Interprocedural Shape Analysis for Cutpoint-Free Programs

Technical Report TAU-CS-104/05

Noam Rinetzky*
Tel Aviv University
maon@tau.ac.il

Mooly Sagiv*
Tel Aviv University
msagiv@tau.ac.il

Eran Yahav
IBM T.J. Watson
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. We limit our attention to cutpoint-free programs—programs in which reasoning on a procedure call only requires consideration of context reachable from the actual parameters. For such programs, we show that our framework is able to perform an efficient modular analysis. 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. We have implemented a prototype of our framework and used it to verify interesting properties of cutpoint-free programs, including partial correctness of a recursive quicksort implementation.

*This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No 304/03).

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. Handling the heap in a precise manner requires strong pointer updates [6]. However, performing strong pointer updates requires flow-sensitive context-sensitive analysis and expensive heap abstractions that may be doubly-exponential in the program size [36]. The presence of procedures escalates the problem because of interactions between the program stack and the heap [34] and because recursive calls may introduce exponential factors in the analysis. This makes interprocedural shape analysis a challenging problem.

This paper introduces a new approach for shape analysis for a class of imperative programs. The main idea is to restrict the "sharing patterns" occurring in procedure calls. This allows procedures to be analyzed ignoring the part of the heap not reachable from actual parameters. Moreover, shape analysis can conservatively detect violations of the above restrictions, thus allowing to treat existing programs. A prototype of this approach was implemented and used to verify properties that could not be automatically verified before, including the partial correctness of a recursive quicksort [16] implementation (i.e., show that it returns an ordered permutation of its input).

Our restriction on programs is inspired by [33]. There, Rinetzky et. al. present a non-standard semantics for arbitrary programs in which procedures operate on local heaps containing only the objects reachable from actual parameters. The most complex aspect of [33] 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 local heap 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* [33]. We refer to an invocation in which no such cutpoint object exists as a *cutpoint-free invocation*. We refer to an execution of a program in which all invocations are cutpoint-free as a *cutpoint-free execution*, and to a program in which all executions are cutpoint-free as a *cutpoint-free program*. (We define these notions more formally in the following sections).

While many programs are not cutpoint-free, we observe that a reasonable number of programs, including all examples used in [13, 19, 34] are cutpoint-free, as well as many of the programs in [12, 37]. One of the key observations in this paper, is that we can exploit cutpoint-freedom to construct an interprocedural shape analysis algorithm that efficiently reuses procedure summaries.

In this paper, we present $\mathcal{LCPF}$, an operational semantics that efficiently handles cutpoint-free programs. This semantics is interesting because procedures operate on local heaps, thus supporting the notion of heap-modularity while permitting the usage of a global heap and destructive updates. Moreover, the absence of cutpoints drastically simplifies the meaning of procedure calls. $\mathcal{LCPF}$ checks that a program execution is indeed cutpoint-free and halts otherwise. As a result, it is applicable to any arbitrary program, and does not require an a priori classification of a program as cutpoint-free. We show that for cutpoint-free programs, $\mathcal{LCPF}$ is observationally equivalent to the standard global-heap semantics.

$\mathcal{LCPF}$ gives rise to an efficient interprocedural shape-analysis for cutpoint-free programs. Our interprocedural shape-analysis is a functional interprocedural analysis [2, 10, 11, 19, 20, 29, 38]. It tabulates abstractions of memory states before and after procedure calls. However, memory states are represented in a non-standard way *ignor-*

ing 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 on the heap from calling contexts. Indeed, this makes the analysis modular in the heap and thus allows reusing the summarized 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 total number of local variables [34].

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [23, 36]. 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, a combination of theorems in Appendix B and [36] guarantees that every instance of our *interprocedural* framework is sound (see Section 3).

This paper also provides an initial empirical evaluation of our algorithm. Our empirical evaluation indicates that the analysis is precise enough to prove properties such as the absence of null dereferences, preservation of data structure invariants such as list-ness, tree-ness, and sorted-ness for iterative and recursive programs with deep references into the heap and destructive updates. We observe that the cost of analyzing recursive procedures is comparable to the cost of analyzing their iterative counterparts. Moreover, the cost of analyzing a program with procedures is smaller than the cost of analyzing the same program with procedure bodies inlined.

1.1 Main Results

The contributions of this paper can be summarized as follows:

1. We define the notion of cutpoint-free programs, in which reasoning about a procedure allows ignoring the context not reachable from its actual parameters.
2. We show that interesting cutpoint-free programs can be written naturally, e.g., programs manipulating unshared trees and a recursive implementation of quicksort. We also show that some interesting existing programs are cutpoint-free, e.g., all programs verified using shape analysis in [13, 19, 34], and many of those in [12, 37].
3. We define an operational semantics for arbitrary Java-like programs that verifies that a program execution is cutpoint free. In this semantics, procedures operate on local heaps, thus supporting the notion of heap-modularity while permitting the usage of a global heap and destructive updates.
4. We present an interprocedural shape analysis for cutpoint-free programs. Our analysis is modular in the heap and thus allows reusing the effect of a procedure at different calling contexts and at different call-sites. Our analysis goes beyond the limits of existing approaches and was used to verify a recursive quicksort implementation.
5. We implemented a prototype of our approach. Preliminary experimental results indicate that: (i) the cost of analyzing recursive procedures is similar to the cost of analyzing their iterative versions; (ii) our analysis benefits from procedural abstraction; (iii) our approach compares favorably with [19, 34].

```
public class List{
   List n = null;
   int data;
   public List(int d){
      this.data = d;
   }
   static public List create3(int k) {
      List t1 = new List(k), t2 = new List(k+1), t3 = new List(k+2);
      t1.n = t2; t2.n = t3;
      return t1;
   }
   public static List splice(List p, List q) {
      List w = q;
      if (p != null) {
         List pn = p.n;
         p.n = null;
         p.n = splice(q, pn);
         w = p;
      }
      return w;
   }
   public static void main(String[] argv) {
      List x = create3(1), y = create3(4), z = create3(7);
      List t = splice(x, y);
      List s = splice(y, z);
   }
}
```

Figure 1: A Java program recursively splicing three singly-linked lists using destructive updates.

1.2 Motivating Example

Figure 1 shows a simple Java program that splices three unshared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. This program serves as a running example in this paper.

For each invocation of `splice`, our analyzer verifies that the returned list is acyclic and not heap-shared;[1] that the first parameter is aliased with the returned reference; and that the second parameter points to the second element in the returned list.

For this example, our algorithm effectively reuses procedure summaries, and only analyzes `splice(p,q)` once for every possible abstract input. As shown in Section 3.3, this means that `splice(p,q)` will be only analyzed a total number of 9 times. This should be contrasted with [34], in which no summaries are computed, and the procedure is analyzed 66 times. Compared to [19], our algorithm can summarize procedures in a more compact way (see Section 5).

1.3 Local heaps, Relevant Objects, Cutpoints, and Cutpoint-freedom

In our semantics, procedures operate on local heaps. The local heap contains only the part of the program's heap accessible to the procedure. Thus, procedures are invoked on local heaps containing only objects reachable from actual parameters. We refer to these objects as the *relevant* objects for the invocation.

> **Example 1.1** Figure 2 shows the concrete memory states that occur at the call `t=splice(x,y)`. S_2^c shows the state at the point of the call, and S_2^e

[1] An object is heap-shared if it is pointed-to by a field of more than one object.

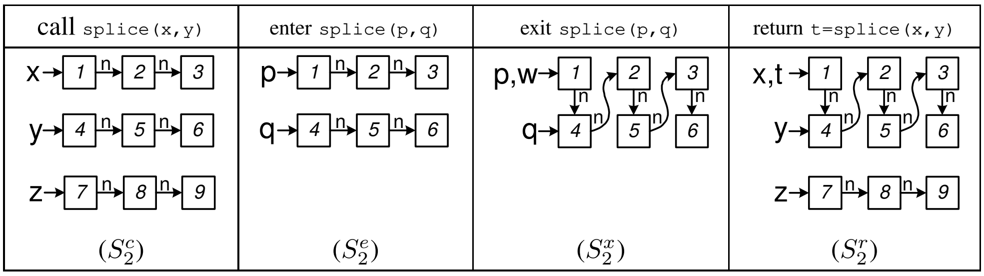
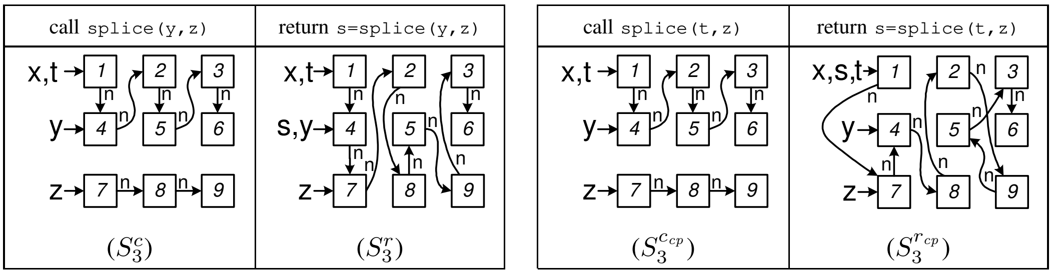


Figure 2: Concrete states for the invocation `t = splice(x, y)` in the running example.



(a) (b)

Figure 3: Concrete states for: (a) the invocation `s=splice(y,z)` in the program of Figure 1; (b) a variant of this program with an invocation `s=splice(t,z)`.

shows the state on entry to `splice`. Here, `splice` is invoked on local heap containing the (relevant) objects reachable from either `x` or `y`.

