A Semantics for Procedure Local Heaps and its Abstractions

AVACS-TR-1

Noam Rinetzky[1*] Jörg Bauer[2†] Thomas Reps[3‡]
Mooly Sagiv[1‡] Reinhard Wilhelm[2]

[1] School of Comp. Sci.; Tel Aviv Univ.; Tel Aviv 69978; Israel.
{*maon,msagiv*}*@post.tau.ac.il*
[2] Informatik; Univ. des Saarlandes; Saarbrücken, Germany.
{*joba,wilhelm*}*@cs.uni-sb.de*
[3] Comp. Sci. Dept.; Univ. of Wisconsin; Madison, WI 53706; USA.
reps@cs.wisc.edu

Abstract

The goal of this work is to develop compile-time algorithms for automatically verifying properties of imperative programs that manipulate dynamically allocated storage. The paper presents an analysis method that uses a characterization of a procedure's behavior in which parts of the heap not relevant to the procedure are ignored. The paper has two main parts: The first part introduces a non-standard concrete semantics, $\mathcal{LSL}$, in which called procedures are only passed *parts* of the heap. In this semantics, objects are treated specially when they separate the "local heap" that can be mutated by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. The second part concerns abstract interpretation of $\mathcal{LSL}$ and develops a new static-analysis algorithm using canonical abstraction. It also provides insight into Deutsch's may-alias algorithm.

*Supported in part by a grant from the the Israeli Academy of Science.

†Supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

‡Supported by the office of Naval Research under contract N00014-01-1-0796.

Contents

1 Introduction **4**
- 1.1 Store-based vs. Storeless Semantics 4
- 1.2 Main Results 5
- 1.3 Outline 6

2 Preliminaries **6**
- 2.1 Syntax of **EAlgol** 6
- 2.2 Running Example 6
- 2.3 Global-Heap Store-Based Semantics 7
- 2.4 Observable Properties 9

3 Cutpoints and their Use **11**

4 The Localized-Heap Storeless Semantics and its Properties **14**
- 4.1 The Localized-Heap Storeless Semantics 14
  - 4.1.1 Memory States 14
  - 4.1.2 Inference Rules 16
- 4.2 Properties of the Semantics 22
  - 4.2.1 Semantic Equivalence 23
  - 4.2.2 Standard Properties 24
  - 4.2.3 Modularity 24
- 4.3 Assertion Language 25

5 Abstract Interpretation **28**
- 5.1 The May-Alias Abstraction 29
- 5.2 Interprocedural Shape Analysis with Local-Heaps 29
  - 5.2.1 Representing $\mathcal{LSL}$ Memory States by 3-Valued Logical Structures 30
  - 5.2.2 Abstract Interpretation 35
  - 5.2.3 Discussion 37

6 Related Work **37**
- 6.1 Storeless Semantics 37
- 6.2 Interprocedural Shape Analysis 38
- 6.3 Local Reasoning 38
- 6.4 Encapsulation 39
- 6.5 Rule of Adaptation 39

7 Conclusions **40**

A Additional Code **47**

B The May-Alias Abstraction **49**

C Proofs **50**
C.1 Properties of the $\mathcal{GSB}$ Semantics 51
C.2 Properties of the $\mathcal{LSL}$ Semantics 52
C.3 Context-Aware Equivalence 57

1 Introduction

The long-time research goal of our work is to develop compile-time algorithms for automatically verifying properties of imperative programs that manipulate dynamically allocated storage. The goal is to verify properties such as the absence of null dereferences, the absence of memory leaks, and the preservation of data-structure invariants. The ability to reason about the effects of procedure calls is a crucial element in program verification, program analysis, and program optimization. This paper presents an approach to the modular analysis of imperative languages with procedures and dynamically allocated storage, based on an abstract interpretation of a novel non-standard storeless semantics.

1.1 Store-based vs. Storeless Semantics

A straightforward way to specify semantics of programs with dynamically allocated objects and pointers is by a store-based operational semantics, e.g., see [34]. This semantics is very natural because it closely corresponds to concepts of the machine architecture. Moreover, it is possible to compute the effect of a procedure on a large heap from its effect on subheaps. This is the semantic basis for O'Hearn's "frame rule" [22,34], which uses assertions about disjoint parts of the heap: the post condition of a procedure call is inferred by combining assertions that hold before the call with ones that characterize the effect of the procedure call.

In programming languages such as Java, where addresses cannot be used explicitly (in contrast to C's `cast` statements), it is possible to represent states in a more abstract way because any two heaps with isomorphic reachable parts are indistinguishable. In particular, garbage cells have no significance. This leads to the notion of storeless semantics, which was pioneered by [24]. There, states are represented as aliases between pointer access paths.

A first step in many heap-abstractions is to abstract away from specific memory addresses, e.g., [13,15,23,38,40,41]. A storeless *concrete* semantics has already done this step, which relieves the designer of an abstraction from having to do it. Thus, it is natural to base powerful pointer (shape) analysis algorithms on storeless semantics. Unfortunately, existing storeless semantics associate the entire heap with each procedure invocation and class instantiation, which makes it difficult to support procedure and data abstraction. Another problem with storeless semantics is that it is hard to relate properties of memory cells before and after a call. As a result, it is hard to scale these methods to prove properties of real-life programs. By "scaling", we mean not just cost issues but also precision. In particular, after a procedure call some information about the calling context may be lost.

In this paper, we present a first step towards addressing the aforementioned scaling issues by (i) developing a storeless semantics that allows representation of parts of the heap *and* relating properties before and after a call, and (ii) presenting an abstraction of this semantics.

1.2 Main Results

In this paper, we develop a method to characterize a procedure's behavior in a way that ignores parts of the heap that are not relevant to the procedure. Toward this end, the paper introduces a non-standard storeless *concrete* semantics, $\mathcal{LSL}$, for *Localized-heap Store-Less*. In this semantics, a called procedure is only passed a *part* of the heap. Based on this semantics, a new static-analysis algorithm is developed using canonical abstraction [40]. This allows us to prove properties of programs that were not automatically verified before. We believe that the modular treatment of the heap will allow the implementation of these abstractions to scale better on larger code bases. The approach also provides insights into Deutsch's may-analysis algorithm [15].

The paper has two main parts: The first part (Sec. 4) concerns $\mathcal{LSL}$, the non-standard concrete storeless semantics. The second part (Sec. 5) concerns abstract-interpretation of this semantics.

$\mathcal{LSL}$ is based on the following ideas: Objects in the heap reachable from an actual parameter are treated differently when they separate the "local heap" that can be accessed by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. We call these objects *cutpoints*. An object *belongs to the local-heap* when it is reachable from a procedure's actual parameters. Such an object is a *cutpoint* when it is reached via a pointer-access path that starts at a variable of a *pending* call and does not *traverse* the local-heap. When a procedure returns, the cutpoints are used to update the caller's local-heap with the effect of the call. Because our goal is to perform static analysis, $\mathcal{LSL}$ is a *storeless semantics* [24]; every dynamically allocated object o is represented by the set of *access paths* that reach o. In particular, unreachable objects are not represented. $\mathcal{LSL}$ is different from previous storeless semantics based on pointer-access paths [13, 41] in the following way. It does not represent access paths that start from variables of pending calls in the "local state" of the current procedure. This means that a procedure has a local view that only includes objects that are reachable from the procedure's parameters and, in addition, any objects that it allocates.

We characterize the manner in which $\mathcal{LSL}$ simulates a standard store-based semantics and identify a class of observations for which $\mathcal{LSL}$ is equivalent to the standard store-based semantics. This allows us to prove properties ranging from the absence of runtime errors to partial and total correctness with respect to the standard store-based semantics. We study the properties of $\mathcal{LSL}$. In particular, we show that it has a number of standard properties including full abstraction and determinism.

