Interprocedural Shape Analysis for Effectively Cutpoint-Free Programs

J. Kreiker[1], T. Reps[2⋆], N. Rinetzky[3⋆⋆], M. Sagiv[4], R. Wilhelm[5], and E. Yahav[6]

[1] Technical University of Munich joba@model.in.tum.de
[2] University of Wisconsin reps@cs.wisc.edu
[3] Queen Mary University of London maon@eecs.qmul.ac.uk
[4] Tel Aviv University msagiv@tau.ac.il
[5] University des Saarlandes wilhelm@cs.uni-sb.de
[6] IBM T.J. Watson Research Center eyahav@us.ibm.com

Abstract. We present a framework for local interprocedural shape analysis that computes procedure summaries as transformers of procedure-local heaps (the parts of the heap that the procedure may reach). A main challenge in procedure-local shape analysis is the handling of *cutpoints*, objects that separate the input heap of an invoked procedure from the rest of the heap, which—from the viewpoint of that invocation—is non-accessible and immutable.

In this paper, we limit our attention to *effectively cutpoint-free* programs—programs in which the only objects that separate the callee's heap from the rest of the heap, when considering *live* reference fields, are the ones pointed to by the actual parameters of the invocation. This limitation (and certain variations of it, which we also describe) simplifies the local-reasoning about procedure calls because the analysis needs not track cutpoints. Furthermore, our analysis (conservatively) verifies that a program is effectively cutpoint-free,

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 [3]. However, performing strong pointer updates requires a flow-sensitive and context-sensitive analysis and expensive heap abstractions, which may be doubly-exponential in the program size [25]. The presence of procedures escalates the problem because of interactions between the program stack and the heap [22] and because recursive calls may introduce additional exponential factors in an analysis. This makes interprocedural shape analysis a challenging problem.

This paper introduces a new approach for *local* [10,18] interprocedural shape analysis for a class of imperative programs. The main idea is to restrict the aliasing between

⋆ Supported by NSF under grants CCF-0540955, CCF-0810053, and CCF-0904371, by ONR under grant N00014-09-1-0510, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.
⋆⋆ Supported by EPSRC.

live access paths at procedure calls. This allows procedure invocations to be analyzed ignoring *non-relevant* parts of the heap, more specifically, the parts of the heap not reachable from actual parameters. Moreover, shape analysis verifies that the above restrictions are satisfied.

The restricted class of programs is chosen based on observations made in [20]. There, Rinetzky et al. present a non-standard semantics in which procedures operate on procedure-local heaps containing only the objects reachable from actual parameters. The most complicated aspect of [20] is the treatment of sharing from the global heap and local variables of pending calls into the procedure-local heap. The problem is that the local heap can be accessed via access paths that bypass actual parameters. Therefore, objects in the local heap are treated differently when they separate the local heap (accessible by a procedure) from the rest of the heap (which—from the viewpoint of that procedure—is non-accessible and immutable). These objects are referred to as *cutpoints* [20].

Example 1. Fig. 1 illustrates the notions of local heaps and cutpoints. To gain intuition, Fig. 1 shows these notions using the familiar *store-based* semantics. (See, e.g., [18]). The figure depicts a memory state of a program comprised of four procedures: `main`, `foo`, `bar`, and `zoo`. The figure depicts a memory state that may occur at the entry to `zoo`. The stack of activation records is depicted on the left side of the diagram. Each activation record is labeled with the name of the procedure it is associated with. Thus, as we can see, `zoo` was invoked by `bar`; procedure `bar` was invoked by `foo`; and `foo` was invoked by the `main` procedure. The activation record at the top of the stack pertains to the *current* procedure (`zoo`). All other activation records pertain to *pending* procedure calls. Thus, for example, the access paths `z1.f1.f1`, `y9`, and `x5.f2` are pending access paths.

Heap-allocated objects are depicted as rectangles labeled with their location. The value of a reference variable (resp. field) is depicted by an edge labeled with the name of the variable (resp. field). The shaded cloud marks the part of the heap that `zoo` can access (i.e., the part of the heap containing the relevant objects for the invocation). The cutpoints for the invocation of `zoo` ($u8$ and $u9$) are heavily shaded. Note that $u7$ is not a cutpoint because it is also pointed to by $h7$, `zoo`'s formal parameter.

Cutpoints present a major challenge for shape abstractions: Procedure-local heaps together with special handling of cutpoints was found to be key in obtaining efficient and precise interprocedural shape-analysis algorithms [28]. Thus, the shape abstraction cannot abstract away the sharing patterns induced by cutpoints between the procedure-local heap of the procedure and the rest of the heap. These sharing patterns may lack any regular shape. However, the regularity of the sharing pattern is, in fact, what enables the effective shape abstraction of unbounded linked data structures.

We observe that cutpoints need special treatment in the analysis of a procedure because the caller may use its direct references to the cutpoint after the procedure returns. We develop an interprocedural shape analysis in which such direct usages are forbidden. We refer to a reference that, at the time when a procedure is invoked, points to a cutpoint and does not come from an object in the callee's local heap as a *piercing reference* for that invocation. An execution is *effectively cutpoint-free* if in every invocation that occurs during the execution, all the piercing references for that invocation are not *live* [26] at the time of the invocation, i.e., their r-values are not used later on in the execution before being set. A program is effectively cutpoint-free if all its executions

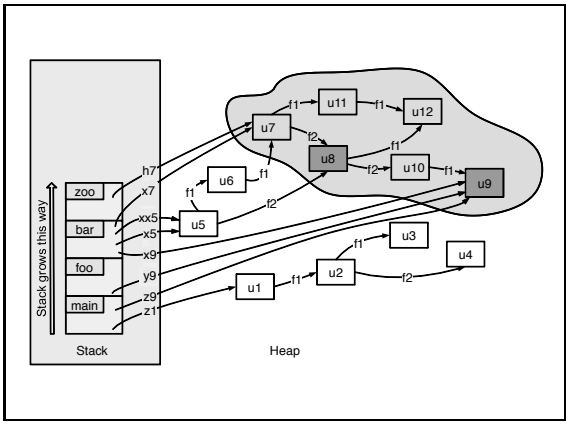

Fig. 1. An illustration of the cutpoints for an invocation in a store-based small-step (stack-based) operational semantics at the entry to `zoo`. We assume that $h7$ is `zoo`'s formal parameter.

are. When analyzing effectively cutpoint-free programs, there is no need to give special care to cutpoint objects. However, to verify that a program is effectively cutpoint-free, special care needs to be taken regarding future usages of piercing references.

In this paper we present $\mathcal{ECPF}$, a small-step operational semantics [16] that handles *effectively cutpoint-free* programs. This semantics is interesting because procedures operate on local heaps, i.e., every procedure invocation starts executing on a memory state in which *parts of the heap not relevant to the invocation are ignored*. Thus, $\mathcal{ECPF}$ supports the notion of *heap-locality* [10,18] while permitting the usage of a global heap and destructive updates. Moreover, the absence of cutpoints drastically simplifies the meaning of procedure calls. $\mathcal{ECPF}$ tracks the set of piercing references and checks that their values are never used, thus dynamically verifying that the program execution is indeed effectively cutpoint-free. As a result, $\mathcal{ECPF}$ is applicable to arbitrary programs, and does not require an a priori classification of a program as effectively cutpoint-free. We show that for effectively cutpoint-free programs, $\mathcal{ECPF}$ is observationally equivalent to the standard global heap semantics.

$\mathcal{ECPF}$ gives rise to a functional [6,27] interprocedural shape analysis for effectively cutpoint-free programs. The analysis tabulates abstractions of memory states before and after procedure calls. Mimicking the semantics, memory states are represented in a procedure-local way *ignoring parts of the heap not relevant to the procedure with no special abstraction for cutpoints*. This reduces the complexity of the analysis because the analysis of procedures does not represent information about references and the heap from calling contexts. Indeed, this makes the analysis local in the heap and thus allows reusing the summarized effect of a procedure at different calling contexts.

Technically, our algorithm is built on top of the 3-valued logical framework for program analysis of [13,25]. Thus, it is parametric in the heap abstraction and in the concrete effects of program statements, which allows experimenting with different instances of interprocedural shape analyzers. For example, we can employ different ab-

stractions for singly-, doubly-linked lists, and trees. Also, a combination of theorems in Appendix A.2 and [25] guarantees that every instance of our *interprocedural* framework is sound (see Sec. 5).

Main results. The contributions of this paper can be summarized as follows:

1. We define the notion of effectively cutpoint-free programs, in which the context not reachable from a procedure's actual parameters can be ignored when reasoning about the procedure's possible effect.
2. We define an operational semantics for a simple imperative language with references and procedures. The semantics dynamically checks that a program execution is effectively cutpoint-free. Procedures operate on procedure-local heaps, thus supporting the notion of heap-locality while permitting the usage of a global heap and destructive updates.
3. We present an interprocedural shape analysis for effectively cutpoint-free programs. The analysis is local in the heap and thus allows reusing the effect of a procedure at different calling contexts and at different call-sites.
4. We describe several extensions to our approach that allow its efficiency, precision, and applicability to be improved by utilizing a limited form of user-supplied annotations.

Outline. The rest of the paper is organized as follows. Sec. 2 presents an informal overview of our approach. Sec. 3 introduces our programming model. Sec. 4 defines our new local heap semantics, which checks whether a program is effectively cutpoint-free. Sec. 5 conservatively abstracts this semantics and provides the semantic foundation of the local interprocedural shape analysis algorithm described in Sec. 6. Sec. 7 describes certain efficiency-oriented extensions of our approach and certain relaxations of our restrictions aimed at increasing the class of effectively cutpoint-free programs. Sec. 8 describes related work, and Sec. 9 concludes.

2 Overview

This section provides an overview of our framework for interprocedural shape analysis using procedure-local heaps. The presentation is at an intuitive level; a more detailed treatment of this material is presented in the later sections of the paper.

2.1 Motivating Example

Fig. 2 shows a simple Java program that splices three non-shared, disjoint, acyclic singly-linked lists using a recursive `splice` procedure. This program serves as a running example in this paper.

