Interprocedural Functional Shape Analysis using Local Heaps

Technical Report TAU-CS-26/04

Noam Rinetzky	Mooly Sagiv	Eran Yahav
Tel Aviv University	Tel Aviv University	IBM T.J. Watson
maon@tau.ac.il	*msagiv@tau.ac.il*	*eyahav@us.ibm.com*

Abstract

We present a framework for interprocedural shape analysis, which is context- and flow-sensitive with the ability to perform destructive pointer updates. Instances of the framework are precise enough to prove the absence of program errors such as null dereferences, memory leaks and verify conformance to API specifications.

Technically, our analysis computes procedure summaries as transformers from inputs to outputs while *ignoring parts of the heap not relevant to the procedure*. This makes the analysis modular in the heap and thus allows reusing the effect of a procedure at different call-sites and even between different contexts occurring at the same call-site.

A prototype of our framework was implemented and used to verify interesting properties of heap manipulating programs.

Contents

1 Introduction

Shape-analysis algorithms statically analyze a program to determine information about the heap-allocated data structures that the program manipulates. The algorithms are *conservative* (sound), i.e., the discovered information is true for every input. The analysis of large programs presents a major problem to existing shape analyzers. Handling the heap in a precise manner requires strong-pointer updates [3]. However, performing strong pointer updates requires flow-sensitive context-sensitive analysis and expensive heap abstractions which are doubly-exponential in the program size [20]. The presence of procedures escalates the problem because of interactions between the program stack and the heap [19] and because recursive calls introduce exponential factors in the analysis.

In [18], Rinetzky et. al. present a non-standard semantics for Java programs in which procedures operate on local-heaps reachable from actual parameters. In principle, abstracting this non-standard semantics may yield a new shape analysis algorithm that is significantly more efficient and scalable compared to existing algorithms. This algorithm may maintain the ability to conduct *precise* analysis of well-behaved programs, in which the number of *sharing patterns* between the local heap and the rest of the heap is small. (For programs with many such sharing patterns the resultant analysis can be imprecise.) In practice, however, it is not trivial to find an abstraction of this semantics that provides a sufficiently precise algorithm that scales relatively well.

This paper presents a detailed design and implementation of a framework for implementing such an algorithm for single threaded Java programs. We also provide an initial empirical evaluation of a new functional interprocedural shape analysis algorithm. Our empirical evaluation indicates that the cost and the precision of the analysis of recursive procedures that manipulate linked data structures, are comparable to the cost and precision of the analysis of iterative versions of the same procedures. The analysis is precise enough to prove properties such as the absence of null dereferences, preservation of data structure invariants such as list-ness and tree-ness, and conformance to API specifications with deep references to the heap and destructive updates. Moreover, the cost of the analysis of a program with procedures is smaller than the cost of the analysis of the same program with procedure bodies inlined.

Our interprocedural shape-analysis is a functional interprocedural analysis [1,5,6,8, 9,15,21]. It tabulates abstractions of memory configurations called shape graphs before and after procedure calls. However, shape graphs are represented in a non-standard way *ignoring parts of the heap not relevant to the procedure*. This reduces the complexity of the analysis because the analysis of procedures does not represent information on references and the heap from calling contexts. Indeed, this makes the analysis modular in the heap and thus allows reusing the effect of a procedure at different calling contexts. Finally, this reduces the asymptotic complexity of the interprocedural shape analysis. For programs without global variables, the worst case time complexity of the analysis is doubly exponential in the maximum number of local variables in a procedure, instead of being doubly exponential in the overall number of local variables [19].

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [11, 20]. Thus, it is parametric in the heap abstraction and in the concrete effects of program statements, allowing to experiment with different instances of interprocedural shape analyzers. For example, we can employ different abstractions for singly-, doubly-linked lists, and trees. Also, the combination of the theorems in [18] and [20] guarantees that every instance of our *interprocedural* framework is sound (see Section 5).

A subtle issue in our analysis is the treatment of sharing between the local heap and the rest of the heap. The problem is that the local heap can be accessed via access paths which bypass actual parameters. Therefore, objects in the heap reachable from an actual parameter are treated differently when they separate the "local heap" that can be accessed by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. We call these objects *cutpoints* [18]. We observe that in many programs the number of cutpoints is small, even in the presence of shared data structures.

Our framework for interprocedural shape analysis allows the specifier to define the expected cutpoints. The analysis becomes imprecise (and expensive) in programs in which several cutpoints are summarized together. Indeed, it is instructive to distinguish between two dimensions of heap abstractions: (i) The abstraction of the local-heap which discriminate between different kinds of aliases inside the part of the reachable part of the heap. For example, a node which is pointed-to by two or more selectors from the local-heap can be treated differently. (ii) The abstraction of the *sharing patterns* between the local-heap and the rest of the heap. For example, in our experimental results we conservatively represent all cutpoints of the same class using a single summary node which leads to a loss of precision when more than one cutpoint is created.

1.1 Main Results

The technical contributions of this paper can be summarized as follows:

- We present a framework for interprocedural shape analysis using local heaps. This requires filling the algorithmic details left open in [18], which mainly addresses semantic issues and generalizing the abstraction to allow parametric heap abstractions. In particular, we define the effect of call and return statements using first order formulas with transitive closure and show how to compute the effect of program statements.
- We implemented a general algorithm to handle arbitrary single-threaded Java programs.
- We empirically evaluated the analysis by experimenting with several small heap-intensive programs.

1.2 Outline

The rest of the paper is organized as follows. Section 2 provides an overview of the main ideas used in the paper. Section 3 describes our interprocedural analysis algorithm adapted from [15]. Section 4 defines an instrumented concrete semantics that serves as the basis for our interprocedural analysis. Section 5 presents a conservative abstract semantics abstracting the instrumented concrete semantics of Section 4, thus providing algorithms to compute the intra- and interprocedural transfer functions. Our prototype implementation and the experimental results are described in Section 6. Section 7 describes related work, and Section 8 concludes. Appendix A provides a formal specification of the operational semantics. Appendix B.1 shows that the concrete semantics used in this paper is equivalent to an instrumented version of the semantics presented in [18]. This allows to use their results to prove the soundness of our algorithm.

2 Overview

This section provides an overview of our framework for interprocedural functional shape analysis using local heaps. The presentation is at a semi-technical level; a more detailed treatment of this material, as well as several elaborations on the ideas covered here, is presented in the later sections of the paper.

2.1 Running Example Program

Figure 1 shows a simple Java client that splices two unshared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. Our interprocedural shape analysis verifies that the returned list is acyclic and not heap-shared; that the first parameter is aliased with the returned reference; and that the second parameter points to the second element in the returned list. This example serves as a running example in this paper.

2.2 Handling Simple Procedure Calls

Our algorithm tabulates abstractions of memory configurations called shape graphs, before and after procedure calls. Figure 2 shows the concrete memory configurations and corresponding shape graphs that occur at the first call `t = splice(x,y);` from `main`.

2.2.1 Concrete States

Concrete states are drawn as directed graphs. Heap-allocated objects (in our case, list-elements) are depicted as boxes with data values inside boxes. The values of pointer fields are depicted as edges labeled with the field name. The values of pointer (reference) variables are drawn as directed edges to nodes. Our concrete semantics does not pass the linked list pointed-to by `z` to the procedure because it is not reachable from the actual parameters `x` or `y`. As we shall see, this increases the scalability of the analysis. At this point, please ignore $\widehat{p}$, $\widehat{q}$, $\widehat{n}$, and the circle nodes. We explain their role in Section 2.3.

2.2.2 Abstract States

Abstract states are shape graphs. Every node in the concrete state is represented by a node in a shape graph. *Summary* nodes, drawn as double-boxes are nodes which may represent more than one allocated object. Again, for now ignore $\widehat{p}$, $\widehat{q}$, $\widehat{n}$, and the circle nodes.

```
public class List{
    List n;
    int data;
    public List(int d){
        this.data = d; this.n = null;
    }
    static public List crt3(int k) {
        List t1 = new List(k);
        List t2 = new List(k+1);
        List t3 = new List(k+2);
        t1.n = t2; t2.n = t3;
        return t1;
    }
    public static List splice(List p, List q) {
        if (p == null) return q;
        List r = p.n;
        p.n = null;
        p.n = splice(q, r);
        return p;
    }
    public static void main(String[] argv) {
        List x = crt3(1), y = crt3(4), z = crt3(7);
        List t = splice(x, y);
        List s = splice(t, z);
    }
}
```

Figure 1: The running example.

Nodes are annotated by properties of represented allocated objects. Moreover, concrete objects with different properties are represented by different nodes. The property r_z holds for objects that are reachable from a variable `z` via a sequence of pointer fields. Thus, in the Figure 2(e) (at the call to `splice`), the second and the third concrete elements of the list pointed-to by `x` (with values 2 and 3) are represented by the summary node annotated by r_x in the abstracted shape graph. We can see that the shape graphs abstract the actual `data` values and the lengths of the lists.

Solid n-edges between nodes in the shape graphs represent `n` pointer fields. Absence of solid edges between nodes represents the absence of pointer fields. Finally, dotted edges represent loss of information in the abstraction, i.e., `n` pointer fields which may or may not exist. Thus, the shape graph at Figure 2(e) represents three disjoint lists of length 2 or more with heads, `x`, `y`, and `z`, respectively.

Figure 2(f) represents the relevant part of the heap at the entry to the callee, two disjoint lists of length 2 or more pointed to by `p` and by `q` respectively.

Figure 2(g) represents the store at the exit — `q` points to the second element of the list pointed to by `p`. Note how the tail of the list is reachable from both `p` and `q`. Finally, Figure 2(h) represents the store upon return.

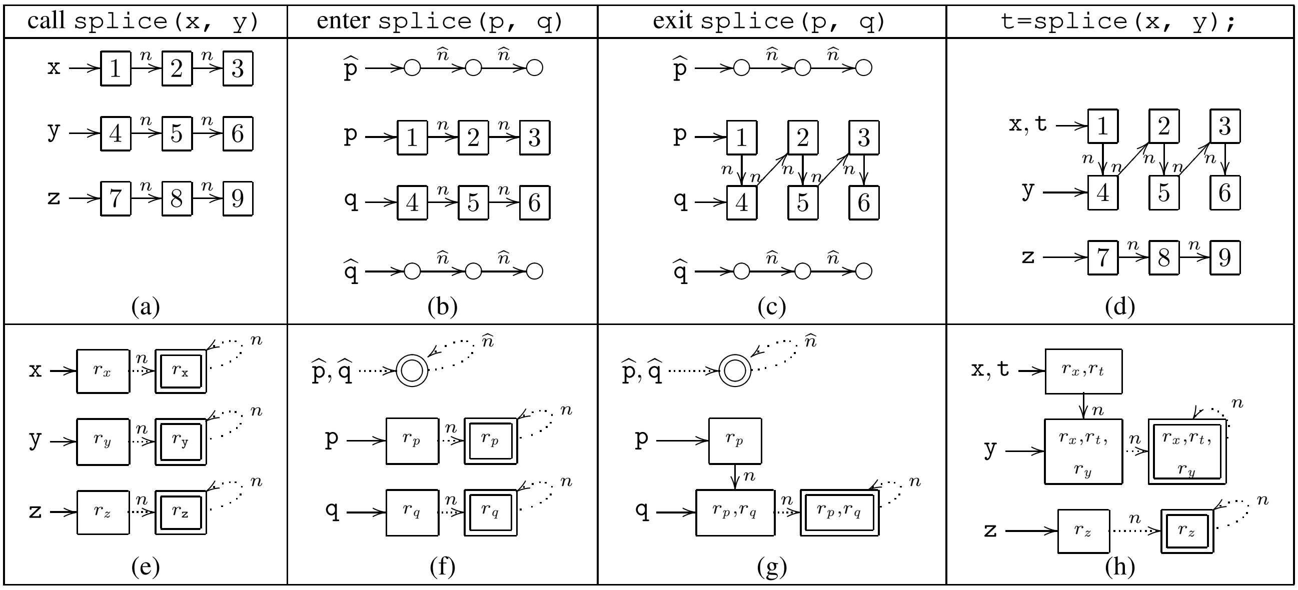

Figure 2: Concrete (first row) and abstract (second row) states in the analysis of the invocation `t = splice(x, y);` in the running example.