The second part of the paper uses $\mathcal{LSL}$ as the starting point for static-analysis algorithms that treat the heap in a more local, more modular way than previous work. In this part of the paper, we make the following contributions:

- $\mathcal{LSL}$ provides insight into previous work on static may-alias analyses based on pointer-access paths [15]—in particular, the treatment of variables of pending calls, which is one of the most complicated aspects of [15]. For instance, a surprising aspect of the method given in [15] is that recursive procedures are handled in a more precise way than loops. The intuitive reason is that the abstractions of values of variables in the current procedure is different from the abstraction used for values of variables in pending procedures. Specifically, we show that the abstract domain used in [15] is an abstraction of $\mathcal{LSL}$.

- Using an abstraction of $\mathcal{LSL}$, we present a new interprocedural shape-analysis algorithm for programs that manipulate dynamically allocated storage. This allows us to prove properties of programs that were not automatically verified before (e.g., destructive merge of two singly-linked lists by a recursive procedure, see Fig. 21). Furthermore, the analysis is done in a way that is more likely to scale up. In particular, our analysis benefits from the fact that the heap is localized: the behavior of a procedure only depends on the contents of its local-heap. This allows analysis results to be reused for different contexts.

1.3 Outline

The remainder of the paper is organized as follows: Sec. 2 sets the scene by defining **EAlgol**, a simple imperative language, and defining its standard store-based semantics. It also introduces our running example. Sec. 3 defines cutpoints and describes their use in $\mathcal{LSL}$. Sec. 4 defines $\mathcal{LSL}$ semantics for **EAlgol** and states its properties. Sec. 5 presents the two aforementioned abstractions of $\mathcal{LSL}$. Sec. 6 reviews closely related work. Sec. 7 concludes our work.

2 Preliminaries

In this section, we introduce a simple imperative language called **EAlgol**. We define its standard semantics, which is operational, large-step, store-based (as opposed to storeless), and global, i.e., the entire heap is passed to a procedure. We refer to this semantics as $\mathcal{GSB}$, for *Global-heap Store-Based*.

2.1 Syntax of EAlgol

Programs in **EAlgol** consist of a collection of functions including a `main` function. The programmer can also define her own types (à la `C` structs) and refer to heap-allocated objects of these types using pointer variables. Parameters are passed by value. Formal parameters cannot be assigned to. Functions return a value by assigning it to a designated variable `ret`.

The syntax of **EAlgol** is defined in Fig. 1. The notation $\bar{z}$ denotes a sequence of z's. We define the syntactic domains $x, y \in \mathit{VarId}$, $f \in \mathit{FieldId}$, $p \in \mathit{FuncId}$, $t \in \mathit{TypeId}$, and $lb \in \mathit{Labels}$ of variables, field names, functions identifiers, type names, and program-labels, respectively. For a function p, V_p denotes the set of its local variables and F_p denotes the set of its formal parameters. We assume $F_p \subseteq V_p$ and that all the variables in $V_p \setminus F_p$ are declared at the beginning of a function declaration.

2.2 Running Example

The **EAlgol** program shown in Fig. 2 is our running example. The program consists of a type definition for an element in a linked list (`Sll`); three list-manipulating functions:

$$
\begin{array}{llcll}
P & \in & \mathit{prog} & ::= & \overline{\mathit{rcdecl}}\,\overline{\mathit{fndecl}} \\
 & & \mathit{rcdecl} & ::= & \texttt{record}\,t := \{\,\overline{\mathit{tname}\,f}\} \\
 & & \mathit{tname} & ::= & \texttt{int} \mid t \\
 & & \mathit{fndecl} & ::= & \mathit{tname}\;p(\overline{\text{tname}\;x}) := \overline{\mathit{vdecl}}\;\mathit{st} \\
 & & \mathit{vdecl} & ::= & \mathit{tname}\;\text{VarId} \\
\mathit{st} & \in & \mathit{stms} & ::= & x{=}c \mid x{=}y \mid\; x{=}y\;\mathit{op}\;z \mid x{=}y.f \mid \\
 & & & & x.f{=}\;\texttt{null} \mid x.f{=}y \mid x = \texttt{alloc}\,t \mid \\
 & & & & y{=}p(\overline{x}) \mid \mathit{lb}:\;\mathit{st} \mid \texttt{while}\;(\mathit{cnd})\;\texttt{do}\;\mathit{st}\;\texttt{od} \mid \\
 & & & & \mathit{st}\,;\mathit{st} \mid \texttt{if}\;(\mathit{cnd})\;\texttt{then}\;\mathit{st}\;\texttt{else}\;\mathit{st}\;\texttt{fi} \\
 & & \mathit{cnd} & ::= & x == y \mid x\,!= y \mid x == c \mid x\,!= c \\
c & \in & \mathit{const} & ::= & \texttt{null} \mid \texttt{n}
\end{array}
$$

Figure 1: Syntax of **EAlgol**.

create (`crt`), destructive append (`app`), and destructive reverse (`reverse`); and a `main` function.

The program allocates three acyclic linked lists. It then destructively appends the list pointed-to by `t2` to the tails of the lists pointed-to by `t1` and `t3`. As a result, at program point lb_c, just before `reverse` is invoked, `x` points-to an acyclic list with five elements, `z` points-to an acyclic list with five elements, and the two lists share their last two elements as a common tail.

The invocation of `reverse`, which is the core of our running example, (destructively) reverses the list passed as an argument. As a result, at lb_r, `reverse`'s return-site, `y` points-to the head of the reversed-list. Note that the shared tail of the list pointed-to by `z` has also changed.

2.3 Global-Heap Store-Based Semantics

We now define the $\mathcal{GSB}$ semantics for **EAlgol**. For simplicity, the semantics tracks only pointer values and assumes that every pointer-valued field or variable is assigned `null` before being assigned a new value.[1] In addition, we assume that before a function terminates it assign a `null` value to every pointer variable that is not a formal parameter.[2]

Fig. 3 defines the semantic domains. *Loc* is an unbounded set of memory locations. A *memory state* for a function p, $\sigma_G^p \in \Sigma_G^p$, keeps track of the allocated memory locations, L, an environment mapping p's local variables to values, ρ, and a mapping

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

[2]These conventions simplify the definition of both $\mathcal{GSB}$ semantics and $\mathcal{LSL}$; in principle, different ones could be used with minor effects on the capabilities of our approach. For clarity, our example programs do not adhere to these restrictions.

```
record Sll := { Sll n; int d }
Sll reverse(Sll h):= lbe:
      Sll p,q,t;
      p=h;
      while (p!=null) do
        q=p.n; p.n=t; t=p; p=q od;
      ret = t lbx:
int main():=
      Sll x,y,z,t1,t2,t3;
      t1=crt(3); t2=crt(2); t3=crt(3);
      x=app(t1,t2);
      z=app(t3,t2);
      t1=null; t2=null; t3=null;
lbc:  y = reverse(x); lbr:
      ret=0
```

Figure 2: The running example. The code of functions `crt` and `app` appears in App. A.

l	$\in$	Loc
v	$\in$	$\mathit{Val} = \mathit{Loc} \cup \{\mathit{null}\}$
ρ	$\in$	$\mathit{Env}_p = V_p \to \mathit{Val}$
h	$\in$	$\mathit{Heap}_G = \mathit{Loc} \times \mathit{FieldId} \to \mathit{Val}$
$\sigma_G, \langle L, \rho, h\rangle$	$\in$	$\Sigma^p_G = 2^{\mathit{Loc}} \times \mathit{Env}_p \times \mathit{Heap}_G$

Figure 3: Semantic domains of the $\mathcal{GSB}$ semantics.

from fields of *allocated* locations to values, h. Due to our simplifying assumptions, a value is either a memory location or *null*.