2.2 Procedure-Local Heaps

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.

```
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);
     List t2 = new List(k+1);
     List t3 = new List(k+2);
     t1.n = t2; t2.n = t3;
     return t1;
  }

  static public int getData(List w) {
     assert(w != null);
     int d = w.data;
     return d;
  }
```

```
  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);
     List y = create3(4);
     List z = create3(7);
     List t = splice(x, y);
     List s = splice(t, z);
     int i = 0;
     ℓ0 : // if (y == null) i++;
     ℓ1 : // if (y == x) i++;
     ℓ2 : // int i = getData(y);
     print(i);
  }
}
```

Fig. 2. An effectively-cutpoint-free program written in Java

Example 2. Fig. 3 shows the concrete memory states that occur at the call `t=splice(x,y)`. S_3^c shows the state at the point of the call, and S_3^e shows the state on entry to `splice`. Here, `splice` is invoked on local heaps 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`.

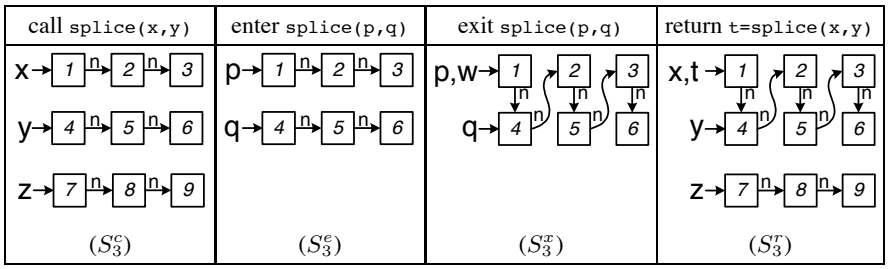

Fig. 3. Concrete states for the invocation `t = splice(x, y)` in the running example.

2.3 Cutpoints and Cutpoint-Freedom

Obviously, this is not always the case. In particular, consider the second call in the example program, `s=splice(t,z)`. Fig. 4(a) shows the concrete states when

`s=splice(t,z)` is invoked. $S_4^{c_{cp}}$ shows the state on invocation, and $S_4^{r_{cp}}$ the state when the call returns. 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`.

To emphasize the effect of this invocation, consider a variant of the example program in which the invocation `s=splice(t,z)` has been replaced with an invocation `s=splice(y,z)`, as shown in Fig. 4(b). In this variant, the invocation can only affect access paths that pass through an object referenced by either `y` or `z`.

We capture the difference between these invocations by introducing the notion of a cutpoint [20]. A cutpoint for an invocation is an object that is: (i) reachable from an actual parameter, (ii) not pointed-to by an actual parameter, and (iii) reachable without going through an object that 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 that is reachable for the invocation from the rest of the heap, but not pointed-to by a parameter.

For example, the object pointed-to by `y` at the call `s=splice(t,z)` (Fig. 4(a)) is a *cutpoint*, thus this invocation is not *cutpoint-free* [23]. In contrast, in the invocation `s=splice(y,z)` (Fig. 4(b)) no object is a cutpoint, and thus this invocation is *cutpoint-free* [23].

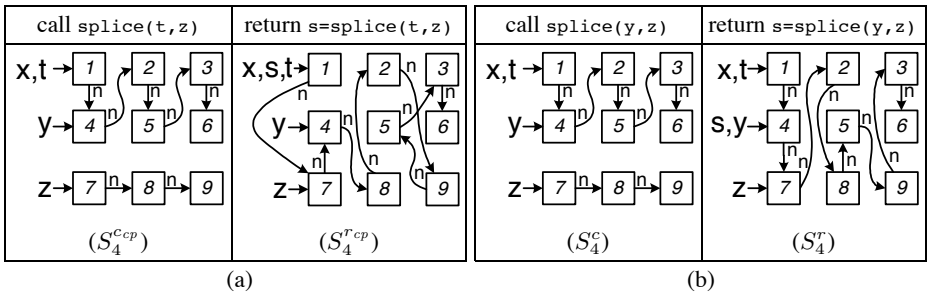

Fig. 4. Concrete states for: (a) the invocation `s=splice(t,z)` in the program of Fig. 2; (b) a variant of this program with an invocation `s=splice(y,z)`.

2.4 Effective Cutpoint-Freedom

The importance of cutpoints is that they allow the analysis to handle more precisely the notion of procedure local variables: No invocation of `splice` can modify the local variables of `main`. Thus, when control returns to `main`, it is guaranteed that the local variable `y` points to the same object that it pointed to before the invocation, and the `main` procedure can use the `y` reference to access directly that object. In general, it is very challenging to design a shape analysis that can track relations between arbitrary objects across the execution of procedure calls. However, if the caller does not use its direct references to the cutpoints after the procedure returns, the analysis does not need to track this relation.

For example, note that after `main` regains control, it does not use the value of the `y` variable. Thus, although the invocation `s=splice(t,z)` has a cutpoint, and is thus not cutpoint-free, in the context of the whole execution this invocation is *effectively cutpoint free*.

The semantics utilizes the above observation and instead of giving special treatment to the cutpoint objects, it assigns a special *inaccessible* value to all piercing references. The inaccessible value is used to track references which should not be used. It is a simple mechanism which the semantics uses to check (in runtime) whether a piercing pointer is used , e.g., in a dereference operation or during the evaluation of a condition, and if such a usage occurs to abort the execution and report that the program is not effectively cutpoint-free. (See Sec. 4).

Example 3. Fig. 5 shows the concrete memory states that occur at the call `s=splice(t,z)`. S_5^c shows the state at the point of the call, in which the object pointed to by `y` is a cutpoint. In S_5^r, the return state of that call, `y` no longer points to an object, instead it has the inaccessible value, depicted by a black bullet. The semantics intentionally does not utilize the information it has regarding the identity of objects. It acts as if it "forgets" that the object referenced by `y` at the call state is the third node in the returned list, mimicking in the concrete semantics the loss of information that occurs in the analysis. Note that the cutpoint object is not treated differently during the execution of `splice`, e.g., S_5^e and S_5^x show the states on entry to `splice` of the call and at its exit, respectively.

Also note that if any of the statements in lines $\ell_0 - \ell_2$ was to be uncommented, variable `y` would have been live at the time of the call `s=splice(t,z)`, and thus the execution would not have been effectively cutpoint-free.

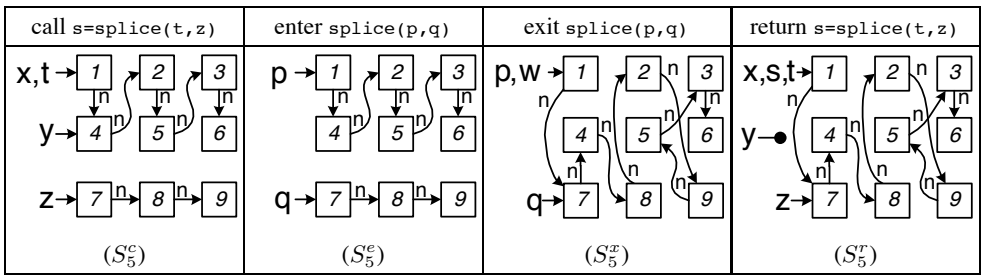

Fig. 5. Concrete states for the invocation `s = splice(t, z)` in the running example.

2.5 Interprocedural Shape Analysis

The algorithm computes procedure summaries by tabulating pairs of abstract input memory-states and abstract output memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The tabulated abstract memory-states represent procedure-local heaps, but do not keep track of cutpoints. However, they do record the inaccessible values. Therefore, these abstract states are

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 while sustaining enough information to verify effective cutpoint freedom.

3 Programming Model

For expository reasons we limit our attention to a small imperative programming language. It has references to objects. Objects have fields, which can be either references to other objects or integers. The analyses developed here can be applied to Java-like languages and other imperative pointer languages alike (unless pointer arithmetic is used).

We abstract from specific control-flow statements and simply assume the presence of one control-flow graph per procedure. Control-flow graph edges are annotated with any one of the following statements below, where `x.f` denotes the `f` field of the object referenced by `x`. The statement `x = alloc()` returns a reference to a newly created object. Conditionals are implemented using `assume` statements.

$$\begin{aligned} \textit{stms} ::= {} & \mathtt{x} = \mathtt{null} \mid \mathtt{x} = \mathtt{y} \mid \mathtt{x} = \mathtt{y.f} \mid \\ & \mathtt{x.f} = \mathtt{y} \mid \mathtt{x} = \mathtt{alloc}() \mid \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}) \mid \\ & \mathtt{y} = \mathtt{p}(\mathtt{x}_1, \ldots, \mathtt{x}_k) \mid \mathtt{return} \end{aligned}$$

In our running example we take the liberty to use integer variables and fields as well.

In the rest of the paper, we assume that we are working with a fixed arbitrary program P. For a procedure p, V_p denotes the set of its local variables and $F_p \subseteq V_p$ denotes the set of its formal parameters. A procedure returns a value by assigning it to a designated variable `ret`. We assume that parameters are passed by value and that formal parameters cannot be assigned to. The set of all local variables of P is written $\mathcal{V}$. We write $\mathcal{F}$ to denote the set of all field names in P.

We assume a standard store-based operational semantics for our language, very much like $\mathcal{GSB}$ defined previously in [19,20]. $\mathcal{GSB}$ treats live cutpoints properly.

4 Concrete Semantics

In this section, we define $\mathcal{ECPF}$ (*effectively cutpoint-free*), a non-standard semantics that checks whether a program execution is effectively cutpoint-free. $\mathcal{ECPF}$ defines the execution traces that are the foundation of our analysis.

$\mathcal{ECPF}$ is a *store-based* semantics (see, e.g., [18]). A traditional aspect of a store-based semantics is that a memory state represents a heap comprised of all the allocated objects. $\mathcal{ECPF}$, on the other hand, is a *procedure-local heap* semantics [20]: A memory state that occurs during the execution of a procedure does not represent objects that, at the time of the invocation, are not reachable from the actual parameters.