2.2.3 Discussion

Because the number of properties such as r_x is finite for a given program, so is the number of nodes and shape graphs. Therefore, by simple tabulation of input/output shape graphs, we obtain a context- and flow-sensitive analysis which enforces matching calls and returns even in the presence of recursion. Moreover, the fact that on a procedure call, we only represent the local heap reachable from actual parameters allows us to abstract facts at the caller which are irrelevant to the callee. For example, Figure 2(f)–(g) also represents all recursive calls in which `q` points to the second element of the list pointed to by `p`. This dramatically improves the scalability of handling procedures because it does not discriminate between contexts with different irrelevant parts of the heap. In fact, the cost of handling recursive `splice` is propositional to the cost of handling the iterative version and in both cases we can prove that the result is indeed an unshared acyclic list (see Section 6).

2.3 Handling Procedure Calls with Cutpoints

We are now ready to explain the role of $\widehat{p}$, $\widehat{q}$, and the circle nodes. Figure 3 shows the concrete memory configurations and corresponding shape graphs that occur at the second call `s = splice(t,z);` from `main`. This call differs from the first call because `y` points-to the second element in the list which was passed to the procedure. Therefore, destructive updates in the procedure may make nodes (un)reachable from `y` and thus change the r_y property. In other words, `y` creates stack sharing into the local heap. The challenge is finding a way to update properties such as reachable-from-`y`

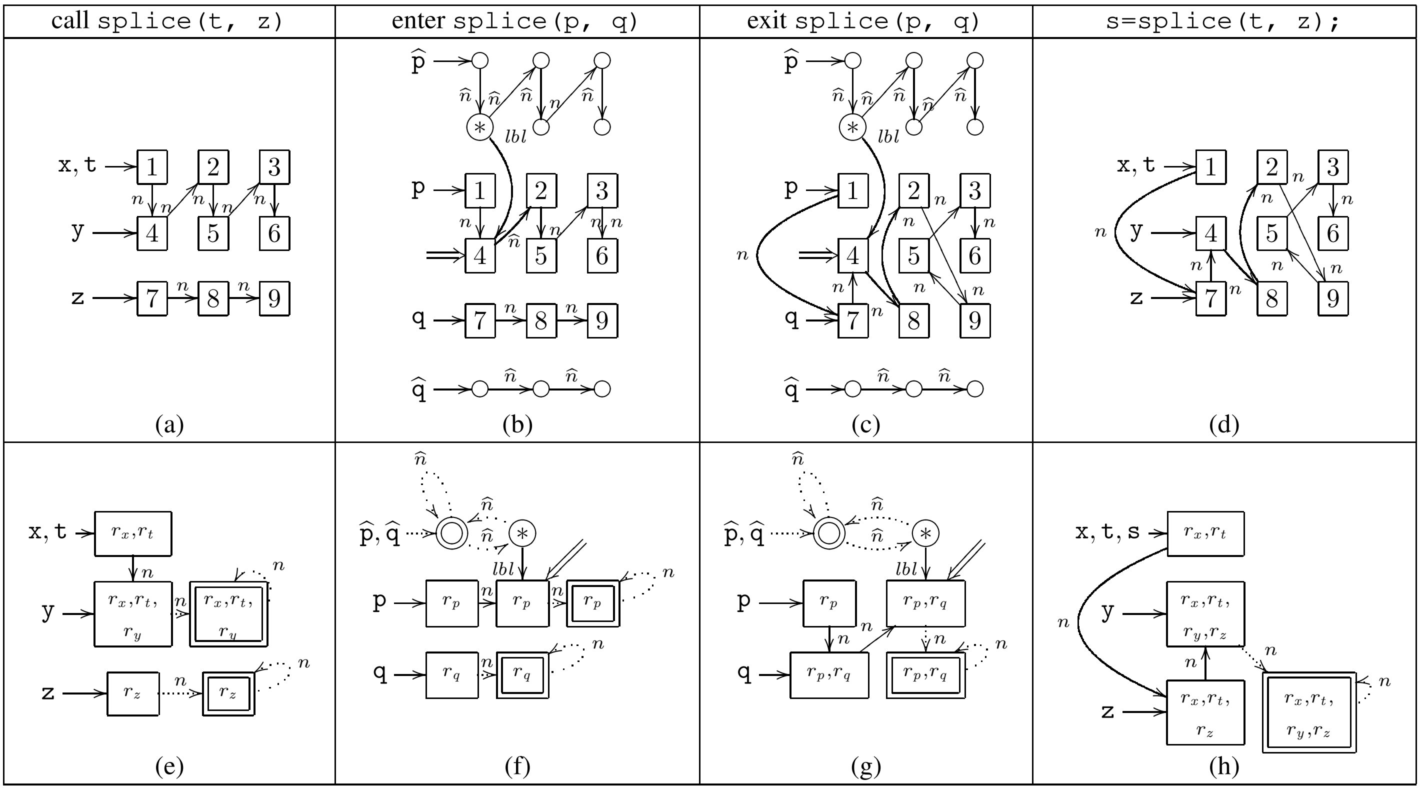

Figure 3: Concrete (first row) and abstract (second row) states in the analysis of the invocation `s = splice(t, z);` in the running example.

without explicitly representing `y`. The ability to do so is crucial for reusing the analysis of the procedure body across different procedure invocations.

Our solution uniformly treats stack sharing and sharing of fields from the rest of the heap by recording *sharing patterns* into the local heap. The main idea is to give special treatment to *cutpoints* [18]. Cutpoints are objects which separate the local heap of the callee from rest of the heap but are not pointed-to by a parameter. Our analysis labels cutpoints with the access paths that points to them and start with formal parameters at procedure-entry. This provides naming scheme for the cutpoints labels (the sharing pattern) which is independent of the irrelevant context. For example, we label the list-element pointed-to by `y` with `p.n`

We use circle nodes, which we refer to as *labels*, to represent the values of access paths emanating from formal parameters at procedure entry. Every object o in the local heap passed to the callee has a corresponding "shadow" circle node which freezes the values of o's pointer-fields. The value of pointer variable `p` at procedure-entry is denoted by $\widehat{p}$. The value of pointer field `f` at procedure-entry is denoted by $\widehat{f}$. The "shadow" of a cutpoint is connected to the cutpoint object with an *lbl*-labeled edge. For readability, we depict cutpoint objects with an incoming double-line arrow. Note that our labeling scheme records the sharing pattern in a way which ignores the *contents* of irrelevant context. This ensures that the analysis results are applicable for every calling context which generates the same sharing pattern.

2.3.1 Concrete States

Consider the concrete state when `splice(p, q)` is entered. The second node in `p`'s list is a cutpoint. Its label is the second circle node on $\widehat{p}$'s list (the asterisk marks the fact that this label corresponds to a cutpoint). The *lbl*-edge connects the label to the cutpoint node. This signifies that the access path `p.n` points-to a cutpoint object. It is used to relate the access-path's value between the call and the return.

Now the code at the procedure body is executed without the need to represent `y` or any other irrelevant part of the global heap but instead with labels marking the cutpoint objects.

Cutpoints labels are frozen between procedure entry and procedure exit. Moreover, without loss of generality we assume that formal parameters refer to the same objects before and after the call. Therefore, the semantics can reconnect `y` to the second node from $\widehat{p}$ when the procedure exits. Thus, the mutation of the heap is transmitted to the caller while correctly updating `y`.

2.3.2 Abstract States

In the interprocedural shape analysis we tabulate shape graphs with special summary labels potentially representing multiple cutpoints. In the example, the circle summary label represents the values of all access paths but `p.n`, which is marked by an asterisk.

In this example, because the procedure only includes one cutpoint object, the latter is not summarized. Therefore, upon exit, the abstract transformer can precisely update reachability from `y`.

In those cases where cutpoint objects are summarized, our analysis is sound but overly conservative. Moreover, this may degrade performance, because the analysis can create many shape graphs at the return-site corresponding to different potential ways in which a cutpoint is considered.

2.4 A Global View of Local Heaps

Our analysis is sound, i.e., upon termination, the shape graphs describe all potential memory configurations. Moreover, as discussed in Section 5.2, soundness is guaranteed by relying on prior theorems in [18, 20]. The key reason for soundness is that the abstract interpretation of every statement takes into account all potential memory configurations represented by the input shape graphs.

Figure 4 illustrates that for abstract (local-heap) shape graph at Figure 3(g). which results from the second call to `s = splice(t, z);`. We give two examples of potential global stores represented by this shape graph and two examples which are not represented by this shape graph. Note that here, we actually draw the global heap as in previous works in shape analysis (i.e., we draw all the allocated objects and we do *not*

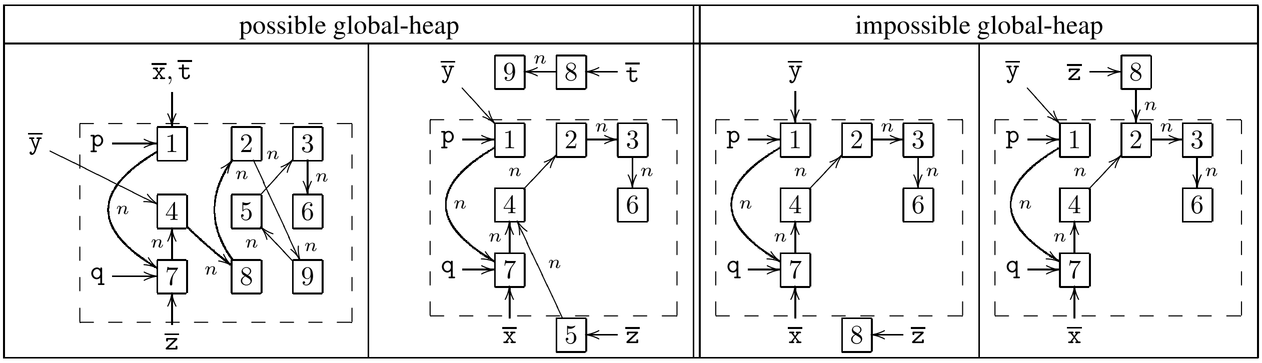

Figure 4: Potential concrete *global-heaps* represented by the abstract (local-heap) shape graph shown in Figure 3(g).

draw labels). We draw a dashed frame around the local heap. Also, we use the notation $\overline{\texttt{x}}$ to denote a reference variable `x` of a pending call.

The first store is the one which actually occurs at the program. The second store is possible because `z.n` points-to the cutpoint object pointed-to by `p.n` at the procedure entry and `p.n.n` at the procedure exit. Notice that here the cutpoint is created via heap sharing and not stack sharing. Also notice that such stores are not propagated to the return-site because the shape graph at the entry forces `t` and `p` to be aliased.

The two right stores represent impossible situations excluded by the cutpoint representation. In the first store, there are no cutpoint at all. This situation is a similar to the one at the exit from the call `t = splice(x, y);` from `main`. In the second store, there is one cutpoint object but it is not the object pointed to by `p.n.n` at the procedure exit.