The meaning of statements is described by a transition relation $\overset{G}{\rightsquigarrow} \subseteq (\sigma_G \times \mathit{stms}) \times \sigma_G$. Fig. 4 shows the *axioms* for assignments. The *inference rule* for function calls is given in Fig. 5. All other statements are handled as usual using a two-level store semantics for pointer languages.

Example 2.1 The memory state at lb_c, the call-site to `reverse`, is depicted graphically in Fig. 6 (labeled σ^c_G). Allocated locations are depicted as rectangles labeled by the location name. The value of each variable is depicted as an arrow from the variable name to the memory location it points-to. The value of a field is depicted by a directed edge labeled with the field name.

The invocation of `reverse` starts in state σ^e_G. The heap of σ^e_G is identical to the one of σ^c_G, but its environment only maps `h`, `reverse`'s formal

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

Figure 4: Axioms for atomic statements in the $\mathcal{GSB}$ semantics. I initializes all pointer fields at l to *null*.

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

$$
\begin{array}{l}
\mathit{where} \\
\quad L_e = L_c,\ \rho_e(v) = \left\{ \begin{array}{ll} \rho_c(x_i) & v = z_i \\ \mathit{null} & \mathit{otherwise} \end{array} \right.,\ h_e = h_c \\
\quad L_r = L_x,\ \rho_r = \rho_c[y \mapsto \rho_x(\mathit{ret})],\ h_r = h_x
\end{array}
$$

Figure 5: Inference rule for function invocation in the $\mathcal{GSB}$ semantics, assuming the formal variables of p are $z_1, \ldots, z_k$ and that p's return value is a pointer.

parameter, to l_0, the value of the actual parameter `x`. The execution of `reverse`'s body ends with `ret` pointing to the head of the reversed list. The memory state at the exit point, lb_x, is denoted by σ_G^x, the state after the invocation of `reverse` is denoted by σ_G^r. Note that the heap in σ_G^r is as in `reverse`'s exit-point, and the environment is as in the call-site, except that the return value (`ret`) is assigned to `y`.

2.4 Observable Properties

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

Definition 2.2 (Field Paths) *A **field path** $\delta \in \Delta = \mathit{FieldId}^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by ϵ.*

Definition 2.3 (Access path) *An **access path** $\alpha = \langle x, \delta\rangle \in V_p \times \Delta$ of a function p is a pair consisting of a local variable of p and a field path. $AccPath_p$ denotes the set of all access paths of function p. $AccPath$ denotes the union of all access paths of all functions in a program.*

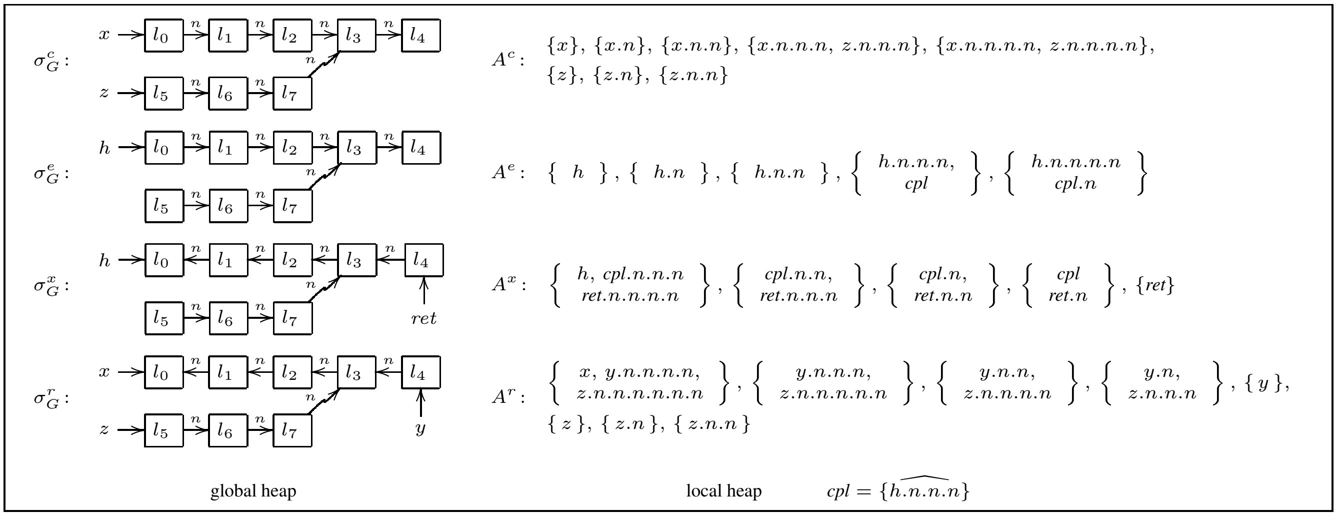


Figure 6: Memory states that arise during the execution of the running example according to the $\mathcal{GSB}$ semantics (left column) and the $\mathcal{LSL}$ semantics (right column). We show the memory states at lb_c, the call-site to `reverse` (first row); lb_e, the entry to `reverse` (second row); lb_x, `reverse`'s exit point (third row); and lb_r, the return-site from `reverse` (fourth row). For the local-heap semantics, the figure shows only the heap (sets of aliased access paths); the memory states at lb_c, lb_e, lb_x, and lb_r are defined as $\sigma_L^c = \langle \emptyset, A^c \rangle$, $\sigma_L^e = \langle \{\{\widehat{h.n.n.n}\}\}, A^e \rangle$, $\sigma_L^x = \langle \{\{\widehat{h.n.n.n}\}\}, A^x \rangle$, and $\sigma_L^r = \langle \emptyset, A^r \rangle$ respectively.

Apart from the above formal definitions, we will sometimes use the notation `x.n.n` for access paths, because its syntax is familiar from a number of programming languages, where it denotes a sequence of field dereferences. Because states and access paths are always associated with a (unique) function p, in the rest of the paper, we omit p whenever it is clear from the context. Also, to simplify notation, we assume that we work with a fixed arbitrary program P.

Definition 2.4 (Access path value) *The value of an access path* $\alpha = \langle x, \delta \rangle$ *in state* $\langle L, \rho, h \rangle$*, denoted by* $[\![\alpha]\!]_G \langle L, \rho, h \rangle$*, is defined to be* $\hat{h}(\rho(x), \delta)$*, where*

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

Note that the value of an access path that traverses a *null*-valued field is defined to be *null*. This definition simplifies the notion of equivalence between the $\mathcal{GSB}$ semantics and $\mathcal{LSL}$, our new semantics. Alternatively, we could have defined the value of such a path to be $\perp$. The semantics given in Fig. 4 checks that a null-dereference is not performed (see the side-conditions listed in the caption).

Definition 2.5 (Access-path equality) *Access paths α and β are **equal** in a given state σ_G, denoted by $[\![\alpha = \beta]\!]_G(\sigma_G)$, if they have the same value in that state, i.e., $[\![\alpha]\!]_G(\sigma_G) = [\![\beta]\!]_G(\sigma_G)$. An access path is **equal to null**, denoted by $[\![\alpha = \mathtt{null}]\!]_G(\sigma_G)$, if $[\![\alpha]\!]_G(\sigma_G) = null$.*

Our semantics is a natural semantics; the stack of activation records is maintained implicitly. However, we need the notion of an access path that starts at a variable of a pending call (i.e., not the current call). In a small-step semantics, this would be an access path that starts at a variable allocated in the activation record of a pending call. We use the term a *pending variable* for a local variable of a pending call, and a *pending access path* for an access path that starts at a pending variable. When we wish to emphasize that a variable (resp. access path) is of the current call, we use the term a *current* variable (resp. a *current* access path). For example, in state σ_G^e, at the entry to `reverse`, `x` is a pending variable, and `z.n.n.n` is a pending access path; the only current variable is `h` and `h.n.n.n` is a current access path.