$\mathcal{ECPF}$ is a small-step operational semantics [16]. Instead of encoding a stack of activation records inside the memory state, as is traditionally done, $\mathcal{ECPF}$ maintains a *stack of program states* [12,21]: Every program state consists of a program point and a memory state. The program state of the *current procedure* is stored at the top of the

stack, and it is the only one that can be manipulated by intraprocedural statements. We refer to this memory state as the *current memory state*. When a procedure is invoked, the *entry memory state* of the callee is computed by a $Call$ operation according to the caller's current memory state, and pushed onto the stack. When a procedure returns, the stack is popped, and the caller's *return memory state* is updated using a Ret operation according to its memory state before the invocation (the *call memory state*) and the callee's (popped) *exit memory state*. The $Call$ and Ret operations of $\mathcal{ECPF}$ are defined in Fig. 8.

The use of a stack of program states allows us to represent in every memory state the (values of) local variables and the local heap of just one procedure. The *lifting* of an intraprocedural semantics to an interprocedural semantics, that uses a stack of program states, is formally defined in [19].

An execution trace of a program P always begins with P's `main` executing on an *initial memory state* in which all its reference variables have the value *null* and the heap is empty. We say that a memory state is *reachable* in a program P if it occurs as the current memory state in an execution trace of P.

$\mathcal{ECPF}$ is a procedure-local heap semantics [20]: when a procedure is invoked, it starts executing on an *input heap* containing only the set of *relevant objects for the invocation*. An object is *relevant for an invocation* if it is a *parameter object*, *i.e.*, either referenced by an actual parameter or reachable from one.

A procedure-local heap semantics and its abstractions benefit from not having to represent irrelevant objects. However, in general, the semantics needs to take special care of cutpoints. In this paper, we avoid the need to take special care of cutpoint objects by assuming and verifying that a program is *effectively cutpoint free*: We refer to a reference that at invocation time points to a cutpoint and does not come from an object in the callee's local heap as a *piercing reference* for that invocation. An execution is *effectively cutpoint-free* if in every of its invocations during an execution all the piercing references for that invocation are *dead* at the time of the invocation, i.e., their r-values are not used before being set. A program is *effectively cutpoint-free* if all of its executions are.

For effectively cutpoint-free programs, there is no need to give special care to cutpoint objects. However, to verify that a program is effectively cutpoint-free, special care needs to be taken regarding the piercing references. In this section, we describe the way $\mathcal{ECPF}$ validates at runtime that an execution is effectively cutpoint-free.

4.1 Memory States

Fig. 6 defines the concrete semantic domains and the meta-variables ranging over them. We assume Loc to be an unbounded set of locations. A value $v \in Val$ is either a location, *null*, or $\bullet$, the inaccessible value used to represent references to locations that should not be accessed.

A memory state in the $\mathcal{ECPF}$ semantics is, essentially, a 2-level store. Formally, a memory state is a 3-tuple $\sigma = \langle \rho, L, h \rangle$: $\rho \in \mathcal{E}$ is an environment assigning values for the variables of the *current* procedure. $L \subset Loc$ is the set of allocated locations. (A dynamically allocated object is identified by its location. We interchangeably use the terms object and location.) $h \in \mathcal{H}$ assigns values to fields of allocated objects.

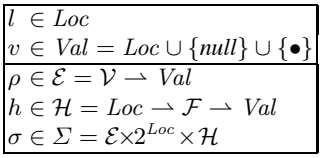

Fig. 6. Semantic domains.

In $\mathcal{ECPF}$, reachability is defined with respect to relevant objects: Informally, an object l_2 is *reachable from* an object l_1 in a memory state σ if there is a directed path in the heap of σ from l_1 to l_2. An object l is *reachable* in σ if it is reachable from a location that is pointed-to by some variable. Note that $\bullet$-valued references do not point to any object.

4.2 Operational Semantics of Intraprocedural Statements

The meaning of atomic statements is described by a transition relation $\overset{i}{\rightsquigarrow} \subseteq (\Sigma \times \mathit{stms}) \times \Sigma \uplus \{\sigma_\bullet\}$, where $\sigma_\bullet$ is a special error state indicating a forbidden usage of the inaccessible value.

Fig. 7 defines the axioms for atomic intraprocedural statements. These are handled as in a standard 2-level store semantics like $\mathcal{GSB}$.[7] The main difference between the $\mathcal{ECPF}$ semantics and $\mathcal{GSB}$ with respect to the meaning of intraprocedural statements is captured by the side-conditions of the form $\rho(x) = \bullet$ or $\rho(y) = \bullet$, which prevent usage of the inaccessible locations.

$\langle \mathtt{x} = \mathtt{null}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto \mathit{null}], L, h\rangle$	
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto \rho(y)], L, h\rangle$	
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto h(\rho(y), f)], L, h\rangle$	$\rho(y) \in Loc$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho, L, h[(\rho(y), f) \mapsto \rho(x)]\rangle$	$\rho(y) \in Loc$
$\langle \mathtt{x} = \mathtt{alloc}(), \sigma\rangle \overset{i}{\rightsquigarrow} \langle \rho[x \mapsto l], L \cup \{l\}, h[l \mapsto I]\rangle$	$l \in Loc \setminus L$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{i}{\rightsquigarrow} \sigma$	$\rho(x) \bowtie \rho(y)$
$\langle \mathtt{x} = \mathtt{y}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$
$\langle \mathtt{x} = \mathtt{y.f}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$ or $h(\rho(y)) = \bullet$
$\langle \mathtt{y.f} = \mathtt{x}, \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(y) = \bullet$ or $\rho(x) = \bullet$
$\langle \mathtt{assume}(\mathtt{x} \bowtie \mathtt{y}), \sigma\rangle \overset{i}{\rightsquigarrow} \sigma_\bullet$	$\rho(x) = \bullet$ or $\rho(y) = \bullet$

Fig. 7. Axioms for intraprocedural statements, where in each line σ is understood as a shorthand for $\langle \rho, L, h\rangle$. I denotes the function $\lambda f \in \mathcal{F}.\mathit{null}$. $\bowtie$ stands for either $=$ or $\neq$. When convenient, we sometimes treat h as an uncurried function, i.e., as a function from $Loc \times \mathcal{F}$ to Val.

4.3 Operational Semantics of Interprocedural Statements

Fig. 8 defines the meaning of the $Call$ and Ret operations pertaining to an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ assuming p's formal parameters are $z_1, \ldots, z_k$, the memory state at the call site is $\sigma_c = \langle \rho_c, L_c, h_c \rangle$, and the memory state at the exit of p is $\sigma_x = \langle \rho_x, L_x, h_x \rangle$. The $Call$ operation is used to compute the state update along a call edge in the control-flow graph; the Ret operation computes the state update along a return edge. As defined in Sec. 3, variable `ret` is used to communicate the return value. We use the function $R_h(L)$ to compute the locations that are reachable in heap h from the set of locations L. This function is formally defined in Appendix A.1.

$$
\begin{array}{ll}
Call_{y=p(x_1,\ldots,x_k)}(\sigma_c) = \sigma_e & Ret_{y=p(x_1,\ldots,x_k)}(\sigma_c, \sigma_x) = \sigma_r \\
\quad \sigma_e = \langle \rho_e, L_c, h_c|_{L_{rel}} \rangle & \quad \sigma_r = \langle \rho_r, L_x, h_r \rangle \\
\quad\quad \rho_e = [z_i \mapsto \rho_c(x_i) \mid 1 \leq i \leq k] & \quad\quad \rho_r = (block \circ \rho_c)[y \mapsto \rho_x(ret)] \\
 & \quad\quad h_r = (block \circ h_c|_{L_c \setminus L_{rel}}) \cup h_x
\end{array}
$$

where:

$$
\begin{array}{l}
L_{parameters} = \{\rho_c(x_i) \in Loc \mid 1 \leq i \leq k\} \\
L_{rel} = R_{h_c}(L_{parameters}) \\
L_{cutpoints} = (L_{rel} \setminus L_{parameters}) \cap \\
\qquad\qquad (\{\rho_c(z) \mid z \in V_q\} \cup \{h_c(l)f \in Loc \mid l \in L_c \setminus L_{rel}, f \in \mathcal{F}) \\
block = \lambda v \in Val. \begin{cases} \bullet & v \in L_{cutpoints} \\ v & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
Call_{y=p(x_1,\ldots,x_k)}(\sigma_c) = \sigma_\bullet & \rho_c(x_1) = \bullet \text{ or } \cdots \text{ or } \rho_c(x_k) = \bullet \\
Ret_{y=p(x_1,\ldots,x_k)}(\sigma_c, \sigma_x) = \sigma_\bullet & \rho_x(ret) = \bullet
\end{array}
$$

Fig. 8. $Call$ and Ret operations for an arbitrary procedure call $y = p(x_1, \ldots, x_k)$ by an arbitrary procedure q, where it is understood that $\sigma_c = \langle \rho_c, L_c, h_c \rangle$, $\sigma_x = \langle \rho_x, L_x, h_x \rangle$, and V_q denotes the set of local variables of procedure q.

Procedure calls The $Call$ operation computes the callee's *entry memory state* (σ_e) from the state at the call-site (σ_c). The entry memory state is computed by binding the values of the formal parameters in the callee's environment to the values of the corresponding actual parameters (ρ_e) and restricting the caller's heap to the relevant objects for the invocation (L_{rel}).

Example 4. Fig. 3 shows the entry state S_3^e that results from applying the $Call$ operation pertaining to the invocation `t=splice(x, y)` to the call memory state S_3^c. Fig. 5 shows the entry state S_5^e that results from applying the $Call$ operation pertaining to the invocation `s=splice(t, z)` to the call memory state S_5^c.

Procedure returns The Ret operation maps the memory state at the exit of a procedure (σ_x) together with the state at call-site (σ_c) to the return state σ_r from which the caller resumes its computation. Ret updates the caller's memory state by carving out the input heap passed to the callee from the caller's heap ($h_c|_{L_c \setminus L_{rel}}$) and replacing it with the callee's (possibly) mutated heap (h_x).

In $\mathcal{ECPF}$, an object never changes its location, and locations are never reallocated. Thus, any pointer to a relevant object in the caller's memory state (either by a field of an irrelevant object or a variable) points after the replacement to an up-to-date version of the object.