3 Interprocedural Analysis

In this section, we describe the iterative interprocedural local-heap shape-analysis algorithm. The algorithm computes procedure summaries by tabulating input shape-graphs to output shape-graphs. The tabulation is restricted to shape-graphs that *occur in the analyzed program*. The abstract domain is the powerset of *shape-graphs* (2^{SG}) with set-union as the *join* operator. The abstract-transformers are always applied point-wise, thus they distribute over the join operator (e.g., see [13]).

The algorithm is a variant of the IFDS-framework [15] adapted to work with local-heaps. The main difference between our framework and [15] is in the way return statements are handled: In [15], the dataflow facts that reach a return-site come either from the call-site (for information pertaining global variables) or from the exit-site (for information pertaining local variables). In our case, the information about the heap is obtained by *combining* pair-wise the shape-graphs at the call-site with the shape-graphs at the exit-site: the information about the values of local variables and fields of objects that point to the part of the heap which was *not* passed to the callee is passed as-is from the call-site. The information about the values of fields of objects in the part of the heap which was passed to the callee is taken as-is from the exit-site. The information about the value of the caller's local variables and the values of fields of objects that were not passed to the callee, but point to objects that are passed to the callee, are computed by the *combine* operation using the *cutpoints*.

For simplicity, in this section we assume that all procedures in the program are *static*. In Section 6, we describe the way dynamic dispatch is handled.

3.1 Program Model

We represent a program P in a standard manner by the set of *control-flow-graphs* of its procedures (with a distinguished `main` procedure), connected by a set of interprocedural call/return edges. The control-flow-graph CFG_p of a procedure p, is comprised of a set of nodes N_p, representing program locations, and a set of intraprocedural edges $E_p \subseteq N_p \times N_p$ labeled with program statements. We assume that every CFG_p has a single entry-node, $entry(p)$, and a single exit-node, $exit(p)$.

We partition the set $N^\star$ of all CFG nodes in the program into five subsets: $Entry^\star$, $Exit^\star$, $Call^\star$, $Ret^\star$, and $Intra^\star$, corresponding to the sets of all entry-nodes, exit-nodes, call-sites, return-sites, and all other nodes, respectively.

The procedural control-flow graphs are connected by a set of interprocedural edges $E_{inter} \subseteq Call^\star \times Entry^\star \cup Exit^\star \times Ret^\star$. We denote the set of all program edges by $E^\star = \bigcup_{p \in pgm} E_p \cup E_{inter}$.

For simplicity, we guarantee that return-sites are not call-sites or exit-sites, by augmenting each return-site with a single `nop` operation. In addition, we assume that a procedure returns a value by assigning it to a designated variable, **`ret`**.

In the sequel we denote the set of outgoing edges for a node $n \in N^\star$ by $out(n)$, and the statement that labels an edge $\langle n, n' \rangle \in E^\star$ by $stmt(\langle n, n' \rangle)$. We also use $callee(n_{call})$ and $return(n_{call})$ to denote the target of the call at n_{call} and the return-site of n_{call}, respectively. For an entry-node $n \in Entry^\star$, we denote the matching exit-node by $exit(n)$.

3.2 Tabulation Algorithm

The tabulation algorithm propagates *path-edges*. A *path-edge* $\langle sg_{entry}, sg \rangle$ is propagated to a control flow graph node $n \in N_p$ iff there exists an interprocedural-valid-path [21] from $entry(p)$ to n such that applying the composed effect of all abstract transformers associated with statements along the path to sg_{entry} results in sg [15].

In this section we describe the algorithm using the following operations as "black boxes" (the contents of which is described in detail in Section 4 and Section 5):

- *apply* : $Stmt \times SG \to 2^{SG}$ applies the abstract transformer associated with a given *intraprocedural* statement to a given shape-graph and returns the resulting set of shape-graphs.
- *extract* : $Stmt \times SG \to 2^{SG}$ applies the abstract transformer associated with the given call-statement to the given shape graph. Thus, computing the shape-graph that represents the local-heap which is passed to the callee.

proc tabulate($Program\ P$, $SG\ sg_0$)
$\quad worklist = \{\langle entry(main) : \langle sg_0, sg_0\rangle\rangle\}$
$\quad$ **while** ($worklist \neq \emptyset$)
$\quad\quad$ remove an event $\langle n : \langle sg_{entry}, sg\rangle\rangle$ from $worklist$
$\quad\quad$ **if** $n \in Entry^\star \cup Ret^\star \cup Intra^\star$ **then**
$\quad\quad\quad$ **foreach** $\langle n, n'\rangle \in out(n)$
$\quad\quad\quad\quad$ **foreach** $sg' \in$ ***apply***$(stmt(\langle n, n'\rangle), sg)$
$\quad\quad\quad\quad\quad$ **if** $\langle sg_{entry}, sg'\rangle \notin PathSet(n')$ **then**
$\quad\quad\quad\quad\quad\quad$ propagate$(n', \langle sg_{entry}, sg'\rangle)$
$\quad\quad$ **else if** $n \in Call^\star$
$\quad\quad\quad n^{entry}_{callee}$ = *callee*(n)
$\quad\quad\quad$ **foreach** $sg' \in$ ***extract***$(stmt(\langle n, n^{entry}_{callee}\rangle), sg)$
$\quad\quad\quad\quad$ *add* $\langle n, \langle sg_{entry}, sg\rangle\rangle$ *to* $CTXs(n^{entry}_{callee})\ sg'$
$\quad\quad\quad\quad$ **if** $\langle sg', sg'\rangle \notin PathSet(n^{entry}_{callee})$ **then**
$\quad\quad\quad\quad\quad$ propagate$(n^{entry}_{callee}, \langle sg', sg'\rangle)$
$\quad\quad\quad\quad$ **else**
$\quad\quad\quad\quad\quad n_{exit} = exit(n^{entry}_{callee})$
$\quad\quad\quad\quad\quad$ **foreach** $sg_{exit} \in Summary(n^{entry}_{callee})\ sg'$
$\quad\quad\quad\quad\quad\quad$ addToRet$(n_{exit}, sg_{exit}, return(n), \langle sg_{entry}, sg\rangle)$
$\quad\quad$ **else** $//n \in Exit^\star$
$\quad\quad\quad$ **foreach**$\langle n_{call}, \langle sg_e, sg_c\rangle\rangle \in CTXs(n^{entry}_{callee})\ sg_{entry}$
$\quad\quad\quad\quad$ addToRet(n, sg, $return(n_{call})$, $\langle sg_{entry}, sg_{call}\rangle$)

proc addToRet($N_{callee}\, n_{exit}, SG\ \ sg_{\mathtt{x}},\ N_{caller}\, n_{ret}, SG \times SG\ \ \langle sg_e, sg_c\rangle$)
$\quad$ **foreach** $sg' \in$ ***combine***$(stmt(\langle n_{exit}, n_{ret}\rangle), \langle sg_c, sg_x\rangle,)$
$\quad\quad$ **if** $(\langle sg_e, sg'\rangle \notin PathSet(n_{ret}))$ **then**
$\quad\quad\quad$ propagate$(n_{ret}, \langle sg_{\mathtt{e}}, sg'\rangle)$

Figure 5: The tabulation algorithm.

- *combine* : $Stmt \times SG \times SG \rightarrow 2^{SG}$ computes the shape-graph representing the local-heap of the caller at the return-site by applying the associated abstract transformer when control returns to the caller. This operation gets two shape graphs as arguments, one from the call-site and the other from the callee exit-site.

The algorithm maintains the following data structures:

- The set $PathSet(n)$ contains all path edges propagated to node n. These sets are initialized to $\emptyset$. Note that $PathSet(exit(p))$ contains the (already-computed) summarized effect of the procedure. Thus, we define $Summary : Entry^\star \rightarrow SG \rightarrow 2^{SG}$ which maintains the procedure summary as
$Summary(entry(p))\ sg_{entry} = \{sg_{exit} : \langle sg_{entry}, sg_{exit}\rangle \in PathSet(exit(p))\}$.
- The multi-map $CTXs$: $Entry^\star \rightarrow (SG \times SG) \rightarrow 2^{Call^\star \times SG \times SG}$ associates

every procedure p, identified by its entry-site, $entry(p)$, with its calling context. The calling-context is a map from every 0-length path-edge $\langle sg_{entry}, sg_{entry}\rangle \in PathSet(entry(p))$ which was propagated to p's entry to a set of pairs of a call-site $n^{caller}_{call} \in Call^{\star}$ and a path-edge $\langle sg^{caller}_{entry}, sg^{caller}_{call}\rangle \in PathSet(n^{caller}_{call})$ such that the analysis of the invocation of p at call-site n^{caller}_{call} extracted the shape-graph sg_{entry} out of sg^{caller}_{call}. This map is initialized to associate entry-nodes with empty maps.

The iterative algorithm (procedure `tabulate`) is defined in Figure 5. The *worklist* is initialized to contain a 0-length path edge from a shape-graph representing the memory at the entry to the program to the same shape graph (in our experiments this is always a shape graph representing an empty memory). It then iterates until the *worklist* is exhausted. In every iteration, the algorithm extracts one *event* out of the *worklist*. An *event* is comprised of a CFG node n and a path edge $\langle sg_{entry}, sg\rangle$. The algorithm performs one of the following operations depending on the role of n:

- If n represents a procedure entry, a return-site or a program location of an intra procedural statement, the algorithm *applies* the abstract transformer associated with each edge emanating from n, propagating an (extended) path edge, if necessary.
- If n represents a call-site to procedure p, the algorithm *extract*s sg', the shape-graph representing the callee-local heap from the target of the path-edge (sg). It then adds the call-site n and path-edge $\langle sg_{entry}, sg\rangle$ to the calling contexts of $CTXs(n^{entry}_{callee})\ sg'$. This operation "registers" $\langle n, \langle sg_{entry}, sg\rangle\rangle$ as a calling-context of p, which means that whenever a *new* path edge whose source is sg' is propagated to $exit(p)$, the algorithm propagates an appropriate shape-graph to the return-site $return(n)$. If the path edge $\langle sg', sg'\rangle$ has not been propagated to $entry(p)$, the algorithm propagates it. Otherwise, the algorithm propagates the known summary effect of p on sg' to the return-site using `addToRet` (see next case).
- If n represents the exit-site of procedure p, the algorithm updates the return-site of every calling-context which is registered for sg_{entry} (i.e., in $CTXs(entry(p))\ sg_{entry}$) using `addToRet`. The function `addToRet` *combine*s the shape-graph at the exit-site of the callee with the shape which is the target of the path-edge at the call-site.

The algorithm also uses the operation `propagate`$(n, \langle sg_{entry}, sg\rangle)$ that adds the edge $\langle sg_{entry}, sg\rangle$ to $PathSet(n)$, the set of path-edges at n; and inserts the event $\langle n : \langle sg_{entry}, sg\rangle\rangle$ to the worklist.