3 Cutpoints and their Use

In this section, we define cutpoints and describe their use in $\mathcal{LSL}$. To assist the reader, we provide some intuition by referring to the global store-based semantics (see Sec. 2.3) and to a small-step (stack-based) operational semantics. $\mathcal{LSL}$ is a storeless semantics, i.e., memory cells are not identified by locations. Thus, we cannot talk about locations as in Sec. 2.3. Instead, we use the term *objects*.

In $\mathcal{LSL}$, every dynamically allocated object o is represented by the set of pointer-access paths that reach o. Unlike existing storeless semantics [13], in $\mathcal{LSL}$, pending access paths are not represented as parts of the local state of the current procedure. The advantage of our approach is that when a procedure is invoked, it operates only on a part of the heap, namely, the objects that are reachable from the procedure's actual parameters. The downside of this approach is that the memory state just after the call cannot always be defined in terms of the state prior to the call. The intuitive reason for this deficiency is that the description of an object may change due to destructive updates. For example, in the running example, to determine that the pointer-access paths `y.n.n` and `z.n.n.n` are aliased after the invocation of `reverse`, we need to know that the list element pointed-to by `h.n.n.n` when the execution of `reverse` begins, is pointed-to by `ret.n` when the execution ends. To capture this kind of temporal relationship, $\mathcal{LSL}$ tracks the effect of a function on *cutpoints*. Cutpoints are the objects that separate the part of the heap that an invoked function can access from the rest of the heap (excluding the objects pointed-to by actual parameters).

Definition 3.1 (Cutpoints) *A **cutpoint** for an invocation of function p is a heap-allocated object that, in the program state in which the execution of p's body starts, is: (i) reachable from a formal parameter of p (but not pointed-to by one) and (ii) pointed-to by a*

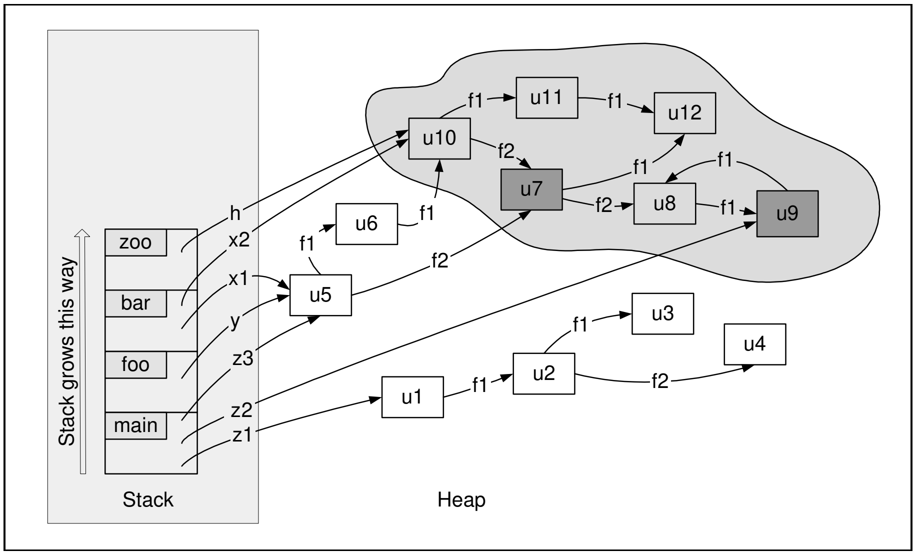


Figure 7: An illustration of the cutpoints for an invocation in a store-based small-step (stack-based) operational semantics. The figure depicts the memory state at the entry to `zoo`. The stack of activation record is depicted on the left side of the diagram. Each activation record is labeled with the name of the function it is associated with. Heap-allocated objects are depicted as rectangles labeled with their location. The value of a pointer 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. The cutpoints for the invocation of `zoo` ($u7$ and $u9$) are heavily shaded. Note that $u10$ is not a cutpoint although it is pointed-to by pending access paths that do not traverse through the shaded part of the heap, e.g., `x2` and `y.f1.f1`. This is because $u10$ is also pointed-to by h, `zoo`'s formal parameter.

pending access path that does not pass through *any object that is reachable from one of* p*'s formal parameters.*

For example, in memory state σ^c_G, the list element at location l_3 is a cutpoint because it is pointed-to by the `n`-field of the list element at location l_7, which is not reachable from the (only) actual parameter `x`. For an additional example, see Fig. 7.

Technically, $\mathcal{LSL}$ uses *cutpoint-labels* to relate the post-state of the function with its pre-state. Cutpoint-labels mark the cutpoints at—and throughout—an invocation.

Definition 3.2 (Cutpoint Labels) *A **cutpoint-label*** $\mathrm{cpl} \in 2^{F_p \times \Delta}$ *for function* p *is a set of access paths that start at a formal parameter of* p*. The set* $2^{F_p \times \Delta}$ *is denoted by* $CPLbs_p$*.*

In every function invocation, $\mathcal{LSL}$ labels all the cutpoints. A cutpoint-label is the set of all access paths that start with a formal parameter (of the invoked function) and point-to the cutpoint when the function execution starts. The label of a cutpoint does not change throughout the execution of the function's body, even if the heap is modified by destructive updates.

For example, the fourth list element in x's list is a cutpoint for the invocation `y=reverse(x)`. The label of this cutpoint is $\{h.n.n.n\}$ because $h.n.n.n$ is the (only) access path that points-to the cutpoint at the entry to the function. A good analogy for the role of cutpoint-labels in our semantics is the use of auxiliary variables in formal verification. Auxiliary variables are used to record variable values at the entry to a function; a cutpoint-label is used to record the access paths that reach a cutpoint at function entry. To emphasize this similarity, we use the notation $\widehat{a}$ where $a \in CPLbs_p$ for cutpoint-labels for function p.

$\mathcal{LSL}$ is able to infer the effect of an invoked function on the heap of its caller by including in the representation of an object all the field paths that reach it and start at a cutpoint.

Definition 3.3 (Cutpoint-anchored paths) *A **cutpoint-anchored path** $\alpha = \langle \text{cpl}, \delta \rangle \in CPLbs_p \times \Delta$ for a function p is a cutpoint-label for function p and a (possibly empty) sequence of fields.*

For example, at the memory state after the execution of `reverse`'s body, the cutpoint-anchored path $\widehat{\{h.n.n.n\}}.n$ is aliased with the access path $ret.n.n$. From this information, our semantics can infer that in the `main` function, at the state after the invocation of `reverse`, $z.n.n.n.n$ is aliased with $y.n.n$.

Technically, during the invocation of a function, an object is represented by the access paths and cutpoint-anchored paths that point-to it.

Definition 3.4 (Generalized access paths) *A **generalized access path** for a function p is either an access path of p or a cutpoint-anchored path of p. $GAccPath_p$ denotes the set of all access paths of function p. $GAccPath$ denotes the union of all access paths of all functions in a program.*

When there is no risk of confusion, we abbreviate a generalized access path of the form $\langle r, \epsilon \rangle$ by r. Note that r can be either a variable, or a cutpoint-label.