The fact that the local heap of the invocation `t=splice(x,y)` contains only the lists referenced by `x` and `y`, guarantees that destructive updates performed by `splice` can only affect access paths that pass through an object referenced by either `x` or `y`. Similarly, the invocation `s=splice(y,z)` in the concrete memory state S_3^c, shown in Figure 3(a), can only affect access paths that pass through an object referenced by either `y` or `z`.

Obviously, this is not always the case. For example, consider a variant of the example program in which the second call `s=splice(y,z)` is replaced by a call `s=splice(t,z)`. $S_3^{c_{cp}}$ and $S_3^{r_{cp}}$, depicted in Figure 3(b), show the concrete states when `s=splice(t,z)` is invoked and when it returns, respectively. As shown in the figure, the destructive updates of the `splice` procedure change not only paths from `t` and `z`, but also change the access paths from `y`.

A *cutpoint* for an invocation is an object which is: (i) reachable from an actual parameter, (ii) not pointed-to by an actual parameter, and (iii) reachable without going through an object which is pointed-to by an actual parameter (that is, it is either pointed-to by a variable or by an object not reachable from the parameters). In other words, a cutpoint is a relevant object that separates the part of the heap which is passed to the callee from the rest of the heap, but which is not pointed-to by a parameter. The object pointed-to by `y` at the call `s=splice(t,z)` (Figure 3(b)) is a *cutpoint*, and this invocation is not *cutpoint-free*. In contrast, the call `t=splice(x,y)` (Figure 2) does not have any cutpoints and is therefore *cutpoint-free*. In fact, all invocations in the program of Figure 1, including recursive ones, are cutpoint-free, and the program is a cutpoint-free program.

Our analyzer verifies that the running example is a cutpoint-free program. It also detects that in the variant of our running example, the call `s=splice(t,z)` is not a cutpoint-free invocation.

1.4 Outline

The rest of the paper is organized as follows. Section 2 defines our local heap concrete semantics. Section 3 conservatively abstracts this semantics, providing a heap-modular interprocedural shape analysis algorithm. Section 4 describes our implementation and experimental results. Section 5 describes related work, and Section 6 concludes.

2 Concrete Semantics

In this section, we present $\mathcal{LCPF}$, a large-step concrete semantics that serves as the basis for our abstraction. In $\mathcal{LCPF}$, an invoked procedure is passed only relevant objects. $\mathcal{LCPF}$ has two novel aspects: (i) it verifies that the execution is cutpoint-free; (ii) it has a *simple* rule for procedure calls that exploits (the verified) cutpoint-freedom. Nevertheless, in Appendix B, we show that for cutpoint-free programs $\mathcal{LCPF}$ is observationally equivalent to a standard store-based global-heap semantics. For simplicity, $\mathcal{LCPF}$ only keeps track of pointer-valued variables and fields.

2.1 Concrete Memory States

We represent memory states using 2-valued logical structures. 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.
- ι^S is an interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota^S(p) : {U^S}^k \to \{0, 1\}$. Predicates correspond to tracked properties of heap-allocated objects.

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

In the rest of the paper, we assume to be working with a fixed arbitrary program P. The program P consists of a collection of types, denoted by $\mathit{TypeId}^\star$. The set of all reference fields defined in P is denoted by $\mathit{FieldId}^\star$. For a procedure p, V_p denotes the set of its local reference variables, including its formal parameters. The set of all the local (reference) variables in P is denoted by $\mathit{Local}^\star$. For simplicity, we assume formal parameters are not assigned and that p always returns a value using a designated variable $ret_p \in V_p$. For example, $ret_{\mathtt{splice}} = \mathtt{w}$.

Table 1 shows the core predicates used in this paper. A unary predicate $T(v)$ holds for heap-allocated objects of type $\mathtt{T} \in \mathit{TypeId}^\star$. A binary predicate $f(v_1, v_2)$ holds when the $\mathtt{f} \in \mathit{FieldId}^\star$ field of v_1 points-to v_2. The designated binary predicate $eq(v_1, v_2)$ is the equality predicate recording equality between v_1 and v_2. A unary predicate $x(v)$ holds for an object that is pointed-to by the reference variable $\mathtt{x} \in \mathit{Local}^\star$ of the *current* procedure.[2] The role of the predicates $inUc$ and $inUx$ is explained in Section 2.2.

[2] For simplicity, we use the same set of predicates for all procedures. Thus, our semantics ensures that $\iota^S(x) = \lambda u.0$ for every local variable x that does not belong to the current call.

Table 1: Predicates used in the concrete semantics.

Predicate	**Intended Meaning**
$T(v)$	v is an object of type `T`
$f(v_1, v_2)$	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
$x(v)$	reference variable `x` points to the object v
$inUc(v)$	v originates from the caller's memory state at the call site
$inUx(v)$	v originated from the callee's memory state at the exit site

2-valued logical structures are depicted as directed graphs. We draw individuals as boxes. We depict the value of a pointer variable `x` by drawing an edge from `x` to the individual that represent the object that `x` 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. A directed edge between nodes u_1 and u_2 that is labeled with a binary predicate symbol p indicates that $\iota^S(p)(u_1, u_2) = 1$. For clarity, we do not draw the unary *List* predicate, and the binary equality predicate eq.

> **Example 2.1** The structure S_2^c of Figure 2 shows a *2-valued* logical structure that represents the memory state of the program at the call `t=splice(x, y)`. The depicted numerical values are only shown for presentation reasons, and have no meaning in the logical representation.

2.1.1 Admissible Memory States

Not all *2-valued* logical structures represent memory states 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 [33] and to the notion of structures that are *compatible with hygiene conditions* in [36]. We note that $\mathcal{LCPF}$ preserves states admissibility.

Definition 2.2 (Admissible 2-Valued Logical Structures) *A* 2-valued *logical structure $S = \langle U, \iota \rangle$ representing a local-heap for a procedure p at a given point in an execution is **admissible** iff*

(i) Only the local variables of the current call are represented, i.e., $\iota^S(x) \stackrel{\text{def}}{=} \lambda u.0$ for every $x \in Local^\star \setminus V_p$.

(ii) An object has at most one type, i.e., for every individual $u \in U$ there is exactly one type $T \in TypeId^\star$ such that $\iota^S(T)(u) = 1$.

(iii) A variable points-to at most one node, i.e., for every $x \in V_p$, there exists at most one individual such that $\iota^S(x)(u) = 1$.

(iv) A field is a partial function, i.e., for every individual u_1 and every field $f \in FieldId^\star$, there exists at most one individual $u_2 \in U$ such that $\iota^S(f)(u_1, u_2) = 1$.

(v) The predicate eq record equality, i.e., for every $u_1, u_2 \in U$, $\iota^S(eq)(u_1, u_2) = 1$ iff $u_1 = u_2$.

Table 2: Formulae shorthands and their intended meaning.

Shorthand	Formula	Intended Meaning
$F(v_1, v_2)$	$\bigvee_{f \in FieldId^*_P} f(v_1, v_2)$	v_1 has a field that points to v_2
$\varphi^*(v_1, v_2)$	$eq(v_1, v_2) \vee$ $(TC\ w_1, w_2 : \varphi(w_1, w_2))(v_1, v_2)$	the reflexive transitive closure of φ
$R_{\{x_1,\ldots,x_k\}}(v)$	$\bigvee_{x \in \{x_1,\ldots,x_k\}} \exists v_1 : x(v_1) \wedge F^*(v_1, v)$	v is reachable from $\mathtt{x}_1$ or ... or $\mathtt{x}_k$
$isCP_{q,\{x_1,\ldots,x_k\}}(v)$	$R_{\{x_1,\ldots,x_k\}}(v) \wedge$ $(\neg x_1(v) \wedge \ldots \wedge \neg x_k(v)) \wedge$ $(\bigvee_{y \in V_q} y(v) \vee$ $\exists v_1 : \neg R_{\{x_1,\ldots,x_k\}}(v_1) \wedge F(v_1, v))$	v is a cutpoint

2.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{lcpf}{\rightsquigarrow} \subseteq (\mathit{2Struct} \times st) \times \mathit{2Struct}$ that specifies how a statement st transforms an incoming logical structure into an outgoing logical structure. For assignments, 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 [36]. The inference rules for assignments are rather straightforward and can be found in Appendix A. For control statements, we use the standard rules of natural semantics, e.g., see [26].

Our treatment of procedure call and return could be briefly described as follows: (i) the call rule is applied, first checking that the invocation is cutpoint-free (by evaluating the side condition), and (ii) proceeding to construct the memory state at the callee's entry site (S_e) if the side condition holds; (iii) the caller's memory state at the call site (S_c) and the callee's memory state at the exit site (S_x) are used to construct the caller's memory state at the return site (S_r). We now formally define and explain these steps.

Figure 4 specifies the procedure call rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. The rule is instantiated for each call statement in the program.

2.2.1 Verifying Cutpoint-freedom.

The semantics uses the side condition of the procedure call rule to ensure that the execution is cutpoint-free. The side condition asserts that no object is a cutpoint. This is achieved by verifying that the formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, defined in Table 2, does not hold for any object at S_c, the memory state that arises when $p(x_1, \ldots, x_k)$ is invoked by q.

The formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, holding when v is a cutpoint object, is comprised of three conjuncts. The first conjunct, requires that v be reachable from an actual parameter. The second conjunct, requires that v not be pointed-to by an actual parameter. The third conjunct, requires that v be an entry point into p's local heap, i.e., is pointed-to by a local variable of q (the caller procedure) or by a field of an object not passed to p.