4 Concrete Semantics

In this section, we present an instrumented concrete semantics that serves as the basis for our abstraction. The instrumented semantics records object-labels and cutpoint-labels as introduced in Section 2.3. Technically, we use first-order logical structures to represent local heaps, and show how the operations described as black-boxes in Section 3.2 are realized as operations on first-order logical structures. For simplicity, we do not provide full formal details of these operations here. The formal details are

Predicate	Intended Meaning
$T(v)$	v is an object of type T or a label of an object of type T
$x(v)$	reference variable `x` points to the object v
$f(v_1, v_2)$	the `f`-field of object v_1 points to object v_2
$eq(v_1, v_2)$	v_1 and v_2 are the same object or the same label
$isObj(v)$	v is an heap-allocated object
$isLb_O(v)$	v is an object-label
$isLb_{CP}(v)$	the object-label v is also a cutpoint-label
$\widehat{x}(v)$	v labels the object which was pointed-to by the formal parameter `x` when the *current* procedure was invoked
$\widehat{f}(v_1, v_2)$	v_2 labels the object which was the `f`-successor of the object labeled by v_1 when the *current* procedure was invoked
$lbl(v_1, v_2)$	the list element v_2 is labeled by cutpoint-label v_1

Table 1: Predicates used in the instrumented semantics.

given in Appendix A.

4.1 Representing the Heap using 2-valued Logical Structures

We represent the (instrumented) state of a program using a first-order logical structure in which each individual corresponds to a heap-allocated object or to an object label. Predicates of the structure correspond to properties of heap-allocated objects, or to properties of object-labels.

A 2-valued logical structure over a set of predicates $\mathcal{P}$ is a pair $S = \langle U^S, \iota^S \rangle$ where:

- U^S is the universe of the 2-valued structure. Each individual in U^S represents a heap-allocated object or an object-label.
- ι^S is the partial interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) : {U^S}^k \to \{0, 1\}$ or is undefined.

The set of *2-valued* logical structures is denoted by $\mathit{2Struct}$.

In this paper, we allow each procedure to use a different set of predicates. To enable that, we assume that all structures are using the same global set of predicates $\mathcal{P}$, and allow each structure S to define the meaning of a different subset of predicates $\mathcal{P}^S \subseteq \mathcal{P}$. That is, we let ι^S be a partial function only defined for a subset of the predicates in $\mathcal{P}$. Given a single procedure, the set of predicates used in all structures representing its local state remains the same.

Table 1 shows the predicates used to model state of the instrumented concrete semantics in this paper. Predicates shown above the double horizontal line, are used to represent the core semantics of the program. Predicates below the double horizontal line record additional information of the instrumented semantics. A unary predicate $T(v)$ holds for heap-allocated objects of type T. A unary predicate $x(v)$ holds for

Predicate	Intended Meaning	Defining Formula
$r_x(v)$	v_2 is reachable from variable `x`	$isObj(v) \wedge \exists v_x\colon isObj(v_x) \wedge x(v_x) \wedge F^*(v_x, v)$
$r_{obj}(v_1, v_2)$	v_2 is reachable from object v_2 by following some field path	$isObj(v_1) \wedge isObj(v_2) \wedge F^*(v_1, v_2)$
$ils(v)$	v is *locally* shared. i.e., v is pointed-to by a field of more than one object in the *local-heap*	$isObj(v) \wedge \exists v_1, v_2\colon \neg eq(v_1, v_2) \wedge isObj(v_1) \wedge isObj(v_2) \wedge F(v_1, v) \wedge F(v_2, v)$
$ils_f(v)$	v is *locally* `f`-shared. i.e., is v pointed-to by the `f`-field of more than one object in the *local-heap*	$isObj(v) \wedge \exists v_1, v_2\colon \neg eq(v_1, v_2) \wedge isObj(v_1) \wedge isObj(v_2) \wedge f(v_1, v) \wedge f(v_2, v)$
$c(v)$	v resides on a directed cycle of fields	$(TC\ w_1, w_2 : F(w_1, w_2))(v, v)$
$c_f(v)$	v resides on a directed cycle of `f`-fields	$(TC\ w_1, w_2 : f(w_1, w_2))(v, v)$
$cp(v)$	v is a cutpoint	$\exists v_l\colon isLb_{CP}(v_l) \wedge lbl(v_l, v)$

Table 2: The instrumentation predicates used in this paper.

an object that is pointed-to by the reference variable `x`. A binary predicate $f(v_1, v_2)$ holds when the `f` field of v_1 points-to v_2. The predicates $isObj(v)$, $isLb_O(v)$, and $isLb_{CP}(v)$ distinguish individuals that represent heap-allocated objects, object-labels, and cutpoint-labels, respectively. A predicate $\widehat{x}(v)$ records the object that was pointed-to by a formal parameter `x` when the current procedure was invoked. Similarly, a predicate $\widehat{f}(v_1, v_2)$ records the value of an `f`-field when the current procedure was invoked. Finally, the predicate $lbl(v_1, v_2)$ records whether a list element v_2 is labeled by cutpoint-label v_1. The unary predicate $T(v)$ also hold for labels of heap-allocated objects of type T.

Instrumentation Predicates Instrumentation predicates record derived properties of individuals, and are defined using a logical formula over core predicates. They are used to refine the abstract semantics (see Section 5.1). Table 2 lists the instrumentation predicates used in this paper. We use the following shorthand notations. $F(v_1, v_2)$ is a shorthand for $\bigvee_{f \in FieldId^*_P} f(v_1, v_2)$ and $\widehat{F}(v_1, v_2)$ is a shorthand for $\bigvee_{f \in FieldId^*_P} \widehat{f}(v_1, v_2)$. In addition, for a formula φ with two free variables, the notation $\varphi^*(v_1, v_2)$ is a shorthand for the reflexive transitive closure of φ, i.e., $\varphi^*(v_1, v_2) \stackrel{\text{def}}{=} eq(v_1, v_2) \vee (TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2)$.

2-valued logical structures are depicted as directed graphs. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. We draw a node u that represents an object (i.e., $\iota^S(isObj)(u) = 1$) as a box and a node that represents a label (i.e., $\iota^S(isLabel)(u) = 1$) as a circle. We draw cutpoint objects with an incoming double-line arrow. Cutpoint-labels are marked with an asterisk. We depict the value of a pointer variable `x` by drawing an edge from `x` to the node that represent the object that `x` points-to. For a pointer parameter `q`, we also draw an edge from $\widehat{q}$ to the *label* of the object that `q` points-to. For all other unary predicates p, we draw p inside a node u when $\iota^S(p)(u) = 1$; conversely, when $\iota^S(p)(u) = 0$ we do not draw p in u.

Example 4.1 The *2-valued* logical structure that represents the local heap when `s=splice(t,z)` is invoked is depicted in Figure 3(a).

4.1.1 Admissible Memory States

Not all *2-valued* logical structures represent local-heaps that are compatible with the semantics of Java. For example, in Java each pointer variable points to at most one heap-allocated element. To exclude states that cannot arise in any program, we now define the notion of *admissible structures*. This notion is similar to the notion of *admissible states* in [18] and to the notion of structures that are *compatible with hygiene conditions* in [20]. We note that our semantics preserves states admissibility.

Definition 4.2 (Admissible 2-Valued Logical Structures) *A* 2-valued *logical structure $S = \langle U, \iota \rangle$ representing a local-heap for a function p in program P at a given point in an execution is* ***admissible*** *iff*

(i) The interpretation ι is defined for all the predicates in Table 1 except for the predicates representing the (frozen) value of variables. For the latter, $\iota(x)$ is defined only for $x \in V_p$ and $\iota(\widehat{x})$ is defined only for $x \in F_p$.

(ii) Every node $u \in U$ represents either an object *or an* object-label*, i.e., $S \models \forall v : isObj(v) \iff \neg isLb_O(v)$. Furthermore,* cutpoint-labels *must be* object-labels*, i.e., $S \models \forall v : isLb_{CP}(v) \implies isLb_O(v)$.*

(iii) Every node $u \in U$ has exactly one type, i.e., $S \models \forall v : \bigvee_{T \in TypeId^{\star}_P} T(v)$ and $S \models \forall v : \bigwedge_{T_1, T_2 \in TypeId^{\star}_P} T_1(v) \implies \neg T_2(v)$.

(iv) A (frozen) variable points-to at most one node, i.e., $S \models \forall v_1, v_2 : \bigvee_{x \in V_p} x(v_1) \wedge x(v_2) \implies eq(v_1, v_2)$ and $S \models \forall v_1, v_2 : \bigvee_{x \in F_p} \widehat{x}(v_1) \wedge \widehat{x}(v_2) \implies eq(v_1, v_2)$. Furthermore, variables only point to objects while frozen *variables only point to* object-labels*, i.e., $S \models \forall v : \bigvee_{x \in V_p} x(v) \implies isObj(v)$ and $S \models \forall v : \bigvee_{x \in F_p} \widehat{x}(v) \implies isLb_O(v)$.*

(v) A field is a partial function, i.e., $S \models \forall v, v_1, v_2 : \bigvee_{f \in FieldId^{\star}_P} f(v, v_1) \wedge f(v, v_2) \implies eq(v_1, v_2)$. Furthermore, a field maps objects to objects and objet-labels to object-labels, i.e., $S \models \forall v_1, v_2 : \bigvee_{f \in FieldId^{\star}_P} f(v_1, v_2) \implies (isObj(v_1) \iff isObj(v_2))$.

(vi) An lbl is a function that maps cutpoint-labels to objects, i.e., $S \models \forall v_1, v_2 : lbl(v_1, v_2) \implies isLb_{CP}(v_1) \wedge isObj(v_2)$. Furthermore, lbl is an injective function, i.e., $S \models \forall v, v_1, v_2 : lbl(v, v_1) \wedge lbl(v, v_2) \implies eq(v_1, v_2)$ and $lbl|_{\{u \in U : \iota(isLb_{CP})(u) = 1\}}$ is a surjective function, i.e., $S \models \forall v : isLb_{CP}(v) \implies \exists v_1 : lbl(v, v_1)$.

(vii) Every object label is reachable from at least one frozen variable, i.e., $S \models \forall v : isLb_O(v) \implies \bigvee_{x \in X} \exists v_1 : \widehat{x}(v_1) \wedge \widehat{F}^*(v_1, v)$.

4.2 Operational Semantics

We model program statements by *actions* that specify how a statement transforms an incoming logical structure into an outgoing logical structure. This is done primarily by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [20].

Handling dynamic object allocation requires more careful attention, as it changes not only the interpretation mapping, but also the underlying universe. We handle dynamic allocation by adapting the solution of [20].

We now describe the operational semantics for interprocedural statements, in particular, we show how the operations *extract* and *combine* are realized in terms of operations on first-order logical structures.

Procedure Call When a procedure p is called, our algorithm uses the *extract* operation to extract a relevant labeled local-heap from the heap provided at the call-site. To realize this operation using logical structures, our semantics takes the following steps: (i) creates new object labels for all relevant objects (objects reachable from actual parameters of p). This is done by using the function $\mathbf{clone}$ defined in Figure 7 to clone all objects reachable from parameters of p. (ii) adjusts the values of label-predicates of the form $\widehat{x}(v)$ and $\widehat{f}(v_1, v_2)$. (iii) adjusts the values of cutpoint-labels predicate. (iv) remove all irrelevant objects by using the function $\mathbf{remove}$ defined in Figure 7.

> **Example 4.3** Consider the concrete states shown at the first row of Figure 3. These structures represent the caller's and callee's local-heaps at various stages of the call to `s = splice(t,z)`. Structure (a) represents the caller's local-heap just before the call. Structure (b) represents the extracted local heap upon entrance to the callee.

Return Returning from a call to procedure p, our algorithm uses the *combine* operation to combine the structure representing the caller memory-state at the call-site and the structure representing the memory-state of the callee at the exit-site. To realize this operation using logical structures, our semantics takes the following steps: (i) it combines the structure representing the local-heap at the exit from p with the structure at the call site. This is done by using the function $\mathbf{combine}$ defined in Figure 7. (ii) uses cutpoints to merge the local-heap back into the global heap by finding matching individuals. Technically, this is achieved by using extended transitive closure (transitive closure of pairs) to traverse matching paths in the caller's heap and the callee's local heap. (iii) removes all labels of the local-heap, retaining only those labels originally present in the caller's heap.