Remark 3.5 *Cutpoint-labels isolate the information about the part of the heap that a function cannot access, to the* sharing pattern *of the cutpoints, i.e., to the set of access paths that—at the entry to the function—point-to a cutpoint. Furthermore, the isolation is achieved in a parametric way: although a cutpoint-label expresses the fact that an object is also pointed-to by a pending access path, it is described in terms of the invoked function's formal parameters. This allows us to infer the meaning of a cutpoint-label in a context-independent way.*

Remark 3.6 *Note that because of the "garbage-collecting nature" of storeless semantics, there is a non-trivial technical difficulty in obtaining a local semantics for the storeless model. If a garbage-collection scan was to collect the heap using only the procedure's local variables as the roots, then elements would be garbage collected that are accessible in the global state; adding the cutpoint-labels to the set of "roots" prevent this potential source of unsoundness.*

r	$\in$	$Root_p = V_p \cup CPLbs_p$	
α, β	$\in$	$GAccPath_p = Root_p \times \Delta$	
o	$\in$	$Obj^p_L = 2^{GAccPath_p}$	Objects
A, A_p	$\in$	$Heap^p_L = 2^{Obj^p_L}$	Heaps
$\sigma_L, \langle CPL_p, A_p \rangle$	$\in$	$\Sigma^p_L = 2^{CPLbs_p} \times Heap^p_L$	Memory state

Figure 8: Semantic domains of memory states for function p. We use the syntactic domains V_p, $CPLbs_p$, and $GAccPath_p$ as semantic domains, too (and use italics font to denote a semantics value.)

4 The Localized-Heap Storeless Semantics and its Properties

In this section, we present $\mathcal{LSL}$, the Localized-heap Store-Less semantics and investigate its properties. The semantics is defined in Sec. 4.1. Its properties are described in Sec. 4.2. In Sec. 4.3 we define a language of assertions over access paths and show that $\mathcal{LSL}$ preserves partial and total correctness of assertions expressed in this language.

4.1 The Localized-Heap Storeless Semantics

In this section, we define $\mathcal{LSL}$, the Localized-heap Store-Less semantics. The semantics is a natural semantics and, as before, tracks only pointer values.

To define the semantics, we use the function $\cdot.\cdot$, defined in Fig. 9. It is used as an infix operator. The application $\alpha.\delta$ concatenates the sequence of field identifiers δ to α. We say that a generalized access path α is a *prefix* of a generalized access path β, denoted by $\alpha \leq \beta$, when there is a field path $\delta \in \Delta$, such that $\beta = \alpha.\delta$. We say that α is a *proper prefix* of β, denoted by $\alpha < \beta$, when $\delta \neq \epsilon$. The function $\cdot.\cdot$ is lifted to handle sets of access paths and sets of sequences of field identifiers.

In addition, we make use of the *flat* functional, well-known from functional programming. *flat* M returns the set of all elements of M, if M is a set of sets. Formally, *flat* $M \stackrel{\text{def}}{=} \{x \mid \exists A \in M : x \in A\}$.

4.1.1 Memory States

In this section, we define the representation of memory states in $\mathcal{LSL}$. Traditionally, a storeless semantics represents the heap by an equivalence relation over a set of access paths, where equivalence classes (implicitly) represent allocated objects. For readability, we use the equivalence classes directly.

A *memory state* for a function p is a pair $\langle CPL_p, A_p \rangle$ of a set of cutpoint-labels, (denoted by CPL_p) and a heap (denoted by A_p). A heap is a finite (but unbounded) set of objects. An object (denoted by o) is described by a (possibly infinite) set of *generalized* access paths. Fig. 8 gives the semantic domains used in $\mathcal{LSL}$ for a memory state of a function p.

A memory state $\langle CPL_p, A_p\rangle$ at a given point in an execution is composed of the labels of all the cutpoints of the current invocation (CPL_p) and a representation of the heap (A_p) at that the point in the execution. To exclude states that cannot arise in any program, we now define the notion of *admissible states*.

Definition 4.1 (Admissible memory states) *A memory state $\langle \mathrm{CPL}_p, A_p\rangle$ for a function* p *at a given point in an execution is **admissible** iff (i) A generalized access path points-to (at most) one object, i.e., $\forall o, o' \in A_p$ if $o \neq o'$, then $o \cap o' = \emptyset$; (ii) A is right-regular, i.e., $\forall o_1, o_2 \in A_p$ if $\alpha, \beta \in o_1$ and $\alpha.\delta \in o_2$ then $\beta.\delta \in o_2$; (iii) A_p is prefix-closed, i.e., if $\alpha.f \in \mathit{flat}\, A_p$, then $\alpha \in \mathit{flat}\, A_p$; and (iv) a root of every access path in the description of any object is either a local variable of* p *or a label of one of the cutpoints, i.e., if $\langle r, \delta\rangle \in \mathit{flat}\, A_p$ then either $r \in V_p$ or $r \in CPL_p$; (v) $\emptyset \notin A$; (vi) CPL_p satisfies the following requirements: (a) the cutpoint-labels in CPL_p are mutually disjoint, (b) CPL_p is right-regular (but not necessarily-prefix closed), (c) $\emptyset \notin CPL_p$.*

The first three conditions are standard in storeless semantics. The fourth condition limits the set of cutpoint-anchored paths that are tracked during an invocation to be rooted at a cutpoint of the invocation. The fifth condition is because we only represent objects that are pointed-to by a current or a pending access path. The sixth requirement captures the fact that the set of cutpoints is actually a subset of the objects in the heap when the function is invoked. Thus, CPL_p satisfies the first two requirements of heaps. However, because it is only a subset, it is not necessarily prefix-closed. The fact that the empty set is never in CPL_p is immediate once we recall that cutpoint-labels are generated only for objects that can be reached from the actual parameters when the function is invoked.

Because $\mathcal{LSL}$ preserves admissibility of states (see Lem. 4.9), in the sequel, whenever we refer to an $\mathcal{LSL}$ state, we mean an *admissible* $\mathcal{LSL}$ state.

It is possible to extract aliasing relationships from the sets of generalized access paths that describe the objects in a heap, and in particular to observe the heap structure as follows: a current variable `x` *points-to* an object o iff the access path $\langle x, \epsilon\rangle$ is in o. Similarly, cutpoint-label *cpl labels* object o iff $\langle \mathit{cpl}, \epsilon\rangle$ is in o. The field `f` of an object o_1 *points-to* object o_2 iff for every generalized access path $\langle r, \delta\rangle$ in o_1, the generalized access path $\langle r, \delta f\rangle$ is in o_2. A generalized access path α *points-to* (resp. *passes through*) an object o, if $\alpha \in o$ (resp. $\exists \beta < \alpha$ such that $\beta \in o$). An object o is *reachable* from a variable x, if there exists a field path $\delta \in \Delta$ such that $\langle x, \delta\rangle \in o$.

Example 4.2 The heap of the running example at the state in which `reverse` is invoked is shown in the first row in the second column of Fig. 6 (labeled A^c). It shows eight sets of generalized access paths. Each set represents one allocated list-element. At A^c, *x.n.n.n* and *z.n.n.n* point-to the same object. The set of cutpoint-labels at the call site is empty. This is always the case for the main function. The fourth element in `x`'s list is a cutpoint for the invocation of `reverse`: it is reachable from an actual parameter (its representation includes $x.n.n.n$) and by a field of an object that is not passed to the invoked function (the `n`-field of the third object in `z`'s list). The heap at the beginning of `reverse` (shown in Fig. 6, labeled by A^e) differs from A^c in three ways: (i) there are only five objects in the heap; (ii) the set of cutpoint-labels contains $\{\widehat{h.n.n.n}\}$, which labels the fourth element in the list; and (iii) objects are represented in terms of the generalized access paths that start either with h or with $\{\widehat{h.n.n.n}\}$.