Blocking piercing references. $\mathcal{ECPF}$ detects forbidden accesses that violate the effective-cutpoint-freedom condition, and aborts the program in an error state if such an access is detected. Technically, when a procedure invocation returns, $\mathcal{ECPF}$ assigns the special value $\bullet$ to all piercing references, an operation which we refer to as *blocking*, and uses this special value to detect forbidden accesses. (Recall that in an effectively cutpoint-free execution, every live reference that points to an object which separate the callee's heap from the caller's heap should point to a parameter object, i.e., to one of the objects in $L_{parameters}$.)

Example 5. Fig. 3 shows the return state S_3^r, that results from applying the Ret operation pertaining to the invocation `t=splice(x, y)` to the call memory state S_3^c and the exit memory state S_3^x. Fig. 5 shows the return state S_5^r, that results from applying the Ret operation pertaining to the invocation `s=splice(t, z)` to the call memory state S_5^c and the exit memory state S_5^x. The second node in the list pointed to by `t` at the call state S_5^c is a cutpoint. Thus, variable `y` gets blocked when computing S_5^r.

4.4 Observational Soundness

We say that two values are *comparable* in $\mathcal{ECPF}$ if neither one is $\bullet$. We say that a $\mathcal{ECPF}$ memory state σ is *observationally sound* with respect to a standard semantics σ_G if for every pair of access paths that have comparable values in σ, they have equal values in σ iff they have equal values in σ_G. $\mathcal{ECPF}$ *simulates* the standard 2-level store semantics: Executing the same sequence of statements in the $\mathcal{ECPF}$ semantics and in the standard semantics either results in a $\mathcal{ECPF}$ memory states that is observationally sound with respect to the resulting standard memory state, or the $\mathcal{ECPF}$ execution gets to an *error state* due to a constraint breach (detected by $\mathcal{ECPF}$). A program is *effectively cutpoint-free* if it does not have an execution trace that gets to an error state. (Note that the initial state of an execution in $\mathcal{ECPF}$ is observationally sound with respect to its standard counterpart).

Our goal is to detect structural invariants that are true according to the *standard semantics*. $\mathcal{ECPF}$ acts like the standard semantics as long as the program's execution satisfies certain constraints. $\mathcal{ECPF}$ enforces these restrictions by blocking references that a program should not access. Similarly, our analysis reports an invariant concerning equality of access paths only when these access paths have comparable values.

An invariant concerning equality of access paths in $\mathcal{ECPF}$ for an effectively cutpoint-free program is also an invariant in the standard semantics. This makes abstract

interpretations of $\mathcal{ECPF}$ suitable for verifying data-structure invariants, for detecting memory access violations, and for performing compile-time garbage collection.

5 Abstract Interpretation

In this section, we present $\mathcal{ECPF}^{\#}$, an abstract interpretation [5] of the $\mathcal{ECPF}$ semantics. $\mathcal{ECPF}^{\#}$ is the basis of our static-analysis algorithm which uses the 3-valued logic-based framework of [25]. The soundness of the abstract semantics with respect to $\mathcal{GSB}$[7] is guaranteed by the combination of the theorems in Appendix A.2 and [25]:

- In Appendix A.2, we show that for effectively cutpoint-free programs, $\mathcal{ECPF}$ is observationally equivalent to $\mathcal{GSB}$.
- In [25], it is shown that every program-analyzer that is an instance of the 3-valued logic-based framework is sound with respect to the concrete semantics it is based on.

5.1 Abstract States

We conservatively represent unbounded sets of unbounded memory states using a bounded set of bounded 3-valued logical structures, which we refer to as *abstract states*. Note that there are actually three different notions of *concrete states*. The most concrete states are those in $\mathcal{GSB}$, containing full information including integer variables and fields. Integers are already abstracted away when we talk about $\mathcal{ECPF}$, which, on top of that, also yields errors when cutpoint references are illegally used. $\mathcal{ECPF}$ states are equivalently encoded into *two-valued* logical structures by viewing objects as individuals in a logical structure and references as binary predicates (see below). Note, however, that location identifiers play no role in the logical structure encoding. Indeed, the semantics does not distinguish between isomorphic structures.

We use the term *concrete state* whenever we talk about a state that is not a 3-valued logical structure. We believe that, despite the resulting imprecision, our intentions are clear. In drawings, we use the same graphical notations to depict concrete states in all of the aforementioned semantics. (Integer values, when drawn, should be ignored when considering a figure to be a graphical depiction of a state in $\mathcal{ECPF}$ or of a logical structure.)

3-valued logical structures. A 3-valued logical structure is a logical structure with an extra truth-value $\frac{1}{2}$, which denotes values that may be 1 or may be 0. The information partial order on the set $\{0, \frac{1}{2}, 1\}$ is defined as $0 \sqsubseteq \frac{1}{2} \sqsupseteq 1$, and $0 \sqcup 1 = \frac{1}{2}$. Formally, a 3-valued logical structure is $S^\sharp = \langle U^{S^\sharp}, \iota^{S^\sharp} \rangle$ where:

- $U^{S^\sharp}$ is the universe of the structure.
- $\iota^{S^\sharp}$ 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, $\iota^S(p) \colon {U^{S^\sharp}}^k \to \{0, \frac{1}{2}, 1\}$.

[7] $\mathcal{GSB}$ is a standard two-level store semantics for heap-manipulating programs. It is formally defined in [20].

A 2-valued logical structure is a 3-valued logical structure where the truth-values of predicates are either 0 or 1. The set of *3-valued* logical structures is denoted by $\mathit{3Struct}$. The set of *2-valued* logical structures is denoted by $\mathit{2Struct}$.

Abstraction function. We abstract sets of $\mathcal{ECPF}$ memory states by a point-wise application of an *extraction function* $\beta : \Sigma \rightharpoonup \mathit{3Struct}$ mapping an $\mathcal{ECPF}$ memory state to its *best representation* by an *abstract state*. The extraction function β is defined as a composition of two functions: (i) $\beta_{shape} : \Sigma \rightharpoonup \mathit{2Struct}$, which maps an $\mathcal{ECPF}$ memory state to a 2-valued logical structure and (ii) *canonical abstraction* [25], which maps 2-valued logical structures to a bounded number of 3-valued logical structures.

Representing Memory States using 2-Valued Logical Structures We represent $\mathcal{ECPF}$ memory states using 2-valued logical structures. Every individual in the structure corresponds to a heap-allocated object. Predicates of the structure correspond to properties of heap-allocated objects.

Core predicates. Tab. 1 shows the core predicates used in this paper. A binary predicate $f(v_1, v_2)$ holds when the $\mathtt{f} \in \mathcal{F}$ field of v_1 points to v_2. The designated binary predicate $eq(v_1, v_2)$ is the equality predicate, which records equality between v_1 and v_2. A unary predicate $x(v)$ holds for an object that is referenced by the reference variable $\mathtt{x} \in \mathcal{V}$ of the *current* procedure.[8] The predicate ia holds only for a unique individual, which represents the inaccessible locations. The role of the predicates $inUc$ and $inUx$ is explained in Sec. 5.2.

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. Tab. 2 lists the instrumentation predicates used in this paper. We use $F(v_1, v_2)$ as a shorthand to denote that v_1 has a field $f \in \mathcal{F}$ which points to v_2 and $F^*(v_1, v_2)$ as the reflexive transitive closure of F. (For a formal definition, see Appendix B).

2-valued logical structures are depicted as directed graphs. We draw individuals as boxes. We depict the value of a reference variable `x` by drawing an edge from `x` to the individual representing the object that `x` references. 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 binary equality predicate eq. The inaccessible value is depicted as a line ending with $\bullet$.

Example 6. The structure S_3^c of Fig. 3 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.

The structure S_5^r of Fig. 5 shows a *2-valued* logical structure that represents the memory state of the program at the return of `s=splice(t, y)`. Note that the value of `y` is the inaccessible value.

[8] 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 to represent (concrete) memory states.

Predicate	Intended Meaning
$f(v_1, v_2)$	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
$ia(v)$	v is an inaccessible location
$inUc(v)$	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

Table 2. 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	$\neg ia(v_1) \wedge \neg ia(v_2) \wedge F^*(v_1, v_2)$
$ils(v)$	v is *locally* shared. i.e., v is pointed-to by a field of more than one object in the *local heap*	$\exists v_1, v_2 \colon \neg ia(v)$ $\neg eq(v_1, v_2) \wedge F(v_1, v) \wedge F(v_2, v)$
$c(v)$	v resides on a directed cycle of fields	$\exists v_1 \colon F(v, v_1) \wedge F^*(v_1, v)$
$r_x(v)$	v is reachable from variable `x`	$\neg ia(v) \wedge \exists v_x \colon x(v_x) \wedge F^*(v_x, v)$

Bounded Abstraction We now formally define how memory states are represented using abstract memory states. The idea is that each object from the (concrete) state is mapped to an individual in the abstract state. An abstract memory state may include *summary nodes*, i.e., individuals that correspond to one or more concrete nodes in one of the concrete states represented by the abstract state. For a summary node $u \in U^\sharp$ in abstract state $S^\sharp = \langle U^\sharp, \iota^\sharp \rangle$ it holds that $\iota(eq)(u, u) = \frac{1}{2}$.

Canonical abstraction. A *3-valued* logical structure $S^\sharp$ is a **canonical abstraction** of a *2-valued* logical structure S if there exists a surjective function $\upsilon \colon U^S \to U^{S^\sharp}$ satisfying the following conditions: (i) For all $u_1, u_2 \in U^S$, $\upsilon(u_1) = \upsilon(u_2)$ 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 $u_1^\sharp, u_2^\sharp, \ldots, u_k^\sharp \in U^{S^\sharp}$,

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

3-valued logical structures are also drawn as directed graphs. Definite values (0 and 1) are drawn as for 2-valued structures. Binary indefinite predicate values ($\frac{1}{2}$) are drawn as dotted directed edges. Summary nodes are depicted by a double frame.

Example 7. Fig. 9 shows the abstract states (as *3-valued* logical structures) representing the concrete states of Fig. 3. 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., [22]).