Step (ii) above is accomplished using the formulae shown in Table 4. The formula $match$ is of special importance. It holds only for nodes v_1 and v_2 that meet the following conditions:

- v_1 and v_2 represent the same object o: v_1 represents o in the caller's local-heap at the memory-state in which the function was invoked; v_2 represents o in memory-state at the exit-site of the callee.
- o separates the caller's local-heap from the callee's local-heap, i.e., either o is pointed-to by a parameter ($matchArg$) or it is a cutpoint of the invocation ($matchCP$).

Note that the formula $samePath$ (used by $matchPaths$) is the *only* extended-transitive-closure formula that we use.

$$
\begin{array}{l}
\mathbf{extend}(\iota, O)(p^k)(u_1, \ldots, u_k) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\iota(p)(u_1, \ldots, u_k) & : \langle u_1, \ldots, u_k \rangle \in dom(\iota(p)) \\
0 & : u_i \in O \text{ for some } 0 < i \leq k \text{ and} \\
 & \quad \text{for all } 0 < j \leq k, u_j \in O \cup dom(\iota(p)) \\
\text{undefined} & : \text{otherwise}
\end{cases} \\
\mathbf{project}(\iota, O)(p^k)(u_1, \ldots, u_k) \stackrel{\text{def}}{=} \\
\quad \begin{cases}
\iota(p)(u_1, \ldots, u_k) & : \{u_1, \ldots, u_k\} \subseteq O \\
\text{undefined} & : \text{otherwise}
\end{cases}
\end{array}
$$

Figure 6: Interpretation Manipulating Functions.

The following example demonstrates how the formulae of Table 4 are used to combine structures upon return of `splice(p, q)` in the example program.

Example 4.4 Consider the concrete states shown at the first row of Figure 3. Structure (c) represents the local heap of the callee at the procedure's exit. Upon return, this structure is merged with the structure at the caller's call-site (a), resulting with the structure (d). Our operational semantics for procedure return updates the value of the predicate $y(v)$ (corresponding to a reference variables `y`) using the following update formulae:

$$
\begin{array}{l}
y'(v) = isObj(v) \land ((inUc(v) \land y(v) \land \neg R_{\{t,z\}}(v)) \lor \\
\qquad (inUx(v) \land \exists v_1 : y(v) \land inUc(v_1) \land R_{\{t,z\}}(v_1) \land \\
\qquad match_{main,\{\langle p,t\rangle,\langle q,z\rangle\}}(v_1, v)) \\
\text{where} \quad R_X(v) \stackrel{\text{def}}{=} \bigvee_{x \in X} \exists v_1 . x(v_1) \land F^*(v_1, v).
\end{array}
$$

In the structure which results from the combination of structures (c) and (a), this formula holds for the third individual of the list starting from `p`, as we are able to match the paths from the cutpoint label and the paths from `y`.

The predicates $x(v)$, $z(v)$, and $t(v)$ are updated similarly, but fall to the simpler matching case ($matchArg_{p,bind}(v_1, v_2)$).

Note that the operations used to model procedure call and return are operations that change the universe of a logical structure. We call such operations "universe altering". The universe altering functions use the helper functions **extend** and **project**, defined in Figure 6. The operation **extend** extends the partial interpretation mapping by providing a default value of 0 for objects in O for which ι is undefined. The operation **project** restricts the interpretation to be defined only for objects in a given set O.

The effects of the universe-altering functions are as follows:

- **clone**, duplicates all individuals that satisfy a given formula, and extends the interpretation accordingly. To record information about the newly allocated individuals, we use an additional auxiliary predicates new and $instance$ (defined

Predicate	Intended Meaning
$new(v)$	v is a newly created individual
$instance(v_1, v_2)$	v_1 is an instance of v_2
$inUc(v)$	v is a member of the caller's call-site universe
$inUx(v)$	v is a member of the callee's exit universe

Table 3: Auxiliary predicates for universe-altering operations.

$$
\begin{array}{l}
\textbf{clone}\colon \mathcal{WFF}_1 \times \mathit{2Struct} \to \mathit{2Struct} \text{ s.t.} \\
\quad \textbf{clone}(\varphi, \langle U, \iota \rangle) \stackrel{\text{def}}{=} \langle U', \iota' \rangle \text{ where} \\
\qquad O_{dup} = \{u.2 \mid u \in U, [\![\varphi(v)]\!]_2^{\langle U,\iota\rangle}(\langle v \mapsto u, \emptyset\rangle) = 1\} \\
\qquad U' = \{u.1 \mid u \in U\} \cup O_{dup} \\
\qquad \iota''(p)(u.1, \ldots, u_k.1) = \iota(p)(u_1, \ldots, u_k) \\
\qquad \iota' = \textbf{extend}(\iota'', O_{dup}) \\
\qquad\qquad \left[\begin{array}{l} \iota'(new)(v) \stackrel{\text{def}}{=} v = u, \\ \iota'(instance)(w, v) \stackrel{\text{def}}{=} w = u.1 \text{ and } v = u.2 \end{array} \right] \\
\textbf{combine}\colon \mathit{2Struct} \times \mathit{2Struct} \to \mathit{2Struct} \text{ s.t.} \\
\quad \textbf{combine}(\langle U^1, \iota^1 \rangle, \langle U^2, \iota^2 \rangle) \stackrel{\text{def}}{=} \langle U^{1.1} \cup U^{2.2}, \iota' \rangle \text{ where} \\
\qquad U^{1.1} = \{u.1 \mid u \in U^1\} \\
\qquad \iota^{1.1}(p)(u_1.1, \ldots, u_k.1) = \iota^1(p)(u_1, \ldots, u_k) \\
\qquad U^{2.2} = \{u.2 \mid u \in U^2\} \\
\qquad \iota^{2.2}(p)(u_1.2, \ldots, u_k.2) = \iota^2(p)(u_1, \ldots, u_k) \\
\qquad \iota' = (\textbf{extend}(\iota^{1.1}, U^{2.2}) \ \vee \ \textbf{extend}(\iota^{2.2}, U^{1.1})) \\
\qquad\qquad \left[\ inUc(u) \stackrel{\text{def}}{=} u = w.1, inUx(u) \stackrel{\text{def}}{=} u = w.2 \ \right] \\
\textbf{remove}\colon \mathcal{WFF}_1 \times \mathit{2Struct} \to \mathit{2Struct} \text{ s.t.} \\
\quad \textbf{remove}(\varphi, \langle U, \iota \rangle) \stackrel{\text{def}}{=} \langle U \setminus O, \iota' \rangle \text{ where} \\
\qquad O = \{u \in U \mid [\![\varphi(v)]\!]_2^{\langle U,\iota\rangle}(\langle v \mapsto u, \emptyset\rangle) = 1\} \\
\qquad \iota' = \textbf{project}(\iota, U \setminus O)
\end{array}
$$

Figure 7: Universe altering functions. $\mathcal{WFF}_1$ denotes the set of well-formed formulae in first-order logic with transitive closure that have a single free variable v.

in Table 3). These predicates record newly created individuals and the source object for each cloned object, respectively.

- **remove**, removes all individuals that satisfy a given formula, and restricts the interpretation accordingly.
- **combine**, combines two given structures into a single structure and extends the interpretation accordingly. To distinguish individuals that come from different universes we use the auxiliary predicates $inUc$ and $inUx$, defined in Table 3.

Shorthand	Formula
$match_{p,bind}(v_1, v_2)$	$inUc(v_1) \land isObj(v_1) \land inUx(v_2) \land isObj(v_2) \land$ $(matchArg_{p,bind}(v_1, v_2) \lor matchCP_{p,bind}(v_1, v_2))$
$matchArg_{p,bind}(v_1, v_2)$	$\bigvee_{\langle h,z\rangle \in bind} z(v_1) \land h(v_2)$
$matchCP_{p,bind}(v_1, v_2)$	$isCP_{p,\{z \mid \langle h,z\rangle \in bind\}}(v_1) \land \bigwedge_{\langle h,z\rangle \in bind} (R_{\{z\}}(v_1) \implies matchPaths_{h,z}(v_1, v_2))$
$matchPaths_{h,z}(v_1, v_2)$	$\forall v_z \colon inUc(v_z) \land isObj(v_z) \land$ $\forall l_h \colon inUx(l_h) \land isLb_O(l_h) \land \forall l_2 \colon inUx(l_2) \land isLb_{CP}(l_2) \land$ $(z(v_z) \land \widehat{h}(l_h) \land lbl(l_2, v_2) \implies samePath(v_z, v_1, l_h, l_2))$
$samePath(v_x, v_{cp}, l_x, l_{cp})$	$(TC\ v_1, v_2; w_1, w_2 : \bigvee_{f \in FieldId^\star_P} f(v_1, v_2) \land \widehat{f}(w_1, w_2))\ (\ v_x, v_{cp}; l_x, l_{cp})$
$isCP_{p,X}(v)$	$R_X(v) \land \bigwedge_{x \in X} \neg x(v) \land$ $(\bigvee_{y \in V_p} y(v) \lor \exists v_1. \neg R_X(v_1) \land F(v_1, v) \lor \exists v_1. isLb_{CP}(v_1) \land lbl(v_1, v))$

Table 4: Shorthand notations for formulae used to match individuals and paths when combining structures on procedure return. $FieldId^\star_P$ denotes the set of all the pointer-valued fields that are used in the program. V_p denotes the set of all local-variables (including formal parameters) of procedure p.

5 Abstract Semantics

In this section, we present a conservative abstract semantics abstracting the instrumented concrete semantics of Section 4.

5.1 Abstract States

We conservatively represent multiple states using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 and which may be 0.

An *abstract state* is a 3-valued logical structure $S{=}\langle U^S, \iota^S\rangle$ where:

- $U^{S^\sharp}$ is the universe of the structure. Each individual in $U^{S^\sharp}$ possibly represents many heap-allocated objects or object-labels.
- $\iota^{S^\sharp}$ is the partial interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota^S(p) \colon {U^{S^\sharp}}^k \to \{0, \frac{1}{2}, 1\}$ or is undefined.

An abstract state may include *summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. For a summary node u, $\iota^{S^\sharp}(eq)(u, u) = \frac{1}{2}$.

Canonical Abstraction We now formally define how states are represented using abstract states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. More generally, it is possible to map individuals from an abstract state into an individual in another less precise abstract state.

Formally, let $S^\sharp = \langle U^{S^\sharp}, \iota^{S^\sharp} \rangle$ and $S^{\sharp\prime} = \langle U^{S^{\sharp\prime}}, \iota^{S^{\sharp\prime}} \rangle$ be abstract states. A function $f\colon U^{S^\sharp} \rightarrow U^{S^{\sharp\prime}}$ such that f is surjective is said to *embed* $S^{S^\sharp}$ *into* $S^{S^{\sharp\prime}}$ if for each predicate $p \in \mathcal{P}$ of arity k, and for each $u_1, \ldots, u_k \in U^{S^\sharp}$ the following holds:

$$\iota^{S^\sharp}(p)(u_1, \ldots, u_k) = \iota^{S^{\sharp\prime}}(p)(f(u_1), \ldots, f(u_k)) \\ \text{or } \iota^{S^{\sharp\prime}}(p)(f(u_1), \ldots, f(u_k)) = \tfrac{1}{2}$$

One way of creating an embedding function f is by using *canonical abstraction*. Canonical abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual. Only summary nodes (i.e., nodes with $\iota^{S^{\sharp\prime}}(eq)(u, u) = \frac{1}{2}$) can have more than one node mapped to them by the embedding function.