> **Example 2.3** The structure S^c_2 of Figure 2 depicts the memory state at the point of the call `t = splice(x, y)`. In this state, the formula $isCP_{main,\{x,y\}}(v)$ does not hold for any object. On the other hand, when `s = splice(t, z)` is invoked at $S_3^{c_{cp}}$ of Figure 3(b), the object pointed-

$$\frac{\langle \textit{body of } p, S_e\rangle \overset{lcpf}{\rightsquigarrow} S_x}{\langle y = p(x_1,\ldots,x_k), S_c\rangle \overset{lcpf}{\rightsquigarrow} S_r} \quad S_c \models \forall v\colon \neg isCP_{q,\{x_1,\ldots,x_k\}}(v)$$

where

$$\begin{array}{l}
S_e = \langle U_e, \iota_e\rangle \text{ where} \\
\quad U_e = \{u \in U^{S_c} \mid S_c \models R_{\{x_1,\ldots,x_k\}}(u)\} \\
\quad \iota_e = updCall_q^{y=p(x_1,\ldots,x_k)}(S_c) \\
S_r = \langle U_r, \iota_r\rangle \text{ where} \\
\quad Let\ \ U' = \{u.c \mid u \in U_c\} \cup \{u.x \mid u \in U_x\} \\
\qquad \iota' = \lambda p \in \mathcal{P}. \begin{cases} \iota_c[inUc \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : \quad u_1 = w_1.c,\ldots,u_m = w_m.c \\ \iota_x[inUx \mapsto \lambda v.1](p)(u_1,\ldots,u_m) & : \quad u_1 = w_1.x,\ldots,u_m = w_m.x \\ 0 & : \quad otherwise \end{cases} \\
\quad in\ \ U_r = \{u \in U' \mid \langle U', \iota'\rangle \not\models inUc(u) \wedge R_{\{x_1,\ldots,x_k\}}(u)\} \\
\qquad \iota_r = updRet_q^{y=p(x_1,\ldots,x_k)}(\langle U', \iota'\rangle)
\end{array}$$

Figure 4: The inference rule for a procedure call $y = p(x_1,\ldots,x_k)$ by a procedure q. The functions $updCall_q^{y=p(x_1,\ldots,x_k)}$ and $updRet_q^{y=p(x_1,\ldots,x_k)}$ are defined in Figure 5.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} x_i(v) & : \quad z = h_i \\ 0 & : \quad z \in Local^\star \setminus \{h_1,\ldots,h_k\} \end{cases}$$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$

$$z'(v) = \begin{cases} ret_p(v) & : \quad z = y \\ inUc(v) \wedge z(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee & : \quad z \in V_q \setminus \{y\} \\ \quad \exists v_1 : z(v_1) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) & \\ 0 & : \quad z \in Local^\star \setminus V_q \end{cases}$$

$$\begin{array}{l}
f'(v_1,v_2) = inUx(v_1) \wedge inUx(v_2) \wedge f(v_1,v_2) \vee \\
\qquad inUc(v_1) \wedge inUc(v_2) \wedge f(v_1,v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\
\qquad\quad inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} : f(v_1, v_{sep}) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{sep}, v_2) \\
inUc'(v) = inUx'(v) = 0
\end{array}$$

Figure 5: Predicate-update formulae for the core predicates used in the procedure call rule. We assume that the p's formal parameters are $h_1,\ldots,h_k$. There is a separate update formula for every local variable $\texttt{z} \in Local^\star$ and for every field $\texttt{f} \in FieldId^\star$.

to by `y` is a cutpoint. Note, that the formula $isCP_{main,\{t,z\}}(v)$ evaluates to 1 when v is bound to this object: the formula $R_{\{t,z\}}(v)$ holds for every object in `t`'s list. In particular, it holds for the second object which is pointed-to by a local variable (`y`) but not by an actual parameter (`t`, `z`).

Note that $\mathcal{LCPF}$ considers only the values of variables that belong to the current call when it detects cutpoints. This is possible because all pending calls are cutpoint-free. This greatly simplifies the cutpoint detection compared to [33].

2.2.2 Computing The Memory State at the Entry Site.

S_e, the memory state at the entry site to p, represents the local heap passed to p. It contains only these individuals in S_c that represent objects that are relevant for the invocation. The formal parameters are initialized by $updCall_q^{y=p(x_1,\ldots,x_k)}$, defined in Figure 5(a). The latter, specifies the value of the predicates in S_e using a predicate-update formulae evaluated over S_c. We use the convention that the updated value of x is denoted by x'. Predicates whose update formula is not specified, are assumed to be unchanged, i.e., $x'(v_1,\ldots) = x(v_1,\ldots)$. Note that only the predicates that represent variable values are modified. In particular, field values, represented by binary predicates, remain in p's local heap as in S_c.

> **Example 2.4** The structure S_2^e of Figure 2 depicts the memory state at the entry site to `splice` when `t = splice(x, y)` is invoked at the memory state S_2^c. Note that the list referenced by `z` is not passed to `splice`. Also note that the element which was referenced by `x` is now referenced by `p`. This is the result of applying the update formula $p'(v) = x(v)$ for the predicate p in this call. Similarly, the element which was referenced by `y` is now referenced by `q`.

2.2.3 Computing The Memory State at the Return Site.

The memory state at the return-site (S_r) is constructed as a combination of the memory state in which p was invoked (S_c) and the memory state at p's exit-site (S_x). Informally, S_c provides the information about the (unmodified) irrelevant objects and S_x contributes the information about the destructive updates and allocations made during the invocation.

The main challenge in computing the effect of a procedure is relating the objects at the call-site to the corresponding objects at the return site. The fact that the invocation is cutpoint-free guarantees that the only references into the local heap are references to objects referenced by an actual parameter. This allows us to reflect the effect of p into the local heap of q by: (i) replacing the relevant objects in S_c with S_x, the local heap at the exit from p; (ii) redirecting all references to an object referenced by an actual parameter to the object referenced by the corresponding formal parameter in S_x.

Technically, S_c and S_x are *combined* into an intermediate structure $\langle U', \iota' \rangle$. The latter contains a copy of the memory states at the call site and at the exit site. To distinguish between the copies, the auxiliary predicates $inUc$ and $inUx$ are set to hold for individuals that originate from S_c and S_x, respectively. Pointer redirection is specified by means of predicate update formulae, as defined in Figure 5(b). The most interesting aspect of these update-formulae is the formula $match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}$, defined below:

$$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1,v_2) \stackrel{\text{def}}{=} \bigvee_{i=1}^{k} inUc(v_1) \land x_i(v_1) \land inUx(v_2) \land h_i(v_2)$$

This formula matches an individual that represents an object which is referenced by an actual parameter at the call-site, with the individual that represents the object which is referenced by the corresponding formal parameter at the exit-site. Our assumption that formal parameters are not modified allows us to match these two individuals as representing the same object. Once pointer redirection is complete, all individuals

Table 3: The instrumentation predicates used in this paper.

Predicate	Intended Meaning	Defining Formula
$r_{obj}(v_1, v_2)$	v_2 is reachable from v_1 by some field path	$F^*(v_1, v_2)$
$ils(v)$	v is *locally* shared. i.e., v is pointed-to by a field of more than one object in the *local-heap*	$\exists v_1, v_2 : \neg eq(v_1, v_2) \land F(v_1, v) \land F(v_2, v)$
$c(v)$	v resides on a directed cycle of fields	$\exists v_1 : F(v, v_1) \land F^*(v_1, v)$
$r_x(v)$	v is reachable from variable `x`	$\exists v_x : x(v_x) \land F^*(v_x, v)$

originating from S_c and representing relevant objects are removed, resulting with the updated memory state of the caller.

Note that while the combined structure may not be an admissible memory state, the resulting memory state at the return site is admissible.

> **Example 2.5** S_2^c and S_2^x, shown in Figure 2, represent the memory states at the call-site and at the exit-site of the invocation `t=splice(x,y)`, respectively. Their combination according to the procedure call rule is S_2^r, which represents the memory state at the return site. Note that the lists of `x` and `y` from the call-site were replaced by the lists referenced by `p` and `q`. The list referenced by `z` was taken as is from the call-site.

3 Abstract Semantics

In this section, we present $\mathcal{LCPF}^{\#}$, a conservative abstract semantics abstracting $\mathcal{LCPF}$.

3.1 Abstract Memory States

We conservatively represent multiple concrete memory 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. The information partial order on the set $\{0, 1/2, 1\}$ is defined as $0 \sqsubseteq 1/2 \sqsupseteq 1$, and $0 \sqcup 1 = 1/2$.

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

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

The set of *3-valued* logical structures is denoted by *3Struct*.

Instrumentation Predicates Instrumentation predicates record derived properties of individuals, and are defined using a logical formula over core predicates. Instrumentation predicates are stored in the logical structures like core predicates. They are used to refine the abstract semantics, as we shall shortly see. Table 3 lists the instrumentation predicates used in this paper.

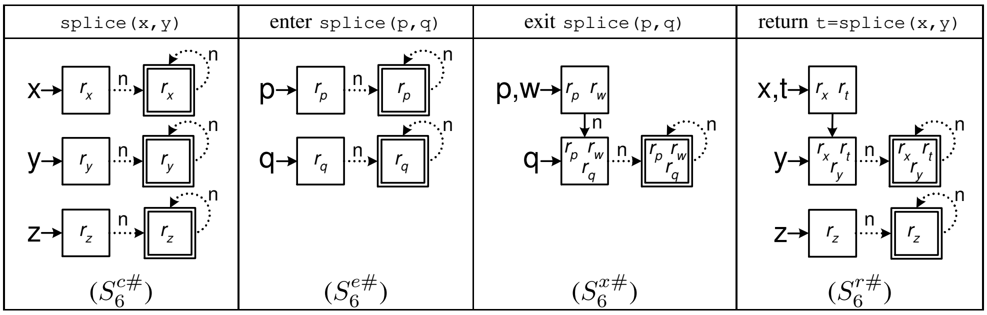


Figure 6: Abstract states for the invocation `t = splice(x, y);` in the running example.