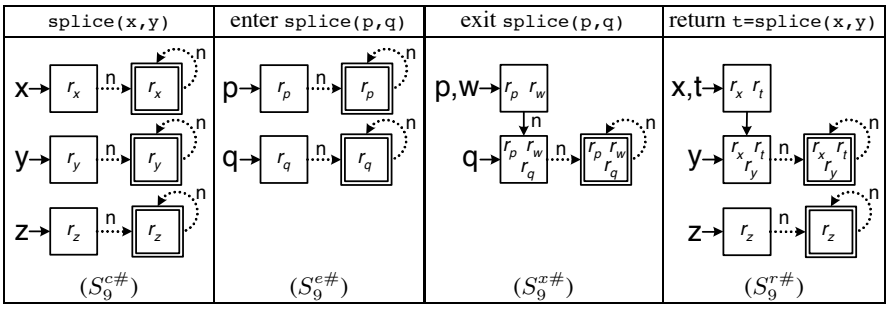

Fig. 9. Abstract states for the invocation `t = splice(x, y);` in the running example.

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 [25]. In our framework, the predicates that record reachability from variables play 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 $S_9^{c\#}$ depicted in Fig. 9, 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.

5.2 Abstract Operational Semantics

The meaning of statements is described by a transition relation $\overset{\sharp}{\rightsquigarrow} \subseteq (\mathit{3Struct} \times \mathit{stms}) \times \mathit{3Struct}$. Because our framework is based on [25], the encoding of the meaning of statements in $\mathcal{ECPF}$ (as transformers of 2-valued structures), 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 our static analysis. In particular, reinterpreting the side conditions of intraprocedural statements conservatively verifies that the *program* is effectively cutpoint-free.

For brevity, we omit the aforementioned encoding from the body of the paper and provide it in Appendix B. We wish to note that all the transformers, including the inter-

procedural operations $Call$ and Ret are specified using predicate-update formulae [9] in first-order logic with transitive closure.

6 Interprocedural Static Analysis

Abstract interpretation of the $\mathcal{ECPF}$ semantics provides the semantic foundations for an interprocedural static-analysis algorithm that computes procedure summaries by tabulating abstract input memory-states to abstract output memory-states. The tabulation is restricted to abstract memory-states that occur in the *analyzed* program. The interprocedural tabulation algorithm is the variant of the IFDS-framework [17] presented in [23], adapted to assume and verify effective cutpoint freedom.

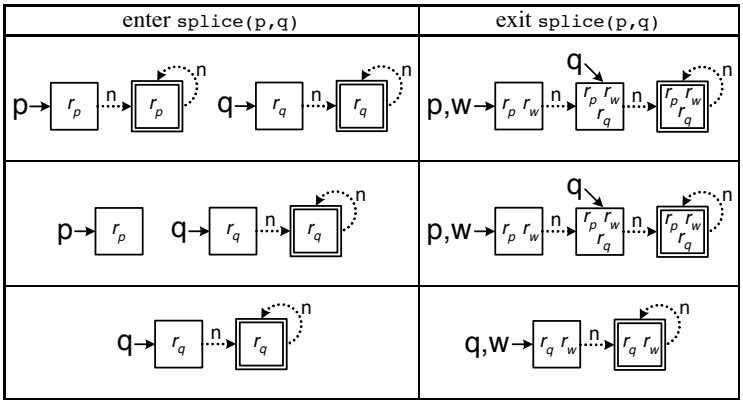

Fig. 10. Partial tabulation of abstract states for the splice procedure.

Example 8. Fig. 10 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 Sec. 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.

Note that this tabulation represents the input/output relation for any call to `splice`, including ones with cutpoints, e.g., the call `s=splice(t, y)` and all recursive calls to `splice` in our running example.

[9] Predicate-update formulae express the semantics of statements: Suppose that σ is a memory state that arises before statement st, that σ' is the store that arises after st is evaluated on σ, and that S is the 2-valued logical structure that encodes σ. A collection of predicate-update formulae–one for each predicate p in the vocabulary of S–allows one to obtain the structure S' that encodes σ'. When evaluated in structure S, the predicate-update formula for a predicate p indicates what the value of p should be in S'. See [25, Observation 2.6]. Evaluation of the predicate-update formulae in 3-valued logic captures the transfer function for st of the abstract semantics. See [25, Observation 2.9].

7 Extensions and Relaxations

In this section, we describe several extensions that use a limited form of annotations on procedures to improve the analysis algorithmic's efficiency, precision, and applicability.

7.1 Blindspots

$\mathcal{ECPF}$ records in every state the value of every formal parameter at the entry to the procedure. This is done to allow the caller to observe the (possibly mutated) part of the heap that was relevant to the callee after the callee returns. However, in certain cases, such observations are not needed or even desirable.

For example, in the program of Fig. 2, the variable `y` is not used after the call `t=splice(x, y)`. Thus, the effort invested to restore its value when the call returns is, for all practical purposes, wasted. Furthermore, direct access to the list returned by `splice` through one of the actual parameters might be considered a form of bad programming. (A clearer example might be a `merge` procedure that merges two sorted lists. When an invocation of `merge` returns, one actual parameter references the head of the list and the other one references one of the list elements. Using the actual parameters at this point makes the code less readable and more sensitive to the implementation details of `merge`. Thus, it is reasonable to expect that the caller uses the returned value, but not the actual parameters.)

Blindspots (for a procedure invocation) are parameter objects for which all the variables and fields pointing to them at the time of the call, excluding fields of relevant objects for the invocation, are *dead* when the procedure returns.[10] $\mathcal{ECPF}$, and its abstract interpretations, can utilize an annotation (e.g., in the form of a subset of the actual/formal parameters) that states which of the parameter objects are blindspots. Such information can improve the efficiency of the analysis algorithim by allowing it to avoid tracking unnecessary information. It also allows verifying good programming style.

For example, Fig. 11 shows the call, entry, exit, and return states that occur in the $\mathcal{ECPF}$ during the invocation `t=splice(x,y)` when both parameter objects are annotated as a *blindspots*. Based on this annotation, the exit state does not record the value of the formal parameters, allowing for more compact summaries. Note that at the return state, `x` and `y` are blocked. As a result, the returned list can be accessed only through `t`.

7.2 Tolerance for a Bounded Number of Cutpoints

$\mathcal{ECPF}$, and its abstract interpretations, *can* allow for procedure invocations to have up to a bounded number of live cutpoints, i.e., cutpoints that are accessed directly by a piercing reference after the procedure returns. The main idea is to treat cutpoints as additional parameters: Every procedure is modified to have k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, the (modified) *semantics* binds the additional parameters with references to the cutpoints.

[10] Note that a blindspot for a procedure invocation is not necessarily a dead *object*.

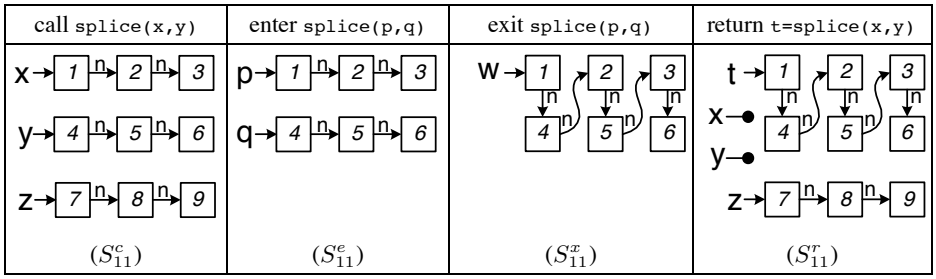

Fig. 11. Concrete states for the invocation `t = splice(x, y)` when both parameter objects are annotated as a *blindspots*.

We can allow for a bounded number of cutpoints by having an annotation regarding the maximal number of allowed cutpoints [11] or by having the user provide a specification (using first-order formulae with transitive closure) of a distinguished set of explicitly-allowed cutpoints. For example, a cutpoint at the last element of a list can be treated differently then other cutpoints.

Fig. 12 depicts the call, entry, exit, and return states that occur in the $\mathcal{ECPF}$ during the invocation `s = splice(t, z)` when procedures are allowed to have at least one cutpoint, or, alternatively, when the second element of the first list is specified as an explicitly-allowed cutpoint. The hidden parameter X_1 gets bound to the cutpoint at the entry state and used to restore the value of `y` at the return state.

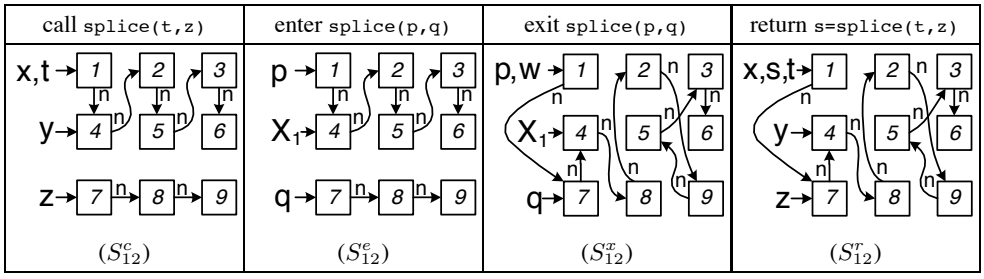

Fig. 12. Concrete states for the invocation `s = splice(t, z)` when one cutpoint is allowed or alternatively, when access path `t.n` is specified as an *explicitly-allowed* cutpoint.

7.3 Restricted Access to the Inaccessible Value

For a program to be effectively cutpoint-free, every piercing reference must not be live at the time of the actual invocation. The reason behind this requirement is to allow the semantics/analysis to avoid maintaining certain aliasing relations, yet still maintain a certain notion of observational soundness with respect to the standard semantics.

[11] This is the essence of the treatment of cutpoints by Gotsman et al. [8].

However, certain *usages* of piercing references are innocuous, i.e., our notion of observational soundness is still maintained as long as programs use piercing references in certain restricted ways. For example, statements such as `x = y`, as well as conditions involving comparisons between •-valued references and `null`, are innocuous. In the former case, the assignment neither affects the control flow of the program nor may lead to a memory fault. In the latter case, it always holds that a •-valued reference is not *null*-valued; thus the condition of the `assume` statement always evaluates to the same value in both semantics.