It is possible to obtain a more coarse abstraction by: (i) fixing an (arbitrary) *subset* of the unary predicate symbols as *abstraction predicates*, and (ii) mapping all individuals having the same values for the abstraction predicates to the same abstract individual. This is done, for example, in [12]. In this paper, we use canonical abstraction, We use all the unary predicate symbols as abstraction predicates, except those of the form $\widehat{q}$, where `q` is a reference parameter. Effectively, this means that we have an heterogenous abstraction [25]: Nodes representing heap-allocated object are abstracted using "standard" canonical abstraction (i.e., different individuals are mapped to the same abstract individual only if they have the same values for *all* unary predicate symbols). For object-labels, we use a type-based abstraction: we conservatively represent all non-cutpoint object-labels resp. cutpoint-labels of the same type using a single abstract object-label resp. abstract cutpoint-label.

3-valued logical structures are also drawn as directed graphs. Definite values (0 and 1) are drawn as for 2-valued structures. Binary indefinite predicate values ($\frac{1}{2}$) are drawn as dotted directed edges. Summary nodes are depicted by a double frame. Also, we draw a dotted edge from $\widehat{q}$ to an *object-label* u when $\iota^{S^\sharp}(\widehat{q})(u) = \frac{1}{2}$.

> **Example 5.1** The *3-valued* logical structure that conservatively represents the local heap when `s=splice(t,z)` is invoked is depicted in Figure 3(e).

Instrumentation Predicates Recording derived properties by means of *instrumentation predicates* may provide additional information that would have been otherwise lost under abstraction. In particular, because canonical abstraction is directed by unary predicates, adding unary instrumentation predicates may further refine the abstraction. This is called the *instrumentation principle* in [20].

5.2 Abstract Operational Semantics

Because our framework is based on [20], the actions we used to define the concrete operational semantics for program statements (as transformers of 2-valued structures) in Section 4, also define the corresponding abstract semantics (as transformers of 3-valued structures). This abstract semantics is obtained by reinterpreting logical formulae using a 3-valued logic semantics and serves as the basis for an abstract interpretation. In this paper, we directly utilize the implementation of these ideas available in TVLA.

In this paper, we manually provide the update formulae of the instrumentation predicates (as done e.g., in [10,19,20,24]). Automatic derivation of update formulae for the instrumentation predicates [16] is currently not implemented for the types of structure manipulation employed by the **clone** and **remove**.

The soundness of our abstract semantics is guaranteed by the combination of the theorems in Appendix B, [18], and [20]:

- In Section B, we show that the concrete semantics used in this paper is equivalent to (an instrumented version of) the $\mathcal{LSL}$ semantics defined in [18].
- In [18], it is shown that the $\mathcal{LSL}$ semantics is *observationally equivalent* to a standard store-based global-heap semantics.
- In [20], it is shown that every program-analyzer which is an instance of their framework is sound with respect to the concrete semantics it is based on.

6 Prototype Implementation

We have implemented a prototype of our framework using TVLA [11]. To translate Java programs and their specifications to TVP (TVLA input language) we have extended an existing Soot-based [22] front-end for Java developed by R. Manevich.

We handle dynamic-dispatch by selectively propagating *3-valued* logical structure over an interprocedural edge according to the type of the receiver object (information maintained in our analysis, see Table 1). In our implementation, we do not use set-union as join-operator. Instead, we use a more "aggressive" partial-join operation [12]. This operation exploits the fact that our abstract domain has a Hoare order and returns an upper approximation of the set-union operator.

Our framework allows control over the heap-abstraction and the cutpoint abstraction in the instantiated algorithms. We have instantiated the framework to produce a shape-analysis algorithm for analyzing general heap-manipulating programs. In our experiments we used a class-based abstraction for cutpoints, i.e., we do not distinguish between different cutpoints of the same type. This means that we lose precision when a procedure is invoked with two or more cutpoint objects of the same class.

We applied our framework to verify various correct list-manipulating programs and to verify client conformance with API specification. Our measurements were obtained on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

Our analysis was able to verify that the list-manipulating programs do not perform null-dereferences and that the lists they manipulate are acyclic. Measurements of analysis cost for these programs are shown in Table 5(a). The table compares the cost of the analysis of a program which invokes an iterative procedure (first column) with the cost of the analysis of the same program, except that the invoked procedure is recursive[1] (second column). For these programs, we found that the cost of analyzing recursive procedures and iterative procedures is comparable in most cases. We note that our tests

[1] revApp is a recursive procedure. We analyzed it once one with an iterative append procedure and once with a recursive append.

a. Program	Iterative		Recursive	
	Space (MB)	Time (Sec)	Space (MB)	Time (Sec)
create creates a list	19.7	10.9	19.3	9.3
find searches an element in a list	22.3	21.3	23.5	35.8
insert allocates and inserts an element into a sorted list	23.3	41.2	23.3	41.2
delete removes an element from a sorted list	23.2	42.0	24.8	45.3
append appends two lists	25.1	17.2	25.6	20.2
reverse destructive list-reversal	23.6	23.7	24.0	33.7
revApp reverse a list by appending its head to the reversed tail	26.0	45.7	26.5	46.8
merge merges two sorted lists	25.9	579.7	27.8	91.9
splice splices two lists	25.5	70.1	26.1	36.9
running the running example	27.7	160.0	28.3	45.7
b. Code fragment	**Code Inline**		**Proc. Calls**	
crt3 creates a list of length 3	22.3	5.4	22.0	6.4
crt3x3 creates 3 lists of length 3	50.7	27.0	26.2	9.2

Table 5: Analysis cost for list-manipulating programs.

Program	Line No.	Space (MB)	Time (Sec)	Rep. / Act. Errors
ISPath	71	24.9	1378.0	0/0
InputStream5	64	61.8	2484.4	0/0
InputStream5b	64	61.7	2550.5	1/1
JDBC Example fixed	153	191.0	25213.0	0/0
JDBC Example	149	191.9	25261.3	1/1

Table 6: Analysis results and cost for verifying client conformance with API specification.

were of *client* programs and not a single procedure, i.e., in all tests, the program also allocates the list it manipulates. In addition, we compared the cost of the analysis of the three calls to `crt3` in our running example to the cost analyzing this code fragment when the body of `crt` is inlined in the `main` procedure (see Table 5(b)). We are encouraged by these results, as they indicate (at least in this simple example) that our analysis benefits from procedural abstraction.

We also applied our framework to verify correct usage of the `IO`-stream and the `JDBC` interfaces. In particular, for we verified that closed files are not read (`IO`-streams) and that clients use database connections correctly (`JDBC`-client). The analysis cost and results are shown in Table 6. The column "Rep. / Act. Error" shows the number of reported errors, compared to the number of actual errors. `ISPath` is a simple correct program manipulating input streams. `InputStream5` is a program that stores input-streams at arbitrarily deep recursive data structures. `InputStream5b` is an erroneous version of `InputStream5` containing a single error. `JDBC Example` is an erroneous program which manipulates 5 database connection. `JDBC Example fixed` is a correct version of the last program. These programs were part of the benchmarks used in [25]. Our analysis verify the correct use of the API with the same precision as the separation-based approach, but *without the need for specification*. However, the cost of the analysis was higher than the cost of the analysis in [25]. We attribute these to the fact that our heap analysis is more precise than the one used in [25].

7 Related Work

Heap Abstractions There are many techniques that can be combined with local heaps to scale shape analysis. First our prototype implementation already benefits from the abstraction described in [12], which merges similar shape graphs. This reduces the number of shape graphs and the running time. There are additional methods that can be employed. Staged analysis can be used to reduce the cost of abstraction by first computing context-sensitive flow-insensitive points-to analysis (e.g., [23]) and then employing our fine-grained abstraction on parts of the heap which may be aliased. It is possible to drastically improve our results by specializing the analysis (e.g, [2,14]) to prove certain program properties. Finally, in [25] it is shown how to radically improve the efficiency of shape analysis by using heterogenous abstractions, specified by the analysis designer.

Interprocedural Shape Analysis Interprocedural shape analysis has been studied in [8, 19]. In [19], the main idea is to make the runtime stack an explicit data structure and abstract it as a linked list. In this method, the entire heap and run-time stack are represented at every program point. As a result, the abstraction may lose information about properties of the heap, *for parts of the heap that cannot be affected by the procedure at all*. In [8], procedures are considered as transformers from the (entire) program heap before the call, to the (entire) program heap after the call. Every heap-allocated object is represented at every program point; on the other hand, only the values of the local variables of the current procedure are represented, which means that the irrelevant parts of the heap are summarized to a single summary node during the analysis of an invoked procedure.

A heap-modular interprocedural shape-analysis algorithm is presented in [4]. A procedure is analyzed only in the part of the heap that is reachable from its parameters. The algorithm is able to relate the memory states at the procedure-entry with the memory states at the procedure-exit by labeling *every* abstract node. However, the mapping is determined by the sharing within the part of the heap that is passed to the procedure, and not by the sharing pattern with the context—which is what is needed.

8 Conclusions

In this paper, we presented an interprocedural analysis which is precise enough for programs with a small number of cutpoints. In particular, it is possible to handle libraries in a scalable and precise way by providing natural Java methods for these libraries.

Also, the analysis of recursive and iterative versions of natural data structure manipulating procedures behave similarly in terms of precision and costs. We are encouraged by this because it means that our method supports procedural abstraction—often the handling of procedure is as precise as inlining the procedure body and the cost can be smaller when the code is reused. In fact, in some cases the precision of the analysis can be improved by procedural abstraction. In particular, the analysis of recursive procedures can be more efficient or even more precise than the analysis of the iterative version.

References

[1] T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[2] B. Blanchet, P. Cousot, R. Cousot, J.Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software (invited chapter). In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *In The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lec. Notes in Comp. Sci.*, pages 85–108. Springer, 2004.

[3] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[4] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977)*, pages 237–277. North-Holland, 1978.

[6] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 7–68, 2002.

[7] N. Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, New York, 1999.

[8] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium*, 2004.

[9] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct.*, pages 125–140, 1992.

[10] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 26–38, 2000.

[11] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. http://www.math.tau.ac.il/∼ tvla.

[12] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In Roberto Giacobazzi, editor, *Static Analysis Symposium*, volume 3148 of *Lec. Notes in Comp. Sci.*, pages 265–279. Springer, August 2004.

[13] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[14] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, volume 37, 5, pages 83–94, June 2002.

[15] 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.

[16] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, volume 2618, 2003.

[17] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. Tech. Rep. 1, AVACS, September 2004. Available at "*http://www.avacs.org*".

[18] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Symp. on Princ. of Prog. Lang.*, 2005.

[19] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct.*, pages 133–149, 2001.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[21] 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.

[22] R. Vallée-Rai, L. Hendren, V. Sundaresan, E. Gagnon P. Lam, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[23] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 131–144. ACM Press, 2004.

[24] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, January 2001.

[25] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.

A Formal Specification of the Operational Semantics

This appendix provides additional formal details. In Section A.2 we define the notion of garbage collected *2-valued* logical structures. In Section A.2 we extend the logic of [20] to support extended transitive closure. In Section A.3 we formally define the operational semantics for function call and return.

