. This technique is tailored for arrays of objects. On the contrary, here we have defined a framework which works for different kind of data structures representable by abstract predicates.

A static analysis for detecting unused (garbage) objects is introduced in [24]. This analysis is similar to ours in its aim. However, the two approaches are substantially different. The authors use finite state automata to encode safety properties of objects (for example "the object referenced by y can be deallocated at line 10"). The global state of program is represented by first-order logical structures and these are augmented with the automaton state of every heap-allocated object. This shape analysis is *non* compositional and works globally. Our technique instead is compositional (since based on bi-abduction) and exploits local reasoning (since based on separation logic). Compositional shape analyses based on bi-abduction and separation logic have a high potential to scale as demonstrated in [2]. Moreover, their approach employs an automaton for each property at a program point, whereas our approach simultaneously proves properties for many objects at all essential program points in a single run of the algorithm.

Different from static approaches as the above and ours there are dynamic techniques for memory leak detection [9, 14, 10, 22]. The main drawback with dynamic techniques is that they cannot give guarantees. Leaks that do not occur in the particular run which is checked will be missed and remain hidden in the program.

7 Conclusion

Allocated but unused objects reachable from program variables cannot be reclaimed by the garbage collector. These objects can be effectively considered memory leaks since they often produce the same catastrophic problems that leaks have in languages like C: applications irreversibly slow down until they run out of memory. In this paper we have defined a static analysis algorithm which allows the detection of such allocated and unused objects which cannot be freed by the garbage collector. Our technique exploits the effectiveness of separation logic to reason locally about dynamic allocated data structures and the power of bi-abductive inference to synthesize the part of allocated memory truly accessed by a piece of code. The paper shows how separation logic based program analyses and bi-abductive inference can be combined to reason statically about memory leaks in garbage collected languages like Java. We have shown the effectiveness of our algorithm on examples involving different sources of leakage among which the Observer pattern, that is one of the most used design patterns in real life.

References

1. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.
2. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
3. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
4. D. Distefano and M. J. Parkinson. jstar: towards practical verification for java. In *OOPSLA*, pages 213–226, 2008.

5. N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *SAS*, pages 115–134, 2000.
6. E.Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
7. D. Flanagan. *Java in a Nutshell.* O'Really, 1996.
8. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, 2005.
9. R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
10. M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.
11. D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI*, pages 168–181, 2003.
12. I.Poddar and R.J.Minshall. Memory leak detection and analysis in webshere application server (part 1 and 2). Internet page, 2006. Available at http://www.ibm.com/developerworks/websphere/library/techarticles/0608_poddar/0608_poddar.html.
13. The java developer's connection. Internet page. Available at http://bugs.sun.com/bugdatabase.
14. N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *ECOOP*, pages 351–377, 2003.
15. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, 2001.
16. M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *SAS*, pages 405–424, 2006.
17. A. Pankajakshan. Plug memory leaks in enterprise java applications. Internet page, 2006. Available at http://www.javaworld.com/javaworld/jw-03-2006/jw-0313-leak.html.
18. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86, 2008.
19. C. Peirce. *Collected papers of Charles Sanders Peirce.* Harvard University Press., 1958.
20. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.
21. R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic removal of array memory leaks in java. In *CC*, pages 50–66, 2000.
22. R. Shaham, E. K. Kolodner, and S. Sagiv. Heap profiling for space-efficient java. In *PLDI*, pages 104–113, 2001.
23. R. Shaham, E. K. Kolodner, and S. Sagiv. Estimating the impact of heap liveness information on space consumption in java. In *MSP/ISMM*, pages 171–182, 2002.
24. R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *Sci. Comput. Program.*, 58(1-2):264–289, 2005.
25. V.B.Livshits. Looking for memory leaks. Internet page. Available at http://www.oracle.com/technology/pub/articles/masterj2ee/j2ee_wk11.html?_template=/ocom/print.
26. Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/SIGSOFT FSE*, pages 115–125, 2005.

A Results of the analyses of the class Driver

In this section we present the assertions for the class Driver obtained by the forward and backward analysis, respectively. The numbers correspond to the labels in the program, and each formula specifies the precondition for a subprogram starting at given label.

A.1 Forward analysis

1. emp
3. $br \mapsto \_$
4. $br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(0, y', nil)$
5. $br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(adder, y', nil) * adder.bag \mapsto bag$
8. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag)$
9. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag)$
10. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag)$
11. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil)$
12. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil)$
13. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(0, v', nil)$
14. $\exists O. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(adderr, v', nil) * adderr \mapsto revbag$
15. $\exists O, O'. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'. o.bag \mapsto bag)$
16. $\exists O, O'. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', i) * ls(i, nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'. o.bag \mapsto bag)$
18. $\exists O, O'. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'. o.bag \mapsto bag)$
19. $\exists O, O'. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'. o.bag \mapsto bag)$

20 $\exists O, O'. br \mapsto \_ * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall * o \in O. o.bag \mapsto bag) * rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto u' * revbag.observers \mapsto v' * ls(u', nil) * ls(O', v', nil) * (\forall * o \in O'. o.bag \mapsto bag)$