4.1.2 Inference Rules

The meaning of statements is described by a transition relation $\overset{L}{\rightsquigarrow} \subseteq (\sigma_L \times \textit{stms}) \times \sigma_L$. We give axioms for assignments and an inference rule for procedure calls in Fig. 10 and Fig. 11, respectively. All other statements are handled in the standard way [25]. To simplify notation, we assume A with a certain index (resp. prime) to be the heap component of a state σ_L with the same index (resp. prime). We use the same convention for indexed (or primed) versions of *CPL* and a state's cutpoint-labels component.

4.1.2.1 Helper Functions To define the inference rules, we use the following functions: $[\cdot]_\cdot$, *rem*$(\cdot,\cdot)$ and *add*$(\cdot,\cdot)$, which are defined in Fig. 9. We use a as a metavariable ranging over sets of generalized access paths, which are not necessarily objects, whereas o always stands for objects.

The function $[\alpha]_A$ returns the object that α points-to in heap A. When α does not point-to any object, $[\alpha]_A$ returns the empty set (which by definition never describes an object pointed-to by a current, or even a pending, access path).

The function *rem* takes as its arguments a heap A and a set of generalized access paths a. It removes from the description of every object in heap A all the access paths that have a prefix in a. Whenever *rem* removes all the (generalized) access paths from the description of an object, that object is removed from the description of the heap. The function $add(A, a, \alpha)$ yields a modified version of heap A, where to every object $o \in A$ reachable from α by following some field path $\delta \in \Delta$, the generalized access paths $a.\delta$ are added.

In addition, we make use of $map()$, another well known functional from functional programming. The functional $map(f)\ M$ applies f to every element of M and returns the resulting set. Formally, $map(f)\ M \stackrel{\text{def}}{=} \{f(x) \mid x \in M\}$.

4.1.2.2 Atomic Statements The *axioms* for atomic statements are given in Fig. 10. We simplify the semantics by making the same assumptions as in Sec. 2.3.

Assigning `null` to a variable `x` does not modify the link structure of the heap. We only need to eliminate all the access paths that start with `x`, using the *rem* function.

The semantics for the assignment `x = y` copies the value of the variable `y` into `x` by adding an access path $\langle x, \delta\rangle$ to any object o that can be reached from `y` by following a field path δ, i.e., $\langle y, \delta\rangle$ points-to o. This is accomplished by applying *add* to the given heap, the singleton set $\{x\}$, and the access path y.

The rule for field dereference `x = y.f` is similar. It adds the access path $\langle x, \delta\rangle$ to any object that can be reached from `y` by following field `f`, and then continuing with field path δ. Note, however, that the rule can be applied only if `y` points-to an object, i.e., the semantics checks that a null-dereference is not performed.

$$.\colon GAccPath \times \Delta \to GAccPath \text{ s.t.}$$
$$\langle r, \delta\rangle.\delta' \stackrel{\text{def}}{=} \langle r, \delta\delta'\rangle$$

$$.\colon 2^{GAccPath} \times \Delta \to 2^{GAccPath} \text{ s.t.}$$
$$a.\delta \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a\}$$

$$.\colon 2^{GAccPath} \times 2^{\Delta} \to 2^{GAccPath} \text{ s.t.}$$
$$a.D \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a, \delta \in D\}$$

$$[]\colon GAccPath \times Heap_L \to Obj_L \text{ s.t.}$$
$$[\alpha]_A \stackrel{\text{def}}{=} \{\beta \in a \mid a \in A, \alpha \in a\}$$

$$rem\colon Heap_L \times 2^{GAccPath} \to Heap_L \text{ s.t.}$$
$$rem(A, a) \stackrel{\text{def}}{=} (map(\lambda o.o \setminus a.\Delta)\ A) \setminus \{\emptyset\}$$

$$add\colon Heap_L \times 2^{GAccPath} \times GAccPath \to Heap_L \text{ s.t.}$$
$$add(A, a, \alpha) \stackrel{\text{def}}{=} map(\lambda o.\, o \cup a.\{\delta \in \Delta \mid \alpha.\delta \in o\})\ A$$

Figure 9: Helper functions.

$\langle \mathtt{x} = \mathtt{null}, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, rem(A, \{x\})\rangle$	
$\langle \mathtt{x} = \mathtt{y}, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, add(A, \{x\}, y)\rangle$	
$\langle \mathtt{x} = \mathtt{y.f}, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, add(A, \{x\}, y.f)\rangle$	$y \in flat\ A$
$\langle \mathtt{x.f} = \mathtt{null}, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, rem(A, [x]_A.f)\rangle$	$x \in flat\ A$
$\langle \mathtt{x.f} = \mathtt{y}, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, \overline{add(A, [x]_A.f, y)}\rangle$	$x \in flat\ A$
$\langle \mathtt{x} = \mathtt{alloc}\ t, \langle CPL, A\rangle\rangle \stackrel{L}{\leadsto} \langle CPL, A \cup \{\{x\}\}\rangle$	

Figure 10: Axioms for atomic statements in the local heap semantics. Note that the set of cutpoint-labels is not changed. The side-condition $x \in flat\ A$ (resp. $y \in flat\ A$) means that `x`'s (resp. `y`) value is not *null*.

A destructive update `x.f = null` (potentially) modifies the link structure of the heap. Thus, every *generalized* access path that has a prefix aliased with $\langle x, f\rangle$ is removed from the description of every object in the heap. Note, that $[x]_A$ returns all the access paths that are aliased with x. Concatenating $[x]_A$ with f returns the set of prefixes of affected access paths. Again, the rule can be applied only if `x` points-to an object.

An assignment `x.f = y` also has a (potential) effect on all the access paths that are aliased with x. After this assignment, any object o that can be reached by following the field path δ from y, i.e., $\langle y, \delta\rangle \in o$, is also reachable by traversing some (generalized) access path aliased with x, followed by an `f`-field, and continuing with δ. As this is a place where cycles can be created, *add* does not necessarily return a right-regular heap. Therefore we apply the operator $\bar{\cdot}$. $\bar{A}$ is defined to be the set of equivalence classes obtained from the least right-regular, prefix-closed, equivalence relation that is a superset of the equivalence relation induced by A.[3] Note that this definition may only add access paths to the description of existing objects.

The (deterministic) semantics of memory allocation `x = alloc t` adds a new object that is described by $\{x\}$ to the heap. Note that this definition (implicitly) initializes the fields of the new object to *null*.

4.1.2.3 Function Calls The *inference rule* for function calls is defined in Fig. 11. The rule defines the program state σ_L^r that results from an invocation `y=p(`$\mathtt{x}_1, \ldots, \mathtt{x}_k$`)` at memory state σ_L^c, assuming that the execution of the body of p at memory state σ_L^e results in memory state σ_L^x. The heaps A^c and A^r are described by sets of generalized access paths starting at the caller's variables and cutpoint-labels, whereas the heaps A^e and A^x are described by sets of generalized access paths that start at the callee's formal parameters, cutpoint-labels, and return variable. The rule provides the means to reconcile the different representations.

The rule uses the functions $Call_q^{y=p(x_1,\ldots,x_k)}$ and $Ret_q^{y=p(x_1,\ldots,x_k)}$, which are parameterized for each call statement in the program. $Call_q^{y=p(x_1,\ldots,x_k)}$ computes the memory state σ_L^e that results at the entry of p when $y = p(x_1, \ldots, x_k)$ is invoked by q in memory state σ_L^c. The caller's memory state after the invocation is restored by the function $Ret_q^{y=p(x_1,\ldots,x_k)}$. This function computes the memory state of the caller at the return-site (σ_L^r) according to q's memory state at the call-site (σ_L^c) and p's memory state at the exit-site (σ_L^x). In the rest of this section we describe the rule for an arbitrary call statement $y = p(x_1, \ldots, x_k)$ by an arbitrary function q. The rule utilizes additional helper functions, defined in Fig. 12, which we gradually explain.