Effectively, the above observation allows us to relax the requirements of executions to be effectively cutpoint-free: Instead of forbidding all future usages of piercing references (i.e., requiring that they are not live when the invocation of the callee returns), we need only to forbid "effective" future usages of this pointers, i.e., we need only to forbid them from being dereferenced or compared with non-null values both in `assume` statements and in assertions.

7.4 Arbitrary Cutpoints in Pure Procedures

An additional relaxation regarding the requirements of a procedure invocation to be effectively cutpoint-free is possible when a procedure invocation is found to be *pure*. A pure invocation does not modify the shared state. Thus, the abstract representation of the heap at the call site can be reused at the return site. As a result, for reconstructing the layout of the heap, the number of cutpoints is irrelevant, and piercing references do not need to be blocked.

The above approach has one rather significant complication: In case the procedure's return value is a pointer to a heap-allocated object, figuring out which object in the call state corresponds to the one returned by the procedure is not simple. (This complication arises because the abstract semantics does not retain the identity of locations.)

One possible remedy is not to use this relaxation when the return value of the invoked procedure is a (non-null) reference. Another possible remedy is to apply a *meet* operator between the call state and the exit state (after certain renaming operations, similar to the ones used in [11]). We note that the framework of [25] provides an algorithmic meet operator [1]. We also note that (some) information regarding cutpoints can (potentially) make the results of the meet operator more precise.

8 Discussion and Related Work

In this section, we review closely related work.

Rinetzky and Sagiv [22] explicitly represent the runtime stack and abstract 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*.

Jeannet et al. [11] consider 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, this 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 [11] 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. On the other hand, the method of [11] can establish properties of called procedures that our method cannot establish (e.g., that a procedure to reverse a list actually reverses all elements of the list).

Rinetzky et al. [20] present a procedure-local storeless concrete semantics and describe an abstract interpretation of their semantics that can be used for interprocedural shape-analysis for programs manipulating singly linked lists. Their abstract interpretation algorithm explicitly records cutpoint objects in the local heap, and may become imprecise when there is more than one cutpoint. Our algorithm can be seen as a specialization of [20] that provides a partial answer to this problem. In addition, because we restricted our attention to effectively cutpoint-free programs, our semantics and analysis are much simpler than the ones in [20].

In [23], the problem of abstracting cutpoint-induced sharing patterns is addressed by forbidding cutpoints: We developed an interprocedural shape analysis for the class of *cutpoint-free* programs, in which program invocations never generate cutpoints. In the present paper, we extend the framework developed in [23] to a larger class of programs: *effectively cutpoint-free* programs. One can see [23] as an eager form of enforcing effective cutpoint-freedom, while the present paper takes a more lazy approach.

Hackett and Rugina [9] develop a staged analysis to obtain a relatively scalable interprocedural shape analysis. Their 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 because it relies on pointer-expressions that appear in the program's text. The analysis tabulates 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 by Chong and Rugina [4] 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.

Gotsman et al. [8] describe a heap-modular interprocedural shape analysis for singly linked lists that can handle a bounded numbers of cutpoints. The main idea is to treat a bounded number of cutpoint-labels as, essentially, additional parameters: Every procedure can be seen as having k additional (hidden) formal parameters (where k is the bound on the number of allowed cutpoints). When a procedure is invoked, their analysis (non-deterministically) binds these additional parameters with references to the cutpoints. If the procedure has more than k cutpoint, they turn every piercing reference to a dangling pointer, which, essentially, makes the reference inaccessible. Thus, their analysis does not differentiate between dangling references and piercing references.

However, every program that it manages to analyze is a k-cutpoint-tolerant effectively cutpoint-free program.

Yang et al. [28] present a heap-modular interprocedural shape analysis that, similar to [8], is based on a domain of separation-logic formulae. Their experimental results indicate that the use of local heaps provides a speedup of $2-3\times$ in the analysis compared to a global heap analysis. Furthermore, the use of an interprocedural analysis that passes only the reachable portion of the heap was found to be one of the three key reasons for the scalability of their analysis. (The other two key reasons being an efficient join operator and the discard of intermediate states.) In this analysis, cutpoints are passed as additional (hidden) parameters to called procedures, but their number is not bounded. This is one of the possible reasons that their analysis may not terminate (although in many interesting cases it does). In later work [2], the problem of cutpoint abstraction is reduced because the compositional nature of the analysis allows to represent only a subset of the reachable heap.

Marron et al. [14] present a context-sensitive shape analysis that is employed for automatic parallelization of sequential heap manipulating programs. The interprocedural analysis is based on an abstraction of local heaps with cutpoints. The analysis employs an abstraction of cutpoint-labels that uses two main ideas: (i) avoid summarizing cutpoints that are generated by the local variables of the *immediate* caller and (ii) abstract all other cutpoints by recording the set of roots of access paths. The analysis also uses liveness information to avoid recording as cutpoints objects that are only pointed to by dead references.

Rubinstein [24] provides a preliminary study regarding the classification of cutpoints that occur in real-life Java programs. The study is conducted by monitoring program executions. Algorithms for detecting usages of piercing references [12] are presented but not implemented. While the experimental results are non-conclusive, they do indicate that in several interesting cases the unbounded number of cutpoints occur when the program manipulates shared *immutable* data structures. This can motivate special treatment for pure (i.e., readonly) methods (see Sec. 7.4).

A local interprocedural may-alias analysis is given in [7]. 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 [7] and local heap shape analysis see [19].

Local reasoning [10,18] provides a way of proving properties of a procedure independently 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. The $\mathcal{ECPF}$ 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.

[12] The term a *live cutpoint* is used in [24] to refer to an object which gets dereferenced using a piercing reference.

Another relevant body of work is that concerning *encapsulation*, also known as *confinement* or *ownership*. (A review about different encapsulation models can be found in [15]). These works allow modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of these works, as described in [15], is that they all place various restrictions on the kind of sharing allowed 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 *live* sharing across procedure calls.

9 Conclusions and Future Work

In this paper, we presented an interprocedural shape analysis for effectively cutpoint-free programs. The analysis is local in the heap and thus allows reusing the effect of a procedure at different calling contexts. We presented the first non-trivial solution for procedure calls with an unbounded number of cutpoints. The solution is limited because it applies only to pure (read-only) procedures; however, we believe that it opens the door for future work to address the important, and still open, problem of handling an unbounded number of live cutpoints under abstraction.

In general, we believe that the distinction between live piercing references and dead ones can benefit analyses that abstract an unbounded number of cutpoints by allowing them to focus on only abstracting cutpoints that are pointed to by live piercing references. We consider this issue to be future work.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Combining shape analyses by intersecting abstractions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 33–48, 2006.
2. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Symp. on Princ. of Prog. Lang. (POPL)*, pages 289–300. ACM, 2009.
3. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1990.
4. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *International Static Analysis Symposium (SAS)*, 2003.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang. (POPL)*, pages 238–252, New York, NY, 1977. ACM Press.
6. 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.
7. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Conf. on Prog. Lang. Design and Impl. (PLDI)*, 1994.
8. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *International Static Analysis Symposium (SAS)*, 2006.
9. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Symp. on Princ. of Prog. Lang. (POPL)*, 2005.

10. 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.
11. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *International Static Analysis Symposium (SAS)*, 2004.
12. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *Int. Conf. on Comp. Construct. (CC)*, 1992.
13. 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.
14. M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Int. Conf. on Comp. Construct. (CC)*, pages 245–259, 2008.
15. 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.
16. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
17. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symp. on Princ. of Prog. Lang. (POPL)*, 1995.
18. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Symp. on Logic in Computer Science (LICS)*, 2002.
19. N. Rinetzky. *Interprocedural and Modular Local Heap Shape Analysis*. PhD thesis, Tel Aviv University, June 2008.
20. 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.
21. N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav. Modular shape analysis for dynamically encapsulated programs. In *16th European Symposium on Programming (ESOP)*, pages 220–236, 2007.
22. N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Int. Conf. on Comp. Construct. (CC)*, 2001.
23. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *International Static Analysis Symposium (SAS)*, 2005.
24. S. Rubinstein. On the utility of cutpoints for monitoring program execution. Master's thesis, Tel Aviv University, Tel Aviv, Israel, 2006.
25. 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.
26. 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.
27. 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.
28. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Conf. on Computer Aided Verification (CAV)*, pages 385–398. Springer-Verlag, 2008.

A Formal Details Pertaining to the $\mathcal{ECPF}$ Semantics

In this section, we provide the technical details that were glanced over in Sec. 4.

A.1 Reachability

In this section, we give formal definitions for the notions of *reachability*. These definitions are based on the corresponding standard notions in 2-level stores. Intuitively, location l_2 is *reachable from* a location l_1 in a memory state σ if there is a directed path in the heap of σ from l_1 to l_2. A locations l is *reachable* in σ if it is reachable from a location which is referenced by some variable. Note that the inaccessible value, similarly to the *null* value, is not a location.

Definition 1 (Heap path). *A sequence of locations* $\zeta : \{0, \ldots, n \mid n \in \mathcal{N}\} \to Loc$ *is a* directed heap path *in a heap* $h \in \mathcal{H}$*, if for every* $0 \leq i < |\zeta| - 1$ *there exists* $f_i \in \mathcal{F}$ *such that* $h(\zeta(i), f_i) = \zeta(i+1)$*. A directed heap path* ζ goes from l_1*, if* $\zeta(0) = l_1$*, it* goes to l_2 *if* $\zeta(|\zeta| - 1) = l_2$*. A heap path* ζ traverses through l *if there exists* i *such that* $0 \leq i < |\zeta|$ *and* $l = \zeta(i)$.

Definition 2 (Reachability). *A location* l_2 *is* reachable from *a location* l_1 *in a memory state* $\sigma = \langle \rho, L, h \rangle$*, if there is a directed heap path in* h *going from* l_1 *to* l_2.