Canonical Abstraction We now formally define how concrete memory states are represented using abstract memory states. The idea is that each individual from the (concrete) state is mapped into an individual in the abstract state. An abstract memory 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.

A *3-valued* logical structure $S^\sharp$ is a **canonical abstraction** of a *2-valued* logical structure S if there exists a surjective function $f\colon U^S \to U^{S^\sharp}$ satisfying the following conditions: (i) For all $u_1, u_2 \in U^S$, $f(u_1) = f(u_2)$ iff for all unary predicates $p \in \mathcal{P}$, $\iota^S(p)(u_1) = \iota^S(p)(u_2)$, and (ii) For all predicates $p \in \mathcal{P}$ of arity k and for all k-tuples $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

$$\iota^{S^\sharp}(p)(u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp) = \bigsqcup_{\substack{u_1,\ldots,u_k \in U^S \\ f(u_i)=u_i^\sharp}} \iota^S(p)(u_1, u_2, \ldots, u_k).$$

The set of concrete memory states such that $S^\sharp$ is their canonical abstraction is denoted by $\gamma(S^\sharp)$. Finally, we say that a node $u^\sharp \in U^{S^\sharp}$ **represents** node $u \in U$, when $f(u) = u^\sharp$. Note that *only* for a summary node u, $\iota^{S^\sharp}(eq)(u, u) = 1/2$.

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 (1/2) are drawn as dotted directed edges. Summary nodes are depicted by a double frame.

> **Example 3.1** Figure 6 shows the abstract states (as *3-valued* logical structures) representing the concrete states of Figure 2. Note that only the local variables `p` and `q` are represented inside the call to `splice(p,q)`. Representing only the local variables inside a call ensures that the number of unary predicates to be considered when analyzing the procedure is proportional to the number of its local variables. This reduces the overall complexity of our algorithm to be worst-case doubly-exponential in the maximal number of local variables rather than doubly-exponential in their total number (as in e.g., [34]).

The Importance of Reachability 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 [36]. In our framework, the predicates that record reachability from variables plays a central role. They enable us to identify the individuals representing objects that are reachable from actual parameters. For example, in the *3-valued* logical structure $S_6^{c\#}$ depicted in Figure 6, we can detect that the top two lists represent objects that are reachable from the actual parameters because either r_x or r_y holds for these individuals. None of these predicates holds for the individuals at the (irrelevant) list referenced by z. We believe that these predicates should be incorporated in any instance of our framework.

3.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{lcpf^{\#}}{\rightsquigarrow} \subseteq (\mathit{3Struct} \times st) \times \mathit{3Struct}$. Because our framework is based on [36], the specification of the concrete operational semantics for program statements (as transformers of 2-valued structures) in Section 2, also defines 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 particular, reinterpreting the side condition of the procedure call rule conservatively, verifies that the *program* is cutpoint free. In this paper, we directly utilize the implementation of these ideas available in TVLA [23].

In principle, the effect of a statement on the values of the instrumentation predicates can be evaluated using their defining formulae and the update formulae for the core predicates. In practice, this may lead to imprecise results in the analysis. It is far better to supply the update formula for the instrumentation predicates too. In this paper, we manually provide the update formulae of the instrumentation predicates (as done e.g., in [22, 34, 36]). Automatic derivation of update formulae for the instrumentation predicates [30] is currently not implemented in our framework. We note that update formulae are provided at the level of the programming language, and are thus applicable to arbitrary procedures and programs. Predicate update-formulae for the instrumentation predicates are provided in Appendix A.2.

The soundness of our abstract semantics is guaranteed by the combination of the theorems in Appendix B and [36]:

- In Appendix B, we show that for cutpoint-free programs $\mathcal{LCPF}$ is observationally equivalent to a standard store-based global-heap semantics.
- In [36], 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.

3.3 Interprocedural Functional Analysis via Tabulation of Abstract Local Heaps

Our algorithm computes procedure summaries by tabulating input abstract memory-states to output abstract memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The tabulated abstract memory-states represent local heaps, and are therefore independent of the context in which a procedure is invoked. As a result, the summary computed for a procedure could be used at different calling contexts and at different call-sites.

Our interprocedural analysis algorithm is a variant of the IFDS-framework [29] adapted to work with local-heaps. The main difference between our framework and [29]

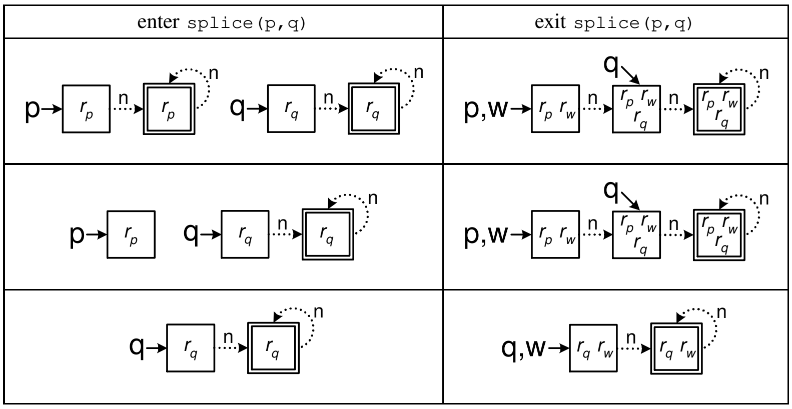


Figure 7: Partial tabulation of abstract states for the splice procedure.

is in the way return statements are handled: In [29], the dataflow facts that reach a return-site come either from the call-site (for information pertaining to local variables) or from the exit-site (for information pertaining to global variables). In our case, the information about the heap is obtained by *combining* pair-wise the abstract memory states at the call-site with their counterparts at the exit-site. A detailed description of our tabulation algorithm can be found in Appendix C.

Example 3.2 Figure 7 shows a partial tabulation of abstract local heaps for the `splice` procedure of the running example. The figure shows 3 possible input states of the list pointed-to by `p`. Identical possible input states of the list pointed-to by `q`, and their combinations are not shown. As mentioned in Section 1, the splice procedure is only analyzed 9 times before its tabulation is complete, producing a summary that is then reused whenever the effect of `splice(p, q)` is needed.

4 Prototype Implementation

We have implemented a prototype of our framework using TVLA [23]. The framework is parametric in the heap-abstraction and in the operational semantics. We have instantiated the framework to produce a shape-analysis algorithm for analyzing Java programs that manipulate (sorted) singly-linked lists and unshared trees. To translate Java programs we have extended an existing Soot-based [39] front-end for Java developed by R. Manevich.

The join operator in our framework can be either set-union or a more "aggressive" partial-join operation [24]. The former ensures that the analysis is fully-context sensitive. The latter exploits the fact that our abstract domain has a Hoare order and returns an upper approximation of the set-union operator. Our experiments were conducted with the partial-join operator.

Our analysis was able to verify that all the tested programs are cutpoint-free and *clean*, i.e., do not perform null-dereference and do not leak memory. For singly-linked-list-manipulating programs (Table 4.a), we also verified that the invoked procedures preserve list acyclicity. The analysis of the tree-manipulating programs (Table 4.b) verified that the tree invariants hold after the procedure terminates. For these programs

Iterative vs. Recursive Programs				
Implementation	**Iterative**		**Recursive**	
a. List manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a list	2.5	11.5	2.3	9.3
find searches an element in a list	3.2	23.7	3.6	37.1
insert inserts an element into a sorted list	5.1	50.1	5.4	46.8
delete removes an element from a sorted list	3.7	41.7	3.9	35.8
append appends two lists	3.7	18.4	3.9	22.5
reverse destructive list-reversal	3.6	26.9	3.4	21.0
revApp reverses a list by appending its head to its reversed tail	4.3	43.6	4.3	41.7
merge merges two sorted lists	12.5	585.1	5.4	87.1
splice splices two lists	4.9	76.5	4.8	33.6
running the running example	5.2	80.5	5.0	36.5
b. Tree manipulating programs	**Space**	**Time**	**Space**	**Time**
create creates a full tree	-	-	2.6	14.3
insert inserts a node	5.4	98.1	5.6	49.6
remove removes a node using `removeRoot` and `spliceLeft`	9.6	480.3	6.6	167.5
find finds a node with a given key	4.9	53.4	6.5	105.7
height returns the tree's height	-	-	5.4	76.1
spliceLeft a tree as the leftmost child of another tree	5.3	51.6	5.3	35.7
removeRoot removes the root of a tree	6.1	107.8	6.1	73.9
rotate rotates the left and right children of every node	-	-	4.9	57.1
c. Sorting programs	**Space**	**Time**	**Space**	**Time**
IinsertionSort moves the list elements into a sorted list	8.6	449.8	7.3	392.2
TailSort inserts the list head to its (recursively) sorted tail	4.9	101.6	4.9	103.4
QuickSort quicksorts a list	-	-	13.5	1017.1

d. [34] (Call String) vs. [19] (Relational) vs. our method						
Method	**Call String**		**Relational**		**Our method**	
Procedure	**Space**	**Time**	**Space**	**Time**	**Space**	**Time**
insert	1.8	20.8	6.3	122.9	3.5	20.0
delete	1.7	16.4	6.8	145.7	2.8	14.9
reverse	1.8	13.9	4.0	6.4	2.8	7.5
reverse8	2.7	123.8	9.1	14.8	2.8	21.7

e. Inline vs. Procedural Abstraction				
	Inline		**Proc. Call**	
Program	**Space**	**Time**	**Space**	**Time**
crt1x3	2.5	5.1	2.5	6.0
crt2x3	4.5	12.5	2.8	7.3
crt3x3	6.4	22.6	3.1	8.6
crt4x3	8.1	38.6	3.3	9.9
crt8x3	17.3	133.4	4.0	15.6

Table 4: Experimental results. Time is measured in seconds. Space is measured in megabytes. Experiments performed on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

we assume (and verify) that the trees are unshared. The analysis of the sorting programs (Table 4.c) verified that the sorting procedure returns a sorted permutation of its input list. To prove this property we adapted the abstraction used in [22]. We note that prior attempts to verify the partial correctness of `quicksort` using TVLA were not successful. For more details, see Appendix D

For two of our example programs (`quicksort` and `reverse8`), cutpoints were created as a result of objects pointed-to by a dead variable or a dead field at the point of a call. We manually rewrote these programs to eliminate these (false) cutpoints.