The main idea behind the rule is to utilize the fact that a function cannot modify objects that are not in its local-heap (i.e., in the part of the heap that is *not* reachable from any actual parameter when the function is invoked). In particular, because $\mathcal{LSL}$ describes objects in terms of the (generalized) access paths that point-to them, these "inaccessible" objects have the same description before and after the call. Thus, only the description of the objects in the function's local-heap (i.e., in the part of the heap that the function can access) is (possibly) updated. The update is carried out using the *cutpoints of the invocation*.[4] In essence, the semantics freezes the initial descriptions of the cutpoints and arranges for them to persist throughout the execution of the called function. This sets up a relation between values on entry to values on exit. At the return, the frozen information is used to update the description of objects in the called function's local-heap via an operation that is (roughly) similar to a relational join [9].

[3] The operator $\bar{\cdot}$ is similar to the ρ_{rstc} operator in [14].

[4] The same mechanism is used to compute the description of objects that the callee allocates.

$$
\begin{array}{l}
Call_q^{y=p(x_1,\dots,x_k)} : \Sigma_L^q \to \Sigma_L^p \ s.t. \\
\quad Call_q^{y=p(x_1,\dots,x_k)}(\langle \mathit{CPL}^c, A^c\rangle) \stackrel{\text{def}}{=} \\
\quad Let \\
\circledast \left\{
\begin{array}{l}
O_c^{args} = \{[x_i]_{A^c} \mid 1 \le i \le k,\ [x_i]_{A^c} \neq \emptyset\} \\
O_c^{passed} = \mathit{RObjs}(A^c)\ O_c^{args} \\
O_c^{cp} = \mathit{CPObjs}_q(\langle \mathit{CPL}^c, A^c\rangle)\ (O_c^{args}, O_c^{passed}) \\
bind_{args} = \lambda o \in O_c^{args}.\{\langle h_i, \epsilon\rangle \mid 1 \le i \le k,\ x_i \in o\} \\
bind_{cp} = \lambda o \in O_c^{cp}.\{\langle sub(bind_{args})\ o, \epsilon\rangle\} \\
bind_{call} = \lambda o \in O_c^{args} \cup O_c^{cp}. \left\{ \begin{array}{ll} bind_{args}(o) & o \in O_c^{args} \\ bind_{cp}(o) & o \in O_c^{cp} \end{array} \right.
\end{array}
\right. \\
\quad in \\
\quad\quad \langle map(sub(bind_{args}))\ O_c^{cp},\ map(sub(bind_{call}))\ O_c^{passed}\rangle
\end{array}
$$

$$
\begin{array}{l}
Ret_q^{y=p(x_1,\dots,x_k)} : \Sigma_L^q \times \Sigma_L^p \to \Sigma_L^q \ s.t. \\
\quad Ret_q^{y=p(x_1,\dots,x_k)}(\langle \mathit{CPL}^c, A^c\rangle, \langle \mathit{CPL}^x, A^x\rangle) \stackrel{\text{def}}{=} \\
\quad Let \\
\quad\quad \circledast \\
\quad\quad bind_{ret} = \lambda a \in range(bind_{call}) \cup \{\{\langle ret, \epsilon\rangle\}\}. \\
\quad\quad\quad \left\{ \begin{array}{ll} \{\langle y, \epsilon\rangle\} & a = \{\langle ret, \epsilon\rangle\} \\ \mathit{Bypass}(O_c^{passed}) \circ bind_{call}^{-1}(a) & \text{otherwise} \end{array} \right. \\
\quad in \\
\quad\quad \langle \mathit{CPL}^c, (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x\rangle
\end{array}
$$

$$
\frac{\langle \text{body of } p, \sigma_L^e\rangle \stackrel{L}{\rightsquigarrow} \sigma_L^x}{\langle y = p(x_1,\dots,x_k), \sigma_L^c\rangle \stackrel{L}{\rightsquigarrow} \sigma_L^r}
$$

$$
\begin{array}{l}
where \\
\quad \sigma_L^e = \mathit{Call}_q^{y=p(x_1,\dots,x_k)}(\sigma_L^c) \\
\quad \sigma_L^r = \mathit{Ret}_q^{y=p(x_1,\dots,x_k)}(\sigma_L^c, \sigma_L^x)
\end{array}
$$

Figure 11: The inference rule for function calls in $\mathcal{LSL}$. The rule is given for an arbitrary call statement $y = p(x_1, \dots, x_k)$ by an arbitrary function q. We assume that the formal parameters of p are $h_1, \dots, h_k$.

(The operation is not a "pure" relational join because of some name adjustments that are needed due to the different representation of objects by the caller and by the callee.)

To find which objects are in the local-heap of the called function, i.e., reachable from the actual parameters ($x_1, \dots, x_k$), we first compute the set of objects that are *pointed-to* by p's actual parameters (O_c^{args}). Then, the auxiliary function *RObjs* finds the part of the caller's heap (A^c) that is reachable from these objects (O_c^{passed}).

$$
\begin{array}{l}
\mathit{RObjs}\colon \mathit{Heap}_L \to (2^{\mathit{Obj}_L} \to 2^{\mathit{Obj}_L})\ \mathit{s.t.}\\
\quad \mathit{RObjs}(A)\ O \stackrel{\text{def}}{=} \{o \in A \mid o' \in O,\ \delta \in \Delta,\ o'.\delta \subseteq o\}\\[1ex]
\mathit{Bypass}\colon 2^{\mathit{Obj}_L} \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}})\ \mathit{s.t.}\\
\quad \mathit{Bypass}(O)\ o \stackrel{\text{def}}{=} \{\langle r, \delta\rangle \in o \mid \forall \delta' < \delta.\ \langle r, \delta'\rangle \notin \mathit{flat}\, O\}\\[1ex]
\mathit{sub}\colon (2^{\mathit{GAccPath}} \to 2^{\mathit{GAccPath}}) \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}})\ \mathit{s.t.}\\
\quad \mathit{sub}(\mathit{bind})\ o \stackrel{\text{def}}{=} \mathit{flat}\left\{ \mathit{bind}(a).\delta \;\middle|\; \begin{array}{c} a \in \mathit{dom}(\mathit{bind}),\\ \delta \in \Delta,\ a.\delta \subseteq o \end{array} \right\}\\[1ex]
\mathit{CPObjs}_q\colon \Sigma_L^q \to (2^{\mathit{Obj}_L^q} \times 2^{\mathit{Obj}_L^q} \to 2^{\mathit{Obj}_L^q})\ \mathit{s.t.}\\
\quad \mathit{CPObjs}_q(\langle \mathit{CPL}^c, A^c\rangle)\ (O_c^{args}, O_c^{passed}) \stackrel{\text{def}}{=}\\
\quad\ \mathit{Let}\\
\quad\quad \begin{array}{lcl}
O_{deep} & = & O_c^{passed} \setminus O_c^{args}\\
O_{vars} & = & \{[\langle x, \epsilon\rangle]_{A^c} \in O_{deep} \mid x \in V_q\}\\
O_{\mathit{fld}} & = & \left\{ o \in O_{deep} \;\middle|\; \begin{array}{c} \exists o' \in A^c \setminus O_c^{passed},\\ \exists f \in \mathit{FieldId},\ o'.f \subseteq o \end{array} \right\}\\
O_{cpl} & = & \{[\langle \mathit{cpl}, \epsilon\rangle]_{A^c} \in O_{deep} \mid \mathit{cpl} \in \mathit{CPL}^c\}
\end{array}\\
\quad\ \mathit{in}\\
\quad\quad O_{vars} \cup O_{cpl} \cup O_{\mathit{fld}}
\end{array}
$$

Figure 12: Helper functions for the function-call rule. The function CPObjs_q is parameterized for every function q in the program. Recall that V_q is the set of q's local variables.