Definition 3 (Reachable locations). *A locations* l *is* reachable *in* σ *if it is reachable from a location which is referenced by some variable. We denote the set of* reachable locations *in* $\sigma \in \Sigma$ *by* $\mathcal{R}(\sigma)$*, i.e.,* $\mathcal{R}(\sigma) = \{l \in L | x \in \mathcal{V}$ *and* l *is reachable in* σ *from* $\rho(x) \in Loc\}$.

A.2 Properties of the $\mathcal{ECPF}$ Semantics

In this section, we formally define the notions of *observational soundness* and of *simulation* between the $\mathcal{ECPF}$ semantics and the standard semantics. To be precise, when referring to the standard semantics we refer to the standard store-based semantics $\mathcal{GSB}$ defined in [19,20]. In short, memory states in $\mathcal{GSB}$ are represented in the same way as memory states in $\mathcal{ECPF}$. The main difference between $\mathcal{GSB}$ and $\mathcal{ECPF}$ is that the operational semantics never blocks references in $\mathcal{GSB}$, and thus $\bullet$ is not a possible value.

Access paths 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 4 (Field Paths). *A* field path $\delta \in \Delta = \mathcal{F}^*$ *is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by* ϵ.

Definition 5 (Access path). *An* access path $\alpha = \langle x, \delta \rangle \in AccPath = \mathcal{V} \times \Delta$ *is a pair consisting of a local variable and a field path.*

Definition 6 (Access path value in the $\mathcal{ECPF}$ semantics). *The value of an access path* $\alpha = \langle x, \delta \rangle$ *in state* $\sigma = \langle \rho, L, h \rangle$ *of the* $\mathcal{ECPF}$ *semantics, denoted by* $[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma)$*, is defined to be* $\hat{h}(\rho(x), \delta)$*, where*

$$\hat{h}\colon Val \times \Delta \rightharpoonup Val \text{ such that}$$

$$\hat{h}(v, \delta) = \begin{cases} v & \text{if } \delta = \epsilon \quad \text{(note that } v \text{ might be } \bullet\text{)} \\ \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta',\ v \in Loc \\ undefined & \text{otherwise} \quad \text{(note that } v \text{ might be } \bullet\text{)} \end{cases}$$

Note that an access to a field of the inaccessible value is not defined.

Definition 7 (Comparable values). *A pair of values of the $\mathcal{ECPF}$ semantics $v_1, v_2 \in Val$ are* comparable, *denoted by $v_1 \overset{?}{\bowtie} v_2$, if $v_1 \neq \bullet$ and $v_2 \neq \bullet$.*

Definition 8 (Access path value in the $\mathcal{GSB}$ semantics). *The value of an access path $\alpha = \langle x, \delta \rangle$ in state $\sigma_G = \langle \rho, L, h \rangle$ of the $\mathcal{GSB}$ semantics, denoted by $[\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G)$, is defined to be $\overline{h}(\rho(x), \delta)$, where $Val_G = Val \setminus \{\bullet\}$ and*

$$\overline{h}\colon Val_G \times \Delta \rightharpoonup Val_G \text{ such that}$$
$$\overline{h}(v,\delta) = \begin{cases} v & \text{if } \delta = \epsilon \\ \overline{h}(h(v,f),\delta') & \text{if } \delta = f\delta',\ v \in Loc \\ undefined & \text{otherwise} \end{cases}$$

Observational soundness We define the notion of observational soundness between a $\mathcal{ECPF}$ memory state σ and a standard 2-level store σ_G of the $\mathcal{GSB}$ semantics as the preservations in σ_G of all equalities and inequalities which hold in σ.[7] Note that the preservation in the other direction is not required. Also note that an equality resp. inequality of values of access paths holds in σ only when the two access paths have comparable values. For simplicity, we define $[\![\texttt{null}]\!]_{\mathcal{ECPF}}(\sigma) = [\![\texttt{null}]\!]_{\mathcal{GSB}}(\sigma) = \mathit{null}$.

Definition 9 (Observational soundness). *The memory state $\sigma \in \Sigma$ is* observationally sound *with respect to memory state $\sigma_G \in \Sigma_G$, denoted by $\sigma_G \leqq \sigma$, if for every $\alpha, \beta \in AccPath \cup \{\texttt{null}\}$ it holds that*

if $[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma) \overset{?}{\bowtie} [\![\beta]\!]_{\mathcal{ECPF}}(\sigma)$ *then*
$[\![\alpha]\!]_{\mathcal{ECPF}}(\sigma)) = [\![\beta]\!]_{\mathcal{ECPF}}(\sigma) \Leftrightarrow [\![\alpha]\!]_{\mathcal{GSB}}(\sigma_G) = [\![\beta]\!]_{\mathcal{GSB}}(\sigma_G)$

We define the notion of observational soundness between two $\mathcal{ECPF}$ memory states (resp. two standard memory states) in a similar manner.

Simulation Before we define the notion of simulation we briefly review some execution traces accessing-functions (formally defined in [19]). Given an execution trace π, the initial resp. final memory state of an execution trace π, denoted by $in(\pi)$ resp. $out(\pi)$, is the current memory state in the first resp. last stack of program states. $\pi(i)$ returns the stack at the ith step of the execution and $|\pi(i)|$ returns its height. $path(\pi)$ is the sequence of program points which the execution traverses. *i.e.*, $path(\pi)(i)$ is the program point in the ith step of the execution. (We assume that every statement is labeled by a program point.)

The following theorem shows that $\mathcal{ECPF}$ simulates the standard semantics. In the lemma, we denote by $path(\pi)$ the sequence of intraprocedural statements and $Call$ and $Return$ operations executed in π. We also use $[\pi]_k$ to denote the memory state of the current procedure at $\pi(k)$, the kth program state of π

Theorem 1 (Simulation). *Let P be an effectively cutpoint-free program according to the $\mathcal{ECPF}$ semantics. Let π_S be a trace of a program P according to the standard semantics. There exists a trace π_E of P according to the $\mathcal{ECPF}$ semantics such that the folowing holds (i) $|\pi_S| = |\pi_E|$, (ii) $path(\pi_S) = path(\pi_E)$, and (iii) $[\pi_S]_k \leqq [\pi_E]_k$ for every $0 \leq k < |\pi_S|$.*

Sketch of Proof: The proof is done by induction on the length of the execution. We look at 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. The graph edges are labeled by field names.

We prove that observational equivalence is preserved by showing a stronger property: every memory state $[\pi_E]_k$ produced by the $\mathcal{ECPF}$ can be seen as a subgraph of $[\pi_S]_k$, the corresponding memory state of the $\mathcal{GSB}$ semantics. Furthermore, that two graphs agree on the values of live references.

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 a memory state of the $\mathcal{GSB}$ to the set of objects in the corresponding memory state of the $\mathcal{ECPF}$ 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 that were irrelevant for the invocation remains as in the call site. The mapping for locations 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.

To show that the return memory state produced by the $\mathcal{ECPF}$ semantics is a subgraph of the corresponding return memory state of the $\mathcal{GSB}$ semantics agrees with it on the values of live references, we make the following argument: The computation of return states in the $\mathcal{ECPF}$ semantics blocks piercing references. The computation of the return states in the $\mathcal{GSB}$ semantics does not. Thus, it remains to show that all the references that gets blocked by the $\mathcal{ECPF}$ semantics are not live in the $\mathcal{GSB}$ semantics.

The computation of return states in the $\mathcal{ECPF}$ semantics restores all references from the caller's local heap to parameter objects which, by the induction assumption, must be in the ϱ relation. It only blocks the value of piercing references (i.e., it changes the value of every pointer field or variable pointing to a cutpoint). The execution π_S never uses a a field f of an object o such that the f-field in $\varrho(o)$ at the corresponding $\mathcal{ECPF}$ points to the inaccessible location. Otherwise, π_E is a non effectively cutpoint-free execution of P in $\mathcal{ECPF}$ which is a contradiction to the assumption that P is effectively cutpoint-free. For similar reasons, the value of a variable which gets blocked by the $\mathcal{ECPF}$ semantics does not get used by the $\mathcal{GSB}$ semantics.

Lemma 1. *Let P be an effectively cutpoint-free program. The following holds:*

[Invariants] *An invariant concerning equality of values of access paths in the $\mathcal{ECPF}$ semantics is an invariant in the standard semantics*

[Cleanness] *P does not dereferences null references in the standard semantics.*

Lemma 2. *Let P be an effectively cutpoint-free program. A reference, that at a given program point always has the inaccessible value, is not live at that program point in the standard semantics.*

Definition 10 (Observational equivalence). *The $\mathcal{ECPF}$ memory states $\sigma_1, \sigma_2 \in \Sigma$ are* observationally equivalent*, denoted by $\sigma_1 \lessapprox \sigma_2$, if $\sigma_1 \leqq \sigma_2$ and $\sigma_2 \leqq \sigma_1$.*

The following lemma shows that $\mathcal{ECPF}$ is indifferent to location names.

Theorem 2 (Indifference to location names). *Let π_1, π_2 be execution traces of a program P according to the $\mathcal{ECPF}$ semantics. If $|\pi_1(1)| = |\pi_2(1)| = 1, in(\pi_1) \lessapprox in(\pi_2)$ and $path(\pi_1) = path(\pi_2)$ then $out(\pi_1) \lessapprox out(\pi_2)$.*

B Update Formulae

In this section, we encode the abstract transformers using the notations of [25].

B.1 Intraprocedural Statements

The meaning of assignments is specified by defining the values of the predicates in the outgoing structure using first-order logic formulae with transitive closure over the incoming structure [25]. The inference rules for assignments are rather straightforward. We encode conditional using `assume()` 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 Fig. 13. The table also specifies the side condition which enables that application of the statement. These conditions check that null-dereference is not performed and that the *inaccessible value is not used*. 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 [25]: 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.

The semantics transitions into the error state ($\sigma_\bullet$) under the same conditions as the $\mathcal{ECPF}$ semantics, i.e., when an inaccessible-valued variable or field are accessed. (See Fig. 7). The following side condition trigers such a transition when a variable `x` points to an inaccessible location $\exists v\colon x(v) \wedge ia(v_2)$. Similarly, the following side condition trigers such a transition when the `f`-field of the object pointed to by a variable `x` points to an inaccessible location $\exists v_1, v_2\colon x(v_1) \wedge f(v_1, v_2) \wedge ia(v_2)$.