Table 4a-c compares the cost of analysis for iterative and recursive implementations of a given program.[3] 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 were of *client* programs and not a single procedure, i.e., in all tests, the program also allocates the data structure that it manipulates.

Table 4.d shows that our approach compares favorably with existing TVLA-based interprocedural shape analyzers [19, 34]. The experiments measure the cost of analyzing 4 recursive procedures that manipulate singly linked lists. For fair comparison with [33] and [18], we follow them and do not measure the cost of list allocation in these tests. All analyzers successfully verified that these (correct) procedures are clean and preserve list acyclicity. [19] was able to prove that `reverse` reverses the list and to pinpoint the location in the list that `delete` removed an element from. However, the cost of analysis for `insert` and `delete` in [19] was higher than the cost in [34] and in our analysis. Procedure `reverse8` reverses the same list 8 times. The cost of its analysis indicates that our approach, as well as [19], profits from being able to reuse the summary of `reverse`, while [34] cannot.

In addition, we examined whether our analysis benefits from reuse of procedure summaries. Table 4.e shows the cost of the analysis of programs that allocate several lists. Program **crtYx3** allocates Y lists. The table compares the cost of the analysis of programs that allocate a list by invoking `create3` (right column) to that of programs that inline `create3`'s body. The results are encouraging as they indicate (at least in these simple examples) that our analysis benefits from procedural abstraction.

5 Related Work

Interprocedural shape analysis has been studied in [7, 15, 19, 33, 34].

[34] explicitly represents the runtime stack and abstracts it as a linked-list. In this approach, the entire heap, and the runtime 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.*

[19] considers procedures as transformers from the (entire) heap before the call, to the (entire) heap after the call. Irrelevant objects are summarized into a single summary node. Relevant objects are summarized using a two-store vocabulary. One vocabulary records the current properties of the object. The other vocabulary encodes the properties that the object had when the procedure was invoked. The latter vocabulary allows to match objects at the call-site and at the exit-site. Note that this scheme never summarizes together objects that were not summarized together when the procedure was invoked. For cutpoint-free programs, these may lead to needlessly large summaries. Consider for example a procedure that operates on several lists and nondeterministically replaces elements between the list tails. The method of [19] will not summarize list elements that originated from different input lists. Thus, it will generate exponentially more mappings in the procedure summary, than the ones produced by our method.

[33] presents a heap-modular interprocedural shape-analysis for programs manipulating singly linked lists (without implementation). The algorithm explicitly records cutpoint objects in the local heap, and may become imprecise when there is more

[3] `revApp` is a recursive procedure. We analyzed it once with an iterative append procedure and once with a recursive append. Tail sort is a recursive procedure. We analyzed it once with an iterative insert procedure and once with a recursive insert.

than one cutpoint. Our algorithm can be seen as a specialization of [33] for handling cutpoint-free programs and as its generalization for handling trees and sorting programs. In addition, because we restricted our attention to cutpoint-free programs, our semantics and analysis are much simpler than the ones in [33].

[15] exploits a staged analysis to obtain a relatively scalable interprocedural shape analysis. This approach uses a scalable imprecise pointer-analysis to decompose the heap into a collection of independent locations. The precision of this approach might be limited as it relies on pointer-expressions appearing in the program's text. Its tabulation operates on global heaps, potentially leading to a low reuse of procedure summaries.

For the special case of singly-linked lists, another approach for modular shape analysis is presented in [7] without an implementation. The main idea there is to record for every object both its current properties and the properties it had at that time the procedure was invoked.

A heap modular interprocedural may-alias analysis is given in [12]. The key observation there is that a procedure operates uniformly on all aliasing relationships involving variables of pending calls. This method applies to programs with cutpoints. However, the lack of *must*-alias information may lead to a loss of precision in the analysis of destructive updates. For more details on the relation between [12] and local-heap shape analysis see [32, Sec. 5.1].

Local reasoning [18, 31] provides a way of proving properties of a procedure independent of its calling contexts by using the "frame rule". In some sense, the approach used in this paper is in the spirit of local reasoning. Our semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap. Local reasoning allows for an arbitrary partitioning of the heap based on user-supplied specifications. In contrast, in our work, the partitioning of the heap is built into the concrete semantics, and abstract interpretation is used to establish properties in the absence of user-supplied specifications.

Another relevant body of work is that concerning *encapsulation* (also known as *confinement* or *ownership*) [1, 3–5, 8, 9, 14, 17, 21, 25, 28]. These works allow modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of these works, as described in [27], is that they all place various restrictions on the sharing in the heap while pointers from the stack are generally left unrestricted. In our work, the semantics allows for arbitrary heap sharing within the same procedure, but restricts both the heap sharing and the stack sharing across procedure calls.

6 Conclusions and Future Work

In this paper, we presented an interprocedural shape analysis for cutpoint-free programs. Our analysis is modular in the heap and thus allows reusing the effect of a procedure at different calling contexts. In the future, we plan to utilize liveness analysis to automatically remove false cutpoints.

Acknowledgments. We are grateful for the helpful comments of N. Dor, S. Fink, T. Lev-Ami, R. Manevich, R. Shaham, G. Yorsh, and the anonymous referees of the SAS paper [35].

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference on Object-Oriented Programming (ESOP)*, 1997.

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

[3] A. Banerjee and D. A. Naumann. Representation independence, confinement, and access control. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2002.

[4] B. Bokowski and J. Vitek. Confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.

[5] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2003.

[6] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1990.

[7] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *International Static Analysis Symposium (SAS)*, 2003.

[8] D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming (ESOP)*, 2001.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1998.

[10] 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.

[11] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2002.

[12] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1994.

[13] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *International Static Analysis Symposium (SAS)*, 2000.

[14] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2001.

[15] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.

[16] C. A. R. Hoare. Algorithm 64: Quicksort. *Comm. of the ACM (CACM)*, 4(7):321, 1961.

[17] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1991.

[18] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2001.

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

[20] J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct. (CC)*, 1992.

[21] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 2002.

[22] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis (ISSTA)*, 2000.

[23] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *International Static Analysis Symposium (SAS)*, 2000. Available at http://www.math.tau.ac.il/∼ tvla.

[24] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *International Static Analysis Symposium (SAS)*, 2004.

[25] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

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

[27] J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *The First International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2003.

[28] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming (ESOP)*, 1998.

[29] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang. (POPL)*, 1995.

[30] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *European Symposium on Programming Languages (ESOP)*, 2003.

[31] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Symp. on Logic in Computer Science (LICS)*, 2002.

[32] 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*".

[33] 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. (POPL)*, 2005.

[34] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct. (CC)*, 2001.

[35] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *International Static Analysis Symposium (SAS)*, 2005.

[36] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst. (TOPLAS)*, 24(3):217–298, 2002.

[37] R. Shaham, E. Yahav, E.K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *International Static Analysis Symposium (SAS)*, 2003.

[38] 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.

[39] 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.

Statement	**Predicate-update formulae**	$\mathbf{side-condition}$
y = null	$y'(v) = 0$	
y = x	$y'(v) = x(v)$	
y = x.f	$y'(v) = \exists v_1 : x(v_1) \wedge f(v_1, v)$	$\exists v_1 : x(v_1)$
y.f = null	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$	$\exists v_1 : y(v_1)$
y.f = x	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$	$\exists v_1 : y(v_1)$
y = alloc(T)	$T'(v) = T'(v) \vee new(v)$	
	$eq'(v_1, v_2) = eq(v_1, v_2) \vee new(v_1) \wedge new(v_2)$	
	$new'(v) = 0$	

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

A Formal Specification of the Operational Semantics

This appendix provides the operational semantics for the intraprocedural statements (Section A.1) and the predicate-update formulae for the instrumentation predicates for interprocedural statements (Section A.2).

A.1 Operational Semantics for Atomic Statements

The operational semantics for assignments is specified by *predicate-update formulae*: for every predicate p and for every statement st, the value of p in the 2-valued structure 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 assignment is given in Figure 8. The table also specifies the side condition which enables that application of the statement. These conditions check that null-dereference is not performed. 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., $p'(v_1, \ldots) = p(v_1, \ldots)$.

None of the assignments, except for object allocation, modifies the underlying universe. Object allocation is handled as in [36]: A new individual is added to the universe to represent the allocated object; the auxiliary predicate new is set to hold *only* at that individual; only then, the predicate-update formulae is evaluated.

A.2 Predicate Update Formulae for Instrumentation Predicates

Figure 9 provides the update formulae for instrumentation predicates used by the procedure call rule. We use $PT_X(v)$ as a shorthand for $\bigvee_{x \in X} x(v)$. The intended meaning of this formula is to specify that v is pointed to by some variable from $X \subseteq Local^\star$. We use $bypass_X(v_1, v_2)$ as a shorthand for $(F(v_1, v_2) \wedge \neg PT_X(v_1))^*$. The intended meaning of this formula is to specify that v_2 is reachable from v_1 by a path that does not traverse any object which is pointed-to by any variable in $X \subseteq Local^\star$. As we can see, formula $match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_1, v_2)$ again plays a central role.

a. Predicate update formulae for $updCall_q^{y=p(x_1,\ldots,x_k)}$
$ils'(v) = ils'(v) \wedge (\neg PT_{x_1,\ldots,x_k}(v) \vee \exists v_1, v_2 \colon R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_1, v) \wedge F(v_2, v) \wedge \neg eq(v_1, v_2))$
$r'_y(v) = \begin{cases} r_{x_i}(v) & : \quad y = h_i \\ 0 & : \quad y \in Local^\star \setminus \{h_1, \ldots, h_k\} \end{cases}$

b. Predicate update formulae for $updRet_q^{y=p(x_1,\ldots,x_k)}$
$ils'(v) = ils(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v)) \vee PT_{x_1,\ldots,x_k}(v) \wedge \exists v_1, v_2, v_3 \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \wedge \neg eq(v_2, v_3) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_2, v_1) \wedge (inUc(v_3) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_3) \wedge F(v_3, v_1) \vee inUx(v_3) \wedge F(v_3, v))$
$r'_{obj}(v_1, v_2) = r_{obj}(v_1, v_2) \wedge inUx(v_1) \wedge inUx(v_2) \vee r_{obj}(v_1, v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee inUc(v_1) \wedge inUx(v_2) \wedge \exists v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge bypass_{\{x_1,\ldots,x_k\}}(v_1, v_a) \wedge r_{obj}(v_f, v_2)$
$r'_x(v) = inUc(v) \wedge r_x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v) \wedge \exists v_x, v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge x(v_x) \wedge bypass_{\{x_1,\ldots,x_k\}}(v_x, v_a) \wedge r_{obj}(v_f, v)$

Figure 9: The predicate update formulae for the instrumentation predicates used in the procedure call rule. 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 Semantics Equivalence

In this section, we define the $\mathcal{GSB}$ semantics, which is operational, large-step, store-based (as opposed to storeless), and global, i.e., the entire heap is passed to a procedure. We refer to this semantics as the standard semantics.

For simplicity, the semantics tracks only pointer values and assumes that every pointer-valued field or variable is assigned `null` before being assigned a new value.[4] In addition, we assume that before a procedure terminates it assigns a `null` value to every pointer variable that is not a formal parameter or the return argument.[5]

Figure 10 defines the semantic domains. *Loc* is an unbounded set of memory locations. A *memory state* for a procedure p, $\sigma_G^p \in \Sigma_G^p$, keeps track of the allocated memory locations, L, an environment mapping p's local variables to values, ρ, and a mapping from fields of *allocated* locations to values, h. Due to our simplifying assumptions, a value is either a memory location or *null*.

The meaning of statements is described by a transition relation $\overset{G}{\rightsquigarrow} \subseteq (\sigma_G \times stms) \times \sigma_G$.

[4]Special care needs to be taken when handling statements in which the same variable appears both in the left-side of the assignment and in its right-side, e.g., `x = x.f`. Such statements require additional source-to-source transformations and the introduction of temporary variables.

[5]These conventions were chosen as a matter of simplicity; in principle, different ones could be used with minor effects on the capabilities of our approach.

$$
\begin{array}{lcl}
l & \in & Loc \\
v & \in & Val = Loc \cup \{null\} \\
\rho & \in & Env_p = V_p \to Val \\
h & \in & Heap_G = Loc \times FieldId \to Val \\
\sigma_G, \langle L, \rho, h \rangle & \in & \Sigma_G^p = 2^{Loc} \times Env_p \times Heap_G
\end{array}
$$

Figure 10: Semantic domains of the $\mathcal{GSB}$ semantics.

$$
\begin{array}{ll}
\langle \mathtt{x} = \mathtt{null}, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto null], h \rangle & \\
\langle \mathtt{x} = \mathtt{y}, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto \rho(y)], h \rangle & \\
\langle \mathtt{x} = \mathtt{y.f}, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto h(\rho(y), f)], h \rangle & \rho(y) \neq null \\
\langle \mathtt{x.f} = \mathtt{null}, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto null] \rangle & \rho(x) \neq null \\
\langle \mathtt{x.f} = \mathtt{y}, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto \rho(y)] \rangle & \rho(x) \neq null \\
\langle \mathtt{x} = \mathtt{alloc}\, t, \langle L, \rho, h \rangle \rangle \overset{G}{\rightsquigarrow} \langle L \cup \{l\}, \rho[x \mapsto l], h \cup I(l) \rangle & l \notin L
\end{array}
$$

Figure 11: Axioms for atomic statements in the $\mathcal{GSB}$ semantics. I initializes all pointer fields at l to *null*.

Figure 11 shows the *axioms* for assignments. The *inference rule* for function calls is given in Figure 12. All other statements are handled as usual using a two-level store semantics for pointer languages.

B.1 Observable Properties

In this section, we introduce access paths, which are the only means by which a program can observe a state. Note that the program cannot observe location names.

Definition B.1 (Field Paths) *A **field path** $\delta \in \Delta = FieldId^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ.*

$$
\frac{\langle \text{body of } p, \langle L_e, \rho_e, h_e \rangle \rangle \overset{G}{\rightsquigarrow} \langle L_x, \rho_x, h_x \rangle}{\langle y = p(x_1, \ldots, x_k), \langle L_c, \rho_c, h_c \rangle \rangle \overset{G}{\rightsquigarrow} \langle L_r, \rho_r, h_r \rangle}
$$

$$
\begin{array}{l}
where \\
L_e = L_c,\ \rho_e(v) = \begin{cases} \rho_c(x_i) & v = z_i \\ null & otherwise \end{cases},\ h_e = h_c \\
L_r = L_x,\ \rho_r = \rho_c[y \mapsto \rho_x(ret_p)],\ h_r = h_x
\end{array}
$$

Figure 12: Inference rule for function invocation in the $\mathcal{GSB}$ semantics, assuming the formal variables of p are $z_1, \ldots, z_k$ and that p's return value is a pointer.

Definition B.2 (Access path) *An **access path** $\alpha = \langle x, \delta \rangle \in V_p \times \Delta$ of a procedure p is a pair consisting of a local variable of p and a field path. $AccPath_p$ denotes the set of all access paths of procedure p. $AccPath$ denotes the union of all access paths of all procedures in a program.*

B.2 Observable Properties in $\mathcal{GSB}$

Definition B.3 (Access path value) *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\langle L, \rho, h \rangle$, denoted by $[\![\alpha]\!]_G \langle L, \rho, h \rangle$, is defined to be $\hat{h}(\rho(x), \delta)$, where*

$$\hat{h} \colon \text{Val} \times \Delta \to \text{Val} \textit{ such that}$$
$$\hat{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \\ \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in \text{Loc} \\ \textit{null} & \text{otherwise} \end{cases}$$

Note that the value of an access path that traverses a *null*-valued field is defined to be *null*. This definition simplifies the notion of equivalence between the $\mathcal{GSB}$ semantics and $\mathcal{LSL}$, our new semantics. Alternatively, we could have defined the value of such a path to be $\perp$. The semantics given in Figure 11 checks that a null-dereference is not performed (see the side-conditions listed in the caption).

Definition B.4 (Access-path equality) *An access path α is **equal to null** in a given state σ_G, denoted by $[\![\alpha = \texttt{null}]\!]_G(\sigma_G)$, if $[\![\alpha]\!]_G(\sigma_G) = $ null. Access paths α and β are **equal** in σ_G, denoted by $[\![\alpha = \beta]\!]_G(\sigma_G)$, if they have the same value in that state, i.e., $[\![\alpha]\!]_G(\sigma_G) = [\![\beta]\!]_G(\sigma_G)$.*

B.3 Observable Properties in $\mathcal{LCPF}$

In this section, we define the value of access paths equality in $\mathcal{LCPF}$.

Definition B.5 (Reachability via a field path) *Let $u_1, u_2 \in U^S$ be individuals in memory state $S \in$ 2Struct, u_1 **reaches** u_2 **via** δ, if $\delta = \epsilon$ and $\iota^S(eq)(u_1, u_2) = 1$, or $\delta = f\delta'$ and there exists $u' \in U$ such that $\iota^S(f)(u_1, u') = 1$ and u' reaches u_2 via δ'.*

Definition B.6 (Access-path equality) *An access path $\alpha = \langle x, \delta \rangle$ is **equal to null**, denoted by $[\![\alpha = \texttt{null}]\!]_{\mathcal{LCPF}}(S)$, if there do not exist individuals $u_1, u \in U^S$ such that $\iota^S(x)(u_1) = 1$ and u_1 reaches u via δ. Access paths $\alpha_1 = \langle x_1, \delta_1 \rangle$ and $\alpha_2 = \langle x_2, \delta_2 \rangle$ are **equal** in a given memory state S, denoted by $[\![\alpha_1 = \alpha_2]\!]_{\mathcal{LCPF}}(S)$, if both are equal to null, or there exist individuals u_1, u_2 and u, such that $\iota^S(x_1)(u_1) = 1$, $\iota^S(x_2)(u_2) = 1$, u_1 reaches u via δ_1, and u_2 reaches u via δ_2.*

B.4 Observable Equivalence

Definition B.7 (Observational equivalence) *Let p be a method. The states $S \in$ 2Struct and $\sigma_G \in \Sigma_G^p$ are **observationally equivalent** if for all $\alpha, \beta, \gamma \in AccPath_p$,*

(i) $[\![\alpha = \beta]\!]_{\mathcal{LCPF}}(S) \Leftrightarrow [\![\alpha = \beta]\!]_G(\sigma_G)$, *and*

(ii) $[\![\gamma = \texttt{null}]\!]_{\mathcal{LCPF}}(S) \Leftrightarrow [\![\gamma = \texttt{null}]\!]_G(\sigma_G)$.

Theorem B.8 (Equivalence) *Let P be a cutpoint free program. Let p be a procedure in P. Let $S \in \mathit{2Struct}$ and $\sigma_G \in \Sigma_G$ be observationally equivalent states. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, S\rangle \overset{lcpf}{\rightsquigarrow} S' \iff \langle st, \sigma_G\rangle \overset{G}{\rightsquigarrow} \sigma'_G.$$

Furthermore, S' and σ'_G are observationally equivalent.

Sketch of Proof: The proof is done by induction on the shape of the derivation tree. We prove that observational equivalence is preserved by showing a stronger property: the two memory states are isomorphic up to garbage (i.e., elements not reachable from any program variable). We look at the two memory states as graphs. The graph nodes are the allocated objects and the graph edges are the object fields. The graph nodes may be labeled by variables.

We maintain an injective and a surjective function ρ from the set of objects that are reachable from the variables of the current procedure in the memory state of the $\mathcal{GSB}$ to the same set of objects in the memory state of the $\mathcal{LCPF}$ semantics. Clearly when a program starts, and prior to the allocation of any object, the two memory states are isomorphic. It is easy to verify that atomic statement preserves the isomorphism: ρ remains unchanged, except that object allocation maps the new location to the new individual.

When a procedure is invoked, the mapping ρ is projected on the set of objects passed to the invoked procedure. When a procedure returns, the mapping of locations resp. individuals that were irrelevant for the invocation remains as in the call site. The mapping for locations resp. individuals that were relevant for the invocation, as well as those that were allocated during the invocation, are taken from the exit site. Note that the induction assumption ensures that the above scheme is well defined and that the subgraph containing the objects that were (resp. were not) in the invoked procedure local heap are isomorphic. To see why the whole graphs are isomorphic we recall that P is cutpoint free. Thus, any object which was not passed to the invoked procedure which has a field that points to such an object, must point to an object which is pointed to by an actual parameter, and the $\mathcal{LCPF}$ semantics restores all outside references to these objects, which by the induction assumption, must be the same in both semantics. A similar reasoning applies to variables.

C Tabulation Algorithm

In this section, we describe the iterative interprocedural local-heap shape-analysis algorithm. For simplicity, we describe it in a generic way as an algorithm that manipulates shape graphs. In this paper, shape graphs are implemented by *3-valued* logical structures.

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 [26]). The algorithm remains sound in case the join operator is an over approximation of set union.

The algorithm is a variant of the IFDS-framework [29] adapted to work with local-heaps. The main difference between our framework and [29] is in the way return statements are handled: In [29], the dataflow facts that reach a return-site come either from the call-site (for information pertaining to local variables) or from the exit-site (for information to pertaining global 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 (see Appendix C.2).

C.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 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 $exit(n)$.

C.2 Tabulation Algorithm

We describe the algorithm using the following operations as "black boxes":

- *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.
- *applicable* : $Stmt \times SG \to \{\textbf{true}, \textbf{false}\}$ verifies that the given procedure call statement can be applied to the given shape graph. Thus, verifying that the invocation is cutpoint free.
- *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.
- *combine* : $Stmt \times SG \times SG \to 2^{SG}$ computes the shape-graph representing the local-heap of the caller at the return-site by applying the associated abstract

proc `tabulate(`$Program\ P$`,` $SG\ sg_0$`)`
  $worklist = \{\langle entry(main) : \langle sg_0, sg_0 \rangle \rangle\}$
  while $(worklist \neq \emptyset)$
    `remove an event` $\langle n : \langle sg_{entry}, sg \rangle \rangle$ `from` $worklist$
    if $n \in Entry^\star \cup Ret^\star \cup Intra^\star$ **then**
      foreach $\langle n, n' \rangle \in out(n)$
        foreach $sg' \in$ ***apply***$(stmt(\langle n, n' \rangle), sg)$
          if $\langle sg_{entry}, sg' \rangle \notin PathSet(n')$ **then**
            `propagate`$(n', \langle sg_{entry}, sg' \rangle)$
    else if $n \in Call^\star$
      n^{entry}_{callee} = `callee(`n`)`
      if **not** $applicable(stmt(\langle n, n^{entry}_{callee} \rangle), sg)$ **then** **HALT**
      foreach $sg' \in$ ***extract***$(stmt(\langle n, n^{entry}_{callee} \rangle), sg)$
        add $\langle n, \langle sg_{entry}, sg \rangle \rangle$ *to* $CTXs(n^{entry}_{callee})\ \ sg'$
        if $\langle sg', sg' \rangle \notin PathSet(n^{entry}_{callee})$ **then**
          `propagate`$(n^{entry}_{callee}, \langle sg', sg' \rangle)$
        else
          $n_{exit} = exit(n^{entry}_{callee})$
          foreach $sg_{exit} \in Summary(n^{entry}_{callee})\ \ sg'$
            `addToRet`$(n_{exit}, sg_{exit}, return(n), \langle sg_{entry}, sg \rangle)$
    else `//`$n \in Exit^\star$
      foreach$\langle n_{call}, \langle sg_e, sg_c \rangle \rangle \in CTXs(n^{entry}_{callee})\ \ sg_{entry}$
        `addToRet`$(\mathtt{n}, sg, return(n_{call}), \langle sg_{entry}, sg_{call} \rangle)$

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

Figure 13: The tabulation algorithm.

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.

In this paper, we implement *apply* by evaluating the abstract transformer associate with the given atomic statement (see Section A.1 and Section 3). The operations *applicable*, *extract*, and *combine* are implemented by evaluating the different steps in the procedure call rule instantiated for the given call: The operation *applicable* is implemented by evaluating the side condition on the abstract memory state that arises at the call site. The operations *extract* and *combine* are implemented by following the rule's specification on how to construct the (abstract) memory states at the callee's entry state and at the caller's return site, respectively (see Section 2 and Section 3).

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 [38] 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 [29].

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 13. 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. 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.

D Analyzing Sorting Programs

The analysis presented in [22] allows to prove partial correctness of sorting and list manipulating procedures. In this section, we briefly describe our adaptation of their abstraction to local-heaps.

Core Predicates		
predicate	**Intended Meaning**	
$dle(v_1, v_2)$	The `data` component of v_1 is less-than-or-equal-to the data component of v_2	
Instrumentation Predicates		
predicate	**Defining Formula**	**Intended Meaning**
$O(v)$	$\forall v_1 : n(v, v_1) \implies dle(v, v_1)$	v's data field is not strictly greater than that of its successor

Table 5: Core and instrumentation predicates used in the analysis of sorting programs.

The main idea is to track the relative order between the data components of list elements using a binary core relation (dle). In addition, an additional instrumentation predicate records the relative order between the data components of successive list elements (O). Table 5 lists the additional predicates we use here, together with their intended meaning.

Concrete memory states are represented using *2-valued* logical structures and we abstract them into *3-valued* logical structures using *canonical abstraction*. However, we represent two individuals that differ only in the value of the predicate O by a single summary individual. This improves the performance of the analysis, but may reduce its precision.

We use the same predicate-update formulae as in [22] to specify the effect of intraprocedural statements on the values of the added predicates. The main idea is to assume (and verify) that a list element is allocated with a random data field, which is not modified afterwards. The program can establish the relative relation between the data fields using comparisons.

The only change in the procedure call rule amounts to updating the rule to construct the memory state at the return site. Note that the values of both the O-predicate and the dle-predicate at the entry to the callee is the same as it was at the call-site.

At the return site, the value of the O-predicate can be taken as is from the call site (for objects not passed to the invoked procedure) and from the exit site (for objects in the invoked procedure local heap). This is possible because the invoked procedure could not have modified the external references into its local heap. In addition, if it also has not changed the data-value, the O-predicate still holds for those objects whose successor was passed to the callee.

The dle relation between objects that were (Resp. were not) passed to the callee does not change. The only difficulty is in restoring the dle relation between objects that were not passed to the invoked procedure and those that were. The main problem is that we cannot relate an individual that represents a certain heap allocated object at the call site, with the individual that represents the same object at the exit site. This information is lost in our *semantics* for all objects, except for the ones that are pointed to by parameters. Thus, we try to restore the relative relation using the parameters, resorting to an indefinite value as our last choice. The predicate update formulae for the dle-predicate is given in Figure 14.

We applied our analysis to verify the iterative and recursive sorting programs listed in Table 4c and in Table 6. Our analysis was able to verify that these programs are clean and preserve list acyclicity. Furthermore, it verified that:

(i) `find`, `last`, `insert`, and `delete` preserve list sorted-ness.