A.2 Backward analysis

1. emp
3. $br \mapsto \_$
4. $\exists O1”.bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O1”, y', nil) *$
   $(\forall * o \in O1”.o.bag \mapsto bag) * br \mapsto \_$
5. $\exists O1.bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O1', y', nil) *$
   $(\forall * o \in O1'.o.bag \mapsto bag) * br \mapsto \_$
8. $\exists O1.bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O1, y', nil) *$
   $(\forall * o \in O1.o.bag \mapsto bag) * br \mapsto \_$
9. $bag.list \mapsto x' * ls(x', nil)$
10. $bag.list \mapsto x' * ls(x', nil)$
11. $rlist \mapsto z' * ls(z', nil) * bag.list \mapsto x' * ls(x', nil)$
12. $rlist \mapsto z' * ls(z', nil)$
13. $\exists O”.rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto x' * revbag.observers \mapsto y' *$
    $ls(x', nil) * ls(O”, y', nil) * (\forall * o \in O”.o.bag \mapsto revbag)$
14. $\exists O'.rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto x' * revbag.observers \mapsto y' *$
    $ls(x', nil) * ls(O', y', nil) * (\forall * o \in O'.o.bag \mapsto revbag)$
15. $\exists O.rlist \mapsto z' * ls(z', nil) * revbag.list \mapsto x' * revbag.observers \mapsto y' *$
    $ls(x', nil) * ls(O, y', nil) * (\forall * o \in O.o.bag \mapsto revbag)$
16. $\exists O.ls(i, nil) * revbag.list \mapsto x' * revbag.observers \mapsto y' *$
    $ls(x', nil) * ls(O, y', nil) * (\forall * o \in O.o.bag \mapsto revbag)$
18. $revbag.list \mapsto x' * ls(x, nil)$
19. $revbag.list \mapsto x' * ls(x', nil)$

B Specification of the underlying classes in our example

B.1 Specification for the class IntegerDataBag

```
public class IntegerDataBag implements Subject {
      private ArrayList list = new ArrayList();
      private ArrayList observers = new ArrayList();

      public void add(Integer i);
      public Iterator iterator();
      public Integer remove(int index);
      public void addObserver(Observer o);
      public void removeObserver(Observer o);
      private void notifyObservers();
      public ArrayList reverseList();
      public Integer sum();
      public Integer mult();
      public void printBag();
}
```

$$\{\mathsf{emp}\}$$
$$\mathit{IntegerDataBag}()$$
$$\{\mathit{this.list} \mapsto x' * \mathit{this.observers} \mapsto y' * \mathit{ls}(x', \mathit{nil}) * \mathit{ls}(, y', \mathit{nil})\}$$

$$\{\exists O.\mathit{this.list} \mapsto x' * \mathit{this.observers} \mapsto y' * \mathit{ls}(x', \mathit{nil}) * \mathit{ls}(O, y', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$
$$\mathit{add}(i);$$
$$\{\exists O.\mathit{this.list} \mapsto x' * \mathit{this.observers} \mapsto y' * \mathit{ls}(x', \mathit{nil}) * \mathit{ls}(O, y', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$

$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$
$$\mathit{iterator}();$$
$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$

$$\{\exists O.\mathit{this.list} \mapsto x' * \mathit{this.observers} \mapsto y' * \mathit{ls}(x', \mathit{nil}) * \mathit{ls}(O, y', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$
$$\mathit{remove}(\mathit{index});$$
$$\{\exists O.\mathit{this.list} \mapsto x' * \mathit{this.observers} \mapsto y' * \mathit{ls}(x', \mathit{nil}) * \mathit{ls}(O, y', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$

$$\{\exists O.\mathit{this.observers} \mapsto x' * \mathit{ls}(O, x', \mathit{nil}) * \mathit{Observer}(o) * (\forall o \in O.o \mapsto \mathit{this})\}$$
$$\mathit{addObserver}(o);$$
$$\{\exists O.\mathit{this.observers} \mapsto x' * \mathit{ls}(o : O, x', \mathit{nil}) * (\forall o' \in o : O.o' \mapsto \mathit{this})\}$$

$$\{\exists O.\mathit{this.observers} \mapsto x' * \mathit{ls}(o : O, x', \mathit{nil}) * (\forall o' \in o : O.o' \mapsto \mathit{this})\}$$
$$\mathit{removeObserver}(o);$$
$$\{\exists O.\mathit{this.observers} \mapsto x' * \mathit{ls}(O, x', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$

$$\{\mathit{this.observers} \mapsto x' * \mathit{ls}(O, x', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$
$$\mathit{notifyObservers}();$$
$$\{\mathit{this.observers} \mapsto x' * \mathit{ls}(O, x', \mathit{nil}) * (\forall o \in O.o \mapsto \mathit{this})\}$$

$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$
$$\mathit{reverseList}();$$
$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil}) * \mathit{temp} \mapsto y' * \mathit{ls}(y', \mathit{nil}) / \mathsf{return}(\mathit{temp})\}$$

$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$
$$\mathit{sum}();$$
$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$

$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$
$$\mathit{mult}();$$
$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$

$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$
$$\mathit{printBag}();$$
$$\{\mathit{this.list} \mapsto x' * \mathit{ls}(x', \mathit{nil})\}$$

B.2 Specification for the class IntegerAdder

```
public class IntegerAdder implements Observer {
      private IntegerDataBag bag;

      public IntegerAdder(IntegerDataBag bag);
      public void update(Subject o);
}
```

$\{\exists O.bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall o \in O.o \mapsto bag)\}$
$IntegerAdder(bag);$
$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y' * ls(this : O, x', nil) * ls(y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$

$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$
$update(o)$
$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$

B.3 Specification of the class IntegerPrinter

```
public class IntegerPrinter implements Observer {
      private IntegerDataBag bag;

      public IntegerPrinter(IntegerDataBag bag);
      public void update(Subject o);
}
```

$\{\exists O.bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil) * (\forall o \in O.o \mapsto bag)\}$
$IntegerPrinter(bag)$
$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y * ls(x', nil) * ls(this : O, y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$

$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$
$update(o)$
$\{\exists O.this.bag \mapsto bag * bag.list \mapsto x' * bag.observers \mapsto y' * ls(x', nil) * ls(O, y', nil)*$
$(\forall o \in O.o \mapsto bag)\}$