The description of the objects after the call should account for the mutations (destructive updates) of the heap performed by the callee. However, because the invoked function cannot modify objects that it cannot access, it can only modify fields of objects in O_c^{passed}. Thus, to compute the (possibly) updated description of objects in O_c^{passed} (as well as of objects that the callee allocates) it is sufficient to have a description of every object in O_c^{passed} (and of every object allocated by the callee) comprised of the (generalized) access paths that start at objects that separate O_c^{passed} from the rest of the caller's heap: When the function returns, we just replace any (generalized) access paths $\langle r_p, \delta_p\rangle$ in the description of every object in the heap of the callee (A^x) that start at a "separating object" o', by access paths of the caller $\langle r_q, \delta_q\delta_p\rangle$ such that $\langle r_q, \delta_q\rangle$ points-to o', but does not pass through O_c^{passed} (and thus cannot be modified). Technically, this is done as described below.

The auxiliary function CPObjs_q (cf. Fig. 12) determines the cutpoints for this function invocation (O_c^{cp}). Cutpoints are the objects that "separate" O_c^{passed} from the rest of the caller's heap. For expository reasons, we do not want to consider objects that are pointed-to by actual parameters as cutpoints. Thus, the function CPObjs_q, which is passed the caller's memory state as well as the previously computed O_c^{args} and O_c^{passed}, considers only objects in $O_{deep} = O_c^{passed} \setminus O_c^{args}$ as possible cutpoints.

Following the intuition of cutpoints as "separating objects", an object $o \in O_{deep}$ is qualified as a cutpoint if (and only if) one of the following holds:

- o is pointed-to by a local variable of the caller (O_{vars}), or
- o is pointed-to by an object in the part of the caller's heap that is not passed to the function (O_{fld}), or
- o separates the heap of the *caller* from the heap of one of the pending calls, i.e., o is a cutpoint of the invocation of the caller (O_{cpl}).

Back in Fig. 11 we define several binding mappings to bridge the gap between the two different representations of objects (in terms of access paths of the caller and in terms of access paths of the callee). The function $bind_{args}$ maps objects pointed-to by actual parameters to the set of "trivial" access paths that are made up of the corresponding formal parameters. The function $bind_{cp}$ maps every cutpoint (in the caller representation) to the set of access paths that start with a formal parameter of the caller and point-to that cutpoint at the entry to the function, i.e., $bind_{cp}$ maps a cutpoint to its label (see Sec. 3). To compute the label of a cutpoint o, we apply $sub(bind_{args})$. The latter denotes a function that replaces every access path that starts with an actual parameter $\langle x_i, \delta \rangle$ in the representation of o by an access path $\langle h_i, \delta \rangle$ that starts with the corresponding formal parameter. (sub is defined in Fig. 12.) The $bind_{call}$ combines the previous two mappings trivially as they have disjoint domains.

Having defined these mapping functions, computing the memory state of p in which its body will be evaluated (i.e., the description of the heap at the function entry) is straightforward. The set of cutpoint-labels (CPL^e) is computed by applying $bind_{cp}$ to every cutpoint. The heap component (A^e) is constructed by applying $bind_{call}$ to every object in O_c^{passed}. Note that in the resulting description, objects are described by the set of (generalized) access paths that point-to them and start either at a formal parameter or at a cutpoint object.

To handle the return of function p, we use an additional binding, $bind_{ret}$. This mapping is the inverse of $bind_{call}$ (hence getting back to the caller's representation of the object) composed with the function $Bypass(O_c^{passed})$, which filters out generalized access paths (of the caller) that *pass through* the part of the heap that p had access to (O_c^{passed}). In addition, it also takes care of replacing access paths starting with special variable ret with the same access paths starting with result variable y. Note that applying $bind_{ret}$ is well defined because *CPL*^{x} and *CPL*^{e} are equal (the callee cannot modify the set of objects that separate its own local-heap from the local-heap of of some pending call[5]).

The cutpoint-labels component of the state after the return of p is the same as before the invocation (CPL^c) because the callee (p) cannot modify the set of objects that separate the heap of its caller (q) from the heap of some other (earlier) pending-call. The new heap is called A^r. It is derived by removing from the heap at the call-site the passed objects (O_c^{passed}), plugging in the heap that results from evaluating p's body (A^x), and substituting the description of all the objects by applying $sub(bind_{ret})$ to every object in A^x.

[5]Note that in any transition $\langle \sigma_L, st \rangle \overset{L}{\rightsquigarrow} \sigma'_L$, the cutpoint-labels component in σ_L and σ'_L is the same.

Example 4.3 Applying the function-call rule for the invocation of `reverse` in our running example results in the following sets and mappings:

$$
\begin{aligned}
&O_c^{args} = \{\{x\}\} \\
&O_c^{passed} = \{\{x\}, \{x.n\}, \{x.n.n\}, \{x.n.n.n, z.n.n.n\}, \} \\
&O_c^{cp} = \{z.n.n.n, x.n.n.n\} \\
&bind_{args} = \{x\} \mapsto \{h\} \\
&bind_{cp} = \{z.n.n.n, x.n.n.n\} \mapsto \{\{\widehat{h.n.n.n}\}\} \\
&bind_{ret} = \{\{x\} \mapsto \{h\}, \{\{\widehat{h.n.n.n}\}\} \mapsto \{z.n.n.n\}, \{ret\} \mapsto \{y\}\}
\end{aligned}
$$

In particular, the fourth element in `x`'s list is a cutpoint for the invocation of `reverse` (see Sec. 4.1.1) and its label is $\{\widehat{h.n.n.n}\}$. Thus, when the execution of `reverse`'s body starts, the cutpoint is represented by the following set of (generalized) access paths: $\{h.n.n.n, \{\widehat{h.n.n.n}\}\}$. When the execution of the function body ends, the cutpoint-anchored paths in the representation of every object in A^x (see Fig. 6) are replaced by access paths that start with $z.n.n.n$, the only access path that points-to the cutpoint at the call-site and *bypasses* the objects that were passed to `reverse`. For example, the cutpoint-anchored path $\{\widehat{h.n.n.n}\}.n$ in the representation of the third element in the returned list is replaced by $z.n.n.n.n$.

4.2 Properties of the Semantics

The only means by which a program can observe a state is by access paths. In particular, the program cannot refer to the cutpoint-labels component of the state. To state the theorems, we need some preliminary definitions about access-path equality and observational equivalence. We use the same simplifying notational conventions as in Sec. 4.1.2. Note that in both semantics an access path is equal to `null` when it has a prefix which is equal to `null`.

Definition 4.4 (Access path equality) *Access paths α and β are **equal** in a given state σ_L, denoted by $[\![\alpha = \beta]\!]_L(\sigma_L)$, if $\forall a \in A.\, \alpha \in a \iff \beta \in a$. An access path α is **equal to null** in state σ_L, denoted by $[\![\alpha = \texttt{null}]\!]_L(\sigma_L)$, if $\alpha \notin$ flat A.*

Definition 4.5 (Observational equivalence) *Let p be a function. The states $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ are **observationally equivalent** if for all $\alpha, \beta, \gamma \in AccPath_p$,*

(i) $[\![\alpha = \beta]\!]_L(\sigma_L) \Leftrightarrow [\![\alpha = \beta]\!]_G(\sigma_G)$, *and*

(ii) $[\![\gamma = \texttt{null}]\!]_L(\sigma_L) \Leftrightarrow [\![\gamma = \texttt{null}]\!]_G(\sigma_G)$.

We also define observational equivalence between states in $\mathcal{LSL}$ in the same way.

4.2.1 Semantic Equivalence

The following theorem is the main theorem in the paper. It states that $\mathcal{LSL}$