A.1 Garbage Collected States

Definition A.1 (Garbage-Collected 2-Valued Logical Structures) *An* admissible 2-valued *logical structure $S = \langle U, \iota\rangle$ representing a local-heap for a function p in program P at a given point in an execution is **garbage-collected** iff every* object *is reachable from either a variable or from a frozen variable, i.e.,*

$$S \models R_{V_p}(v) \vee \exists v_1, v_2 : isLb_{CP}(v_1) \wedge lbl(v_1, v_2) \wedge (TC\ w_1, w_2 : F(w_1, w_2))(v_2, v)$$

A.2 Extended Transitive Closure

The logic used in [20] is first order logic with transitive closure. In [20], formulae with transitive closure have only one pair of variables. For example, the bounded variables in the formula $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$ are v_1 and v_2. To define the semantics of the return operation we need to allow a transitive closure formulae with two pairs of free variables, as conducted, e.g., in [7].

Formally, extended transitive closure formulae are of the form:

$$(TC\ v_1, v_2; w_1, w_2 : \varphi)(v_3, v_4; w_3, w_4) \tag{1}$$

The meaning of formulae in extended transitive closure is

$$\begin{aligned}&[\![(TC\ v_1, v_2; w_1, w_2 : \varphi)(v_3, v_4; w_3, w_4)]\!]_2^{\langle U,\iota\rangle}(Z) \stackrel{\text{def}}{=} \\ &\max_{\substack{n \in \mathbb{N},\\ u^1_1, \ldots, u^1_{n+1} \in U,\\ Z(v_3) = u^1_1,\, Z(v_4) = u^1_{n+1},\\ u^2_1, \ldots, u^2_{n+1} \in U,\\ Z(w_3) = u^2_1,\, Z(w_4) = u^2_{n+1}}} \min_{i=1}^{n} [\![\varphi]\!]_2^{\langle U,\iota\rangle}(Z\left[\begin{array}{l} v_1 \mapsto u^1_i, v_2 \mapsto u^1_{i+1}, \\ w_1 \mapsto u^2_i, w_2 \mapsto u^2_{i+1}\end{array}\right])\end{aligned} \tag{2}$$

Lemma A.2 (Extended Embedding) *Let φ be an extended transitive closure formulae. For any $S \in 2Struct$ and $S^\sharp \in 3Struct$ such that $S \sqsubseteq^f S^\sharp$ and for any assignment Z it holds that,*

$$[\![\varphi]\!]_2^S(Z) \sqsubseteq [\![\varphi]\!]_3^{S^\sharp}(f \circ Z)$$

Statement	Predicate-update formulae
`y = null`	$y'(v) = 0$
`y = x`	$y'(v) = x(v)$
`y = x.f`	$y'(v) = \exists v_1 . x(v_1) \wedge f(v_1, v)$
`y.f = null`	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$
`y.f = x`	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$

Figure 8: The predicate-update formulae defining the operational semantics of assignments.

Statement	$y = \mathtt{alloc}(T)$
Prepare	$\mathbf{newNode}(\mathbf{addPreds}(S, \{\mathit{new}\}))$
Predicate–update formulae	$\mathit{isObj}'(v) = \mathit{isObj}(v) \vee \mathit{new}(v)$ $y'(v) = \mathit{new}(v)$ $T'(v) = \neg \mathit{new}(v) \wedge T(v) \vee \mathit{new}(v)$
Clean	$\mathbf{removePreds}(S', \{\mathit{new}\})$

Figure 9: The predicate-update formulae defining the operational semantics of object allocation.

A.3 Operational Semantics

The operational semantics is specified by *predicate-update formulae*: for every predicate p and for every statement st, the value of p in the 2-valued structure S', which results by applying st to S, is defined in terms of a formula evaluated over S.

The predicate-update formulae of the core-predicates for assignments is given in Figure 8. The value of every core-predicate p after the statement executes, denoted by p', is defined in terms of the core predicate values before the statement executes (denoted without primes). Core predicates whose update formula is not specified, are assumed to be unchanged, i.e., $p'(v_1, \ldots) = p(v_1, \ldots)$.

The operational semantics for object allocation is given in Figure 9. The operational semantics for function invocations is given in Figure 10 and in Figure 11.

Definition A.3 (Transition Relation) *Let p be a procedure. Let st is a statement in p. The transition relation $\overset{2}{\rightsquigarrow} \subseteq (\mathit{2Struct}_p \times st) \times \mathit{2Struct}_p$ contains $\langle S, S' \rangle$ iff (i) S and S' are garbage-collected* 2-valued *logical structures for procedure p and (ii) applying the predicate-update formulae (action) associated with st to S results in S'.*

Statement : $iCall_q^{y=p(x_1,\ldots,x_k)}$

Prepare

$\mathbf{clone}(R_{\{x_1,\ldots,x_k\}}(v), \mathbf{addPreds}(S^c, \{new, instance\}))$

Predicate − updateformulae

$$y'(v) = \begin{cases} x_i(v) & : \quad y = h_i \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$f'(v_1, v_2) = f(v_1, v_2) \wedge R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2)$$

$$isObj'(v) = r_{x_1,\ldots,x_k}(v)$$

$$T'(v) = r_{x_1,\ldots,x_k}(v) \wedge T(v) \vee \exists v_{obj}, instance(v_{obj}, v) \wedge T(v_{obj})$$

$$isLb'_O(v) = isLb_O(v) \vee new(v)$$

$$isLb_{CP}{}'(v) = new(v) \wedge \exists v_{obj}.instance(v_{obj}, v) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_{obj})$$

$$lbl'(v_1, v_2) = instance(v_2, v_1) \wedge isCP_{q,\{x_1 \ldots x_k\}}(v_2)$$

$$\widehat{y}'(v) = new(v) \wedge \begin{cases} \exists v_{obj} : x_i(v_{obj}) \wedge instance(v_{obj}, v) & : \quad y = h_i \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\widehat{f}'(v_1, v_2) = new(v_1) \wedge new(v_2) \wedge \exists v_{obj1}, v_{obj2} : instance(v_{obj1}, v_1) \wedge instance(v_{obj2}, v_2) \wedge f(v_{obj1}, v_{obj2})$$

$$eq'(v_1, v_2) = (R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge eq(v_1, v_2)) \vee (new(v_1) \wedge new(v_2) \wedge \exists v_{obj} : instance(v_{obj}, v_1) \wedge instance(v_{obj}, v_2))$$

Clean

Let $S'' = \mathbf{remove}(isObj(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee isLb_O(v) \wedge \neg new(v)), S')$

in $\langle U^{S''}, \mathbf{remPreds}(\iota'', \{new, instance\})\rangle$

Figure 10: The operational semantics for function calls: Construction of the structure at the *entry* to a callee. We give the semantics for an arbitrary function call $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. We assume that the p's formal parameters are $h_1, \ldots, h_k$.

Statement : $iRet_q^{y=p(x_1,\ldots,x_k)}$

Prepare

$$\mathbf{combine}(\mathbf{addPreds}(S^c, \{inUc, inUx\}),\ \mathbf{addPreds}(S^x, \{inUc, inUx\}))$$

Predicate – updateformulae

$$x'(v) = isObj(v) \wedge \begin{cases} ret(v) & x = y \\ (inUc(v) \wedge x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v)) \vee \\ (inUx(v) \wedge \exists v_1 : x(v) \wedge inUc(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_1) \wedge & x \in V_q \setminus \{y\} \\ \quad match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v)) \end{cases}$$

$$\begin{aligned} f'(v_1, v_2) = {} & isObj(v_1) \wedge isObj(v_2) \wedge \\ & (inUc(v_1) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\ & \ inUx(v_1) \wedge inUx(v_2) \wedge f(v_1, v_2) \vee \\ & \ \ inUc(v_1) \wedge inUx(v_2) \wedge \\ & \ \ \ \exists v_{cp} : inUc(v_{cp}) \wedge isObj(v_{cp}) \wedge f(v_1, v_{cp}) \wedge \\ & \ \ \ \ match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{cp}, v_2)) \end{aligned}$$

$$\begin{aligned} isObj'(v) = {} & isObj(v) \wedge \\ & (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v)) \end{aligned}$$

$$T'(v) = T(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v))$$

$$isLb'_O(v) = isLb_O(v) \wedge inUc(v)$$

$$isLb_{CP}{}'(v) = isLb_{CP}(v) \wedge inUc(v)$$

$$\begin{aligned} lbl'(v_1, v_2) = {} & isLb_{CP}(v_1) \wedge inUc(v_1) \wedge isObj(v_2) \wedge \\ & (inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge lbl(v_1, v_2) \vee \\ & \ inUx(v_2) \wedge \exists v_{cp} : R_{\{x_1,\ldots,x_k\}}(v_{cp}) \wedge isObj(v_{cp}) \wedge \\ & \ \ lbl(v_1, v_{cp}) \wedge match_{q,\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{cp}, v_2)) \end{aligned}$$

$$\widehat{y}'(v) = \widehat{y}(v) \wedge inUc(v) \wedge isLb_O(v)$$

$$\widehat{f}'(v_1, v_2) = isLb_O(v_1) \wedge isLb_O(v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \widehat{f}'(v_1, v_2)$$

$$eq'(v_1, v_2) = eq(v_1, v_2)$$

Clean

Let $S'' = \mathbf{remove}(isObj(v) \wedge inUc(v) \wedge R_{\{x_1,\ldots,x_k\}}(v) \vee isLb_O(v) \wedge inUx(v), S')$
in $\langle U^{S''}, \mathbf{remPreds}(\iota'', \{inUc, inUx\})\rangle$

Figure 11: The operational semantics for function calls: Construction of the structure at the *return-site* to a caller. We give the semantics for an arbitrary function call $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. We assume that the p's formal parameters are $h_1, \ldots, h_k$.

B Access-Path-Based Local-Heap Semantics

In this section, we show that the concrete semantics used in this paper is equivalent to $i\mathcal{LSL}$,[2] an instrumented version of the $\mathcal{LSL}$ semantics presented in [18]. $\mathcal{LSL}$ is a local-heap storeless semantics which is specified using an equivalence relation over a set of access paths. In [17] it is shown that abstractions of $\mathcal{LSL}$ can be used to verify invariants expressed by access-paths equality, to assert the absence of null-valued pointer dereferences, and to detect memory-leaks. An immediate consequence is that our logic-based semantics presented in this paper can be used for these purposes.

B.1 Local-Heap Instrumented Storeless Semantics

In this section, we define the $i\mathcal{LSL}$ semantics. $i\mathcal{LSL}$ is a non-standard, large-step operational, storeless semantics for E-Algol. Like $\mathcal{LSL}$, the LISLI semantics is also specified using an equivalent relation over a set of access paths. $i\mathcal{LSL}$ is an instrumentation of $\mathcal{LSL}$ [18]: It records more information than $\mathcal{LSL}$. The main difference between $\mathcal{LSL}$ and $i\mathcal{LSL}$ is that in $i\mathcal{LSL}$, when a procedure is invoked, $i\mathcal{LSL}$ records the set of access paths that reach *every* object in the local-heap of the invoked procedure. $\mathcal{LSL}$, on the other hand, only records the set of access paths that reach cutpoint objects.

We refer to the set of access paths that reach an object o at procedure-entry in a given invocation, as o's object-label for that invocation.

B.1.1 Memory States

Figure 12 defines the semantics domains. A memory state $\langle CPL, OLB, H\rangle$ at a given point in an execution is composed of the labels of all the cutpoints of the current invocation (CPL), of the labels of all the objects that were passed to the procedure when it was invoked (OLB), and of a representation of the heap (H) at that the point in the execution. Note that although the state contains the labels of every object in the local-heap that was allocated prior to the invocation, the description of objects is the same as in $\mathcal{LSL}$ [18], i.e., it is comprised *only* of access paths and cutpoint-anchored paths. To exclude states that cannot arise in any program, we now define the notion of *admissible states*.