(ii) `merge` merges 2 sorted lists into a sorted list.

$$
\begin{aligned}
&dle'(v_1, v_2) = dle(v_1, v_2) \land (inUc(v_1) \land inUc(v_2) \lor inUx(v_1) \land inUx(v_2)) \lor \\
&\quad inUc(v_1) \land inUx(v_2) \land \\
&\quad\left(\begin{cases} 1 & : \exists v_c, v_x : match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \land dle(v_1, v_c) \land dle(v_x, v_2) \\ 0 & : \exists v_c, v_x : match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \land \neg dle(v_1, v_c) \land \neg dle(v_x, v_2) \\ 1/2 & : \text{otherwise} \end{cases} \right) \lor \\
&\quad inUx(v_1) \land inUc(v_2) \land \\
&\quad\left(\begin{cases} 1 & : \exists v_c, v_x : match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \land dle(v_1, v_c) \land dle(v_x, v_2) \\ 0 & : \exists v_c, v_x : match_{\{\langle h_1, x_1\rangle, \ldots, \langle h_k, x_k\rangle\}}(v_c, v_x) \land \neg dle(v_1, v_c) \land \neg dle(v_x, v_2) \\ 1/2 & : \text{otherwise} \end{cases} \right)
\end{aligned}
$$

Figure 14: The predicate update formulae for the dle predicate used in the procedure call rule to construct the memory state at the return site.

(iii) Reversing a sorted list by either `reverese` or `revApp`, results in a list in reversed order. To establish the last property, we needed an additional instrumentation predicate $RO(v)$, whose defining formula is $\forall v_1 : n(v, v_1) \implies dle(v_1, v)$. This predicate holds for an individual v, if v's data field is not strictly less than that of its successor [22].

(iv) `MaxFirst` returns the list element with the highest data value in the list, and that its value is *strictly* greater than that of any preceding element. Similar properties were verified for the procedures `MaxLast`, `MinFirst`, and `MinLast`.

We now describe the analysis of `quicksort` in more details. Figure 15 shows a program that allocates a random list and sorts it using the `quicksort` procedure. The quicksort procedure partitions the list by moving all the list elements whose data field is less than that of the pivot (the first parameter) to the beginning of the list. The list is then divided into two sub lists, which are sorted in a recursive fashion. Finally, the two lists are linked together. Note that the pivot is strictly larger than all other elements in the first list. This ensures that when the first recursive call returns, the pivot remains the last element in the first sublist.

We made two modifications to the program in order to eliminate two (false) cutpoints. The `n`-field of the pivot object (pointed-to by `p`) is dead when the first recursive call occurs. Had we not nullified it, the objects pointed-to by `tl` and `last` would have become cutpoints. Similarly, the `pr` is dead when the second recursive call occurs. However, the object it points to would have become a cutpoint had it not been nullified. The nullification of all the local variables prior to the return statement is only conducted to simplify the presentation.

Figure 16 shows several concrete states that occur during the execution of `quicksort` and their abstraction. For clarity, in the concrete states we do not draw the instrumentation predicates. Instead, we draw in each object the numeric value of its data field, from which the dle relation can be easily inferred. In the abstract states, we do not draw the dle and r_{obj} relations. We prove that a list is sorted by showing that the predicate O holds in all its elements. Because this predicate does not take a role in the abstraction and we only care when its value is 1, we draw it only in those nodes in which it hold. If we do not draw it in a node u, we mean that its value in u is either 0 or $1/2$. Because the dle is a transitive relation, we do not draw dle-labeled edges which can be inferred. For example, if the data field of u_1 is less-or-equal to the data field of u_2, and the data

Implementation	Iterative		Recursive	
Program	**Space**	**Time**	**Space**	**Time**
create creates a list	2.6	15.0	2.4	12.3
find searches an element in a list	4.1	42.2	4.6	60.7
insert inserts an element into a sorted list	4.7	67.7	4.9	69.5
delete removes an element from a sorted list	4.7	75.1	4.8	70.4
reverse destructively reverses a sorted list	4.5	54.8	4.4	47.6
revApp reverse a sorted list by appending its head to the reversed tail	4.8	74.8	4.8	70.5
merge merges two sorted lists	11.7	1115.1	5.8	131.2
last returns the last element in a sorted list	4.2	43.3	4.5	47.3
maxFirst returns the first maximal element in an unsorted list	3.6	71.3	3.5	51.6
maxLast returns the last maximal element in an unsorted list	3.6	83.5	3.5	59.0
minFirst returns the first minimal element in an unsorted list	3.5	64.1	3.5	51.1
minLast returns the last minimal element in an unsorted list	3.9	76.3	3.5	59.0

Table 6: Experimental results for additional sorting programs. Time is measured in seconds. Space is measured in megabytes. Experiments performed on a machine with a 1.5 Ghz Pentium M processor and 1 Gb memory.

field of the latter is less-or-equal to the data field of u_3, we draw a dle-labeled edge from u_1 to u_2 and from u_2 to u_3, but we do not draw such an edge from u_1 to u_3. Because the dle is reflexive, we omit self dle-loops in non summary objects.

Note that the second recursive call to, is invoked on a local heap which, under abstraction, is identical to the local heap that was passed by the external call, thus the results of analyzing the recursive call can be reused in the analysis of the external call. Also note that when the procedure returns, every element has the O property, thus, proving that the list is in order.

```
public static List quickSortRec(List p, List q) {
    // Location (A)
    List hd,pr,tl;

    if (p == null)
        return first;

    if (p == q)
        return h2;

    hd = p;
    pr = p;
    tl = p.n;

    // Partitioning of the list
    while (tl != q) {
        if (tl.d < p.d) {
            pr.n = tl.n;
            tl.n = hd;
            hd = tl;
            tl = pr.n;
        }
        else {
            pr = tl;
            tl = tl.n;
        }
    }

    tl = p.n;
    p.n = null;    // removing a false cutpoint due to a dead field
    pr = null;     // removing a false cutpoint due to a dead variable

    // Location (B)
    List s = quickSortRec(hd,p);
    // Location (C)
    List t = quickSortRec(tl,q);
    // Location (D)

    p.n = t;

    t = hd = pr = tl = null;

    // Location (E)
    return s;
}

public static void main(String argv[]) {
    List x = randomList(8);
    List y = quicksort(x,null);
}
```

Figure 15: A program that sorts a list using a quicksort algorithm for singly linked lists.

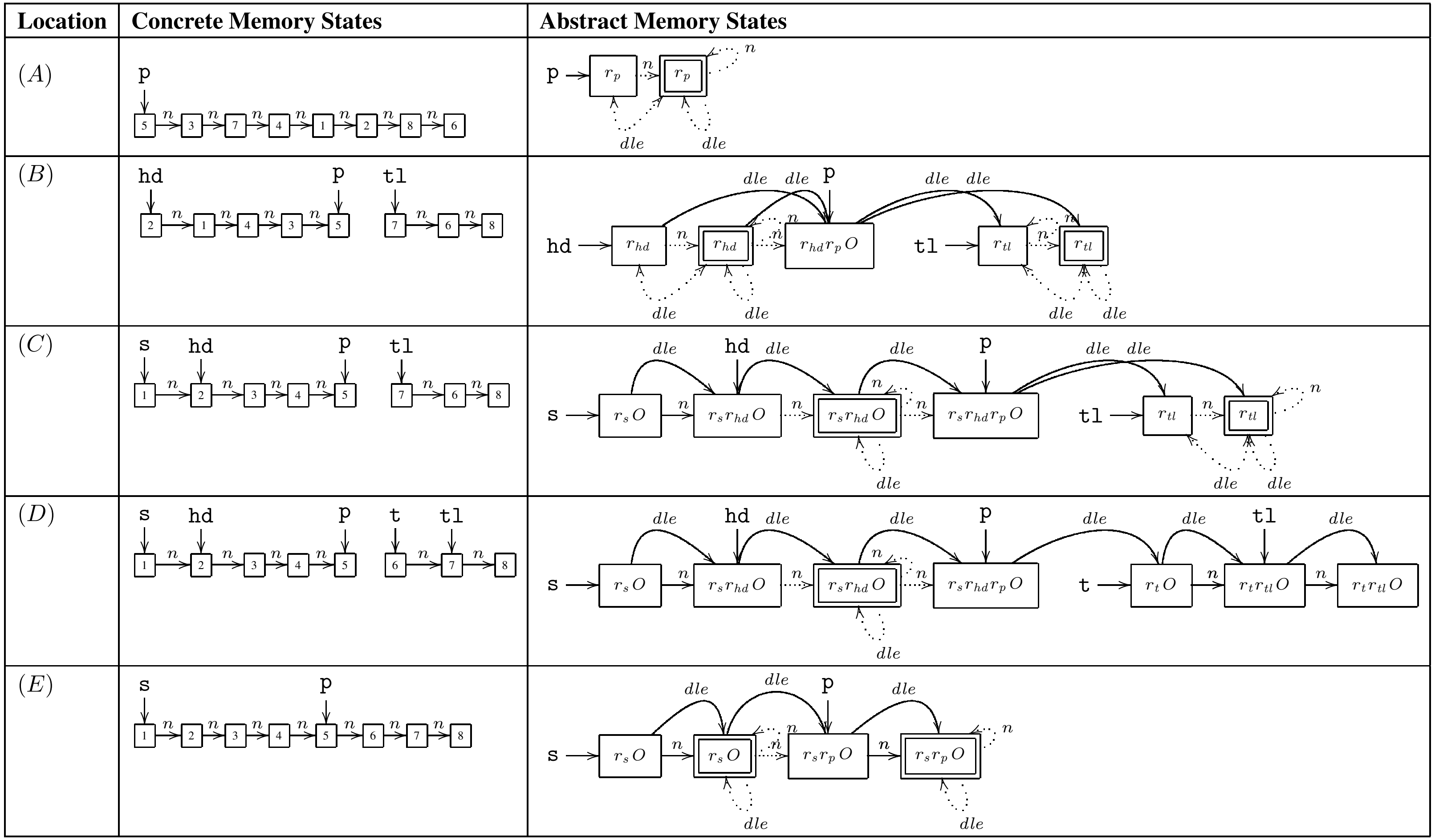


Figure 16: Concrete memory states that occur during the execution of `quicksort` and their abstractions.