B.2 Interprocedural Statements

The treatment of procedure call and return could be briefly described as follows: (i) constructing the memory state at the callee's entry site (S_e) and (ii) 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

Statement	Predicate-update formulae	$\mathbf{side-condition}$
`y` $=$ `null`	$y'(v) = 0$	
`y` $=$ `x`	$y'(v) = x(v)$	$\forall v_1 \colon \neg(x(v_1) \wedge ia(v_1))$
`y` $=$ `x.f`	$y'(v) = \exists v_1 \colon x(v_1) \wedge f(v_1, v)$	$\exists v_1 \colon x(v_1) \wedge \neg ia(v_1) \wedge$ $\forall v_2 \colon \neg(x(v_1) \wedge f(v_1, v_2) \wedge ia(v_2))$
`y.f` $=$ `null`	$f'(v_1, v_2) = f(v_1, v_2) \wedge \neg y(v_1)$	$\exists v_1 \colon y(v_1) \wedge \neg ia(v_1)$
`y.f` $=$ `x`	$f'(v_1, v_2) = f(v_1, v_2) \vee (y(v_1) \wedge x(v_2))$	$\exists v_1 \colon y(v_1) \wedge \neg ia(v_1) \wedge$ $\forall v_2 \colon \neg(x(v_2) \wedge ia(v_2))$
`y` $=$ `alloc`	$eq'(v_1, v_2) = eq(v_1, v_2) \vee new(v_1) \wedge new(v_2)$ $new'(v) = 0$	

Fig. 13. The predicate-update formulae defining the operational semantics of assignments. Note that we always assume that a reference variable is nullified before re-assigned.

construct the caller's memory state at the return site (S_r). We now formally define and explain these steps.

Fig. 14 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.

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. It also contains the individual representing the inaccessible value. The formal parameters are initialized by $updCall_q^{y=p(x_1,\ldots,x_k)}$, defined in Fig. 15(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.

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 effectively cutpoint-free guarantees that the only live 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,

Table 3. Formulae shorthands and their intended meaning.

Shorthand	**Formula**	**Intended Meaning**
$F(v_1, v_2)$	$\bigvee_{f \in \mathcal{F}} f(v_1, v_2)$	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
$R_{\{x_1,\ldots,x_k\}}(v)$	$\neg ia(v) \wedge$ $\bigvee_{x \in \{x_1,\ldots,x_k\}} \exists v_1 \colon x(v_1) \wedge F^*(v_1, v)$	v is reachable from $\mathtt{x}_1$ or $\mathtt{x}_2$ or ... or $\mathtt{x}_k$
$isCP_{q,\{x_1,\ldots,x_k\}}(v)$	$R_{\{x_1,\ldots,x_k\}}(v) \wedge$ $(\neg x_1(v) \wedge \ldots \wedge \neg x_k(v)) \wedge$ $(\bigvee_{y \in V_q} y(v) \vee$ $\exists v_1 \colon \neg R_{\{x_1,\ldots,x_k\}}(v_1) \wedge F(v_1, v))$	v is a cutpoint

$$Call_{y=p(x_1,\ldots,x_k)}(S_c) = S_e \qquad Ret_{y=p(x_1,\ldots,x_k)}(S_c, S_x) = S_r$$

where

$S_e = \langle U_e, \iota_e \rangle$ where
$U_e = \{u \in U^{S_c} \mid S_c \models R_{\{x_1,\ldots,x_k\}}(u) \vee ia(v)\}$
$\iota_e = updCall_q^{y=p(x_1,\ldots,x_k)}(S_c)$

$S_r = \langle U_r, \iota_r \rangle$ where
$Let\ \ U' = \{u.c \mid u \in U_c\} \cup \{u.x \mid u \in U_x\}$

$$\iota' = \lambda p \in \mathcal{P}. \begin{cases} \iota_c[inUc \mapsto \lambda v.1](p)(u_1, \ldots, u_m) & : \quad u_1 = w_1.c, \ldots, u_m = w_m.c \\ \iota_x[inUx \mapsto \lambda v.1](p)(u_1, \ldots, u_m) & : \quad u_1 = w_1.x, \ldots, u_m = w_m.x \\ 0 & : \quad otherwise \end{cases}$$

$in\ \ U_r = \{u \in U' \mid \langle U', \iota' \rangle \models inUx(u) \vee (inUc(u) \wedge \neg ia(u) \wedge \neg R_{\{x_1,\ldots,x_k\}}(u))$
$\iota_r = updRet_q^{y=p(x_1,\ldots,x_k)}(\langle U', \iota' \rangle)$

$Call_{y=p(x_1,\ldots,x_k)}(S_c) = \sigma_\bullet \qquad S_c \models \exists v \colon ia(v) \wedge (x_1(v) \vee \cdots \vee x_k(v))$
$Ret_{y=p(x_1,\ldots,x_k)}(S_c, S_x) = \sigma_\bullet \qquad S_x \models \exists v \colon ia(v) \wedge ret(v)$

Fig. 14. 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 Fig. 15.

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; (iii) block every piercing reference.

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 Fig. 15(b). The most interesting aspect of these update-formulae is the formula

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

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

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

$$z'(v) = \begin{cases} ret_p(v) & : \quad z = y \\ inUc(v) \wedge z(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee & : \quad z \in V_q \setminus \{y\} \\ \quad \exists v_1 \colon z(v_1) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \vee & \\ \quad\quad \exists v_1 \colon z(v_1) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_1) \wedge inUx(v) \wedge ia(v) & \\ 0 & : \quad z \in \mathcal{V} \setminus V_q \end{cases}$$

$$f'(v_1, v_2) = inUx(v_1) \wedge inUx(v_2) \wedge f(v_1, v_2) \vee$$
$$inUc(v_1) \wedge inUc(v_2) \wedge f(v_1, v_2) \wedge \neg ia(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee$$
$$inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} \colon f(v_1, v_{sep}) \wedge match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_{sep}, v_2) \vee$$
$$inUc(v_1) \wedge inUx(v_2) \wedge \exists v_{sep} \colon f(v_1, v_{sep}) \wedge isCP_{q,\{x_1,\ldots,x_k\}}(v_{sep}) \wedge ia(v_2)$$
$$inUc'(v) = inUx'(v) = 0$$

Fig. 15. 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 $\mathtt{z} \in \mathcal{V}$ and for every field $\mathtt{f} \in \mathcal{F}$.

$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}$, defined below:

$$match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v_2) \stackrel{\text{def}}{=} \begin{array}{l} inUc(v_1) \wedge ia(v_1) \wedge inUx(v_2) \wedge ia(v_2) \vee \\ \bigvee_{i=1}^{k} inUc(v_1) \wedge x_i(v_1) \wedge inUx(v_2) \wedge h_i(v_2) \end{array}$$

This formula matches an individual that represents a (parameter) 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. The 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 originating from S_c and representing relevant objects are removed, resulting with the updated memory state of the caller. In addition, the formula matches the individual representing the inaccessible value at the call site with the one representing the inaccessible value at the return site, thus preserving the value of inaccessible references from before the call.

We block piercing references using formula $isCP_{q,\{x_1,\ldots,x_k\}}(v)$, defined in Tab. 3. The formula holds when v is a cutpoint object. It 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.

Predicate update formulae for instrumentation predicates. Fig. 16 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 $X \subseteq \mathcal{V}$. We use $bypass_X(v_1, v_2)$ as a shorthand for $(F(v_1, v_2) \wedge \neg R_X(v_1))^*$. 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 reachable from any variable in $X \subseteq \mathcal{V}$. Note that, again, 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 $updCall_q^{y=p(x_1,\ldots,x_k)}$

$$ils'(v) = ils(v) \wedge \neg(PT_{x_1,\ldots,x_k}(v) \vee isCP_{q,\{x_1,\ldots,x_k\}}(v)) \vee \exists v_1, v_2 \colon R_{\{x_1,\ldots,x_k\}}(v_1) \wedge R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_1, v) \wedge F(v_2, v) \wedge \neg eq(v_1, v_2))$$

$$r'_y(v) = \begin{cases} r_{x_i}(v) & : \quad y = h_i \\ 0 & : \quad y \in \mathcal{V} \setminus \{h_1, \ldots, h_k\} \end{cases}$$

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

$$\begin{aligned}
ils'(v) &= ils(v) \wedge (inUc(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee inUx(v)) \vee \\
&\quad PT_{x_1,\ldots,x_k}(v) \wedge \exists v_1, v_2, v_3 \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_1, v) \wedge \neg eq(v_2, v_3) \wedge \\
&\quad\quad inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \wedge F(v_2, v_1) \wedge \\
&\quad\quad\quad (inUc(v_3) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_3) \wedge F(v_3, v_1) \vee inUx(v_3) \wedge F(v_3, v)) \\
r'_{obj}(v_1, v_2) &= r_{obj}(v_1, v_2) \wedge inUx(v_1) \wedge inUx(v_2) \vee \\
&\quad r_{obj}(v_1, v_2) \wedge inUc(v_1) \wedge inUc(v_2) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v_2) \vee \\
&\quad\quad inUc(v_1) \wedge inUx(v_2) \wedge \exists v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\quad\quad\quad bypass_{\{x_1,\ldots,x_k\}}(v_1, v_a) \wedge r_{obj}(v_f, v_2) \\
r'_x(v) &= inUc(v) \wedge r_x(v) \wedge \neg R_{\{x_1,\ldots,x_k\}}(v) \vee \\
&\quad\quad inUx(v) \wedge \exists v_x, v_a, v_f \colon match_{\{\langle h_1,x_1\rangle,\ldots,\langle h_k,x_k\rangle\}}(v_a, v_f) \wedge \\
&\quad\quad\quad x(v_x) \wedge bypass_{\{x_1,\ldots,x_k\}}(v_x, v_a) \wedge r_{obj}(v_f, v)
\end{aligned}$$

Fig. 16. 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 $h_1, \ldots, h_k$.