Definition B.1 (Admissible memory states) *A memory state $\langle CPL, OLB, H\rangle$ for a procedure p at a given point in an execution is **admissible** iff*

(i) $\langle$CPL$, A\rangle$ *is an admissible $\mathcal{LSL}$ memory-state for procedure p.*

(ii) Object-labels are mutually disjoint, i.e., for every $olb_1, olb_2 \in OLB$, if $olb_1 \neq olb_2$, then $olb_1 \cap olb_2 = \emptyset$.

(iii) OLB is right-regular, i.e., for every $olb_1, olb_2 \in OLB$, if $\alpha, \beta \in olb_1$ and $\alpha.\delta \in olb_2$ then $\beta.\delta \in olb_2$.

[2] $i\mathcal{LSL}$ stands for *Instrumented-$\mathcal{LSL}$*.

α	$\in$	$\text{AP}\textsc{f}_p = F_p \times \Delta$	Access paths that start with a formal parameter.
cpl	$\in$	$\text{LB}\textsc{cp}_p = 2^{\text{AP}\textsc{f}_p}$	Cutpoint labels.
olb	$\in$	$\text{LB}\textsc{o}_p = 2^{\text{AP}\textsc{f}_p}$	Object-labels.
r	$\in$	$\text{Root}_p = V_p \cup \text{LB}\textsc{cp}_p$	Roots of access paths.
α	$\in$	$\text{AP}\textsc{g}_p = \text{Root}_p \times \Delta$	Generalized access-paths.
o	$\in$	$\text{Obj}_L^p = 2^{\text{AP}\textsc{g}_p}$	Objects.
H	$\in$	$\text{Heap}\textsc{f}_L^p = 2^{\text{Obj}_L^p}$	Heaps.
σ_{iL}	$\in$	$\Sigma_{iL}^p = 2^{\text{LB}\textsc{cp}_p} \times 2^{\text{LB}\textsc{o}_p} \times \text{Heap}\textsc{f}_L^p$	Memory states.

Figure 12: Semantic domains of the $i\mathcal{LSL}$ semantics. We use the syntactic domains V_p and $\text{AP}\textsc{g}_p$ as semantic domains, too (and use italics font to denote a semantics value.)

$$\frac{\langle \text{body of } p, \langle CPL^e, OLB^e, H^e\rangle\rangle \overset{iL}{\rightsquigarrow} \langle CPL^e, OLB^e, H^x\rangle}{\langle y = p(x_1, \ldots, x_k), \langle CPL^c, OLB^c, H^c\rangle\rangle \overset{iL}{\rightsquigarrow} \langle CPL^c, OLB^c, H^r\rangle}$$

$$\begin{array}{ll} \textit{where} & \\ & \langle \textit{CPL}^e, A^e\rangle = \textit{Call}_q^{y=p(x_1,\ldots,x_k)}(\langle \textit{CPL}^c, A^c\rangle) \\ & OLB^e = \textit{Let} \circledast \textit{ in } \textit{map}(\textit{sub}(\textit{bind}_{\textit{args}}))\, O_c^{\textit{passed}} \\ & \langle \textit{CPL}^c, A^r\rangle = \textit{Ret}_q^{y=p(x_1,\ldots,x_k)}(\langle \textit{CPL}^c, A^c\rangle, \langle \textit{CPL}^e, A^x\rangle) \end{array}$$

Figure 13: The inference rule for a procedure call in $i\mathcal{LSL}$. $\circledast$ is defined in [18].

(iv) OLB *is prefix-closed, i.e., if* $\alpha.f$ *is in flat* OLB*, then* α *is also in flat* OLB.

(v) CPL $\subseteq OLB$.

(vi) $\emptyset \notin OLB$.

B.1.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{iL}{\rightsquigarrow} \subseteq (\sigma_{iL} \times \textit{stms}) \times \sigma_{iL}$. The execution of intraprocedural statements does not modify the object-labels component of the state. Invoking a parocedure creates a label for every object which is passed to the callee. The inference rule for a procedure-call in the instrumented semantics is given in Figure 13.

B.2 Semantics Equivalence

In this section we show that the concrete semantics used in this paper is equivalent to $i\mathcal{LSL}$.

$$
\begin{array}{|l|}
\hline
\mathit{to2VLSi}_p \colon \Sigma^p_{iL} \to \mathit{2Struct}_p \text{ s.t.} \\
\mathit{to2VLSi}_p(\langle \mathit{CPL}, \mathit{OLB}, H\rangle) = \langle U, \iota\rangle \text{ where} \\
\quad U = H \cup \mathit{OLB} \text{ and} \\
\quad \begin{array}{lcl}
\iota(\mathit{isObj})(v) & = & v \in H \\
\iota(\mathit{isLb}_O)(v) & = & v \in \mathit{OLB} \\
\iota(\mathit{isLb}_{CP})(v) & = & v \in \mathit{CPL} \\
\iota(x)(v) & = & v \in H \text{ and } \langle x, \epsilon\rangle \in v \\
\iota(f)(v_1, v_2) & = & v_1 \in H, v_2 \in H \text{ and } v_1.n \subseteq v_2 \\
\iota(\widehat{x})(v) & = & v \in \mathit{OLB} \text{ and } \langle x, \epsilon\rangle \in v \\
\iota(\widehat{f})(v_1, v_2) & = & v_1 \in \mathit{OLB}, v_2 \in \mathit{OLB} \text{ and } v_1.n \subseteq v_2 \\
\iota(\mathit{eq})(v_1, v_2) & = & v_1 = v_2 \\
\iota(\mathit{lbl})(v_1, v_2) & = & v_1 \in \mathit{CPL},\ v_2 \in H \text{ and } \langle v_1, \epsilon\rangle \in v_2
\end{array} \\
\hline
\end{array}
$$

Figure 14: The procedure *to2VLS* maps admissible memory states to (garbage-collected) 2-valued logical structures.

Lemma B.2 *For every admissible memory state $\sigma_{iL} \in \Sigma^P_{iL}$ for procedure p, there exists a* unique *garbage-collected* 2-valued *logical structure $S \in \mathit{2Struct}_P$ representing a local-heap for procedure p such that $S = \text{to2VLS}(\sigma_{iL})$.*

Sketch of Proof: The function $\mathit{to2VLSi}_p$, defined in Figure 14, maps an admissible instrumented memory-state $\sigma_{iL} = \langle \mathit{CPL}, \mathit{OLB}, H\rangle$ for procedure p to a *2-valued* logical structure S representing a local-heap for procedure p. Every object $o \in H$ and every object-label $\mathit{olb} \in \mathit{OLB}$ is represented by a unique node in U^S. Note that because $\mathit{CPL} \subseteq \mathit{OLB}$ every cutpoint-label is represented by a node in U^S. Tracked properties of memory states are recorded by the *core-predicates* given in Table 1. Note that U^S, the universe of the resulting *2-valued* logical structure, is potentially unbounded because every object and every object-label is mapped to a unique node.

The function toiLSL_p, defined in Figure 14, maps garbage-collected *2-valued* logical structures representing a local-heap for procedure p to admissible memory states for procedure p. Every object-label is mapped to the set of access paths that start with a formal and pointed-to the object when the procedure started. A similar operation is applied to the cutpoint-labels. Every object is mapped to the set of access paths that point to it in the given local-heap *and* the set of cutpoint-anchored paths [18] that point to it.

Corollary B.3 *Let $\sigma_{iL} \in \Sigma^P_{iL}$ be an admissible memory state for procedure p. Let $S \in \mathit{2Struct}_P$ be garbage-collected* 2-valued *logical structure for procedure p. It holds that*

$$S = \mathit{to2VLSi}_p(\sigma_{iL}) \iff \sigma_{iL} = \mathit{toiLSL}_p(S).$$

$$
\begin{array}{l}
\mathit{toiLSL}_p\colon \mathit{2Struct}_p \to \Sigma^p_{iL} \text{ s.t.}\\
\mathit{toiLSL}_p(S) = \text{Let}\\
\quad \mathit{CPL} = \{\ \mathit{reachingAccessPaths}(S,u) \mid S[v \mapsto u] \models \mathit{isLb}_{CP}(v)\ \}\\
\quad \mathit{OLB} = \{\ \mathit{reachingAccessPaths}(S,u) \mid S[v \mapsto u] \models \mathit{isLb}_{O}(v)\ \}\\
\quad H = \{\ \mathit{reachingGeneralizedAccessPaths}(S,u) \mid S[v \mapsto u] \models \mathit{isObj}(v)\ \}\\
\text{in}\\
\quad \langle \mathit{CPL}, \mathit{OLB}, H \rangle\\
\\
\text{Where}\\
\ \mathit{reachingAccessPaths}\colon \mathit{2Struct}_p \times \mathcal{U} \to 2^{\mathrm{APF}_p} \text{ s.t.}\\
\ \mathit{reachingAccessPaths}(S,u) =\\
\qquad \{\ \langle x, \delta \rangle \mid x \in F_p, \delta \in \Delta, S[v_u \mapsto u] \models \widehat{x}(v_{\widehat{x}}) \wedge \mathit{reaches}(\delta, v_{\widehat{x}}, v_u)\ \}\\
\\
\ \mathit{reachingGeneralizedAccessPaths}\colon \mathit{2Struct} \times \mathcal{U} \to 2^{\mathrm{APG}_p} \text{ s.t.}\\
\ \mathit{reachingAccessPaths}(S,u) =\\
\qquad \{\ \langle x, \delta \rangle \mid x \in F_p, \delta \in \Delta, S[v_u \mapsto u] \models \widehat{x}(v_x) \wedge \mathit{reaches}(\delta, v_x, v_u)\ \} \cup\\
\qquad \left\{ \langle \mathit{reachingAccessPaths}(S, u_{cp}), \delta \rangle \;\middle|\; \begin{array}{c} \delta \in \Delta, u_{cp} \in U^S\\ S[v_{cp} \mapsto u_{cp}, v_u \mapsto u,] \models\\ \mathit{isLb}_{CP}(v_{cp}) \wedge \mathit{reaches}(\delta, v_{cp}, u) \end{array} \right\}\\
\\
\mathit{reaches}(\delta, v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} \mathit{eq}(v_1, v_2) & \delta = \epsilon\\ \exists v : f(v_1, v) \wedge \mathit{reaches}(\delta', v, v_2) & \delta = f\delta' \end{cases}
\end{array}
$$

Figure 15: The procedure toiLSL maps garbage-collected 2-valued logical structures to admissible memory states.

Theorem B.4 (Equivalence) *Let p be a procedure. Let $\sigma_{iL} \in \Sigma^P_{iL}$ be an admissible memory state for procedure p and let $S \in \mathit{2Struct}_P$ be garbage-collected* 2-valued *logical structure for procedure p such that $S = \mathit{to2VLSi}_p()$. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, \sigma_{iL} \rangle \stackrel{iL}{\rightsquigarrow} \sigma'_{iL} \iff \langle st, S \rangle \stackrel{2}{\rightsquigarrow} S'.$$

Furthermore, $S' = \mathit{to2VLSi}_p(\sigma'_{iL})$

Remark. The function $\mathit{to2VLSi}_p$, defined in [18], maps $\mathcal{LSL}$ states to *2-valued* logical structures. This function is not injective. In particular, it does not store in the *2-valued* logical structure any information regarding the contents of the cutpoint-labels. On the other hand, the mapping of $i\mathcal{LSL}$ states to *2-valued* logical structures defined in this paper, does not lose *any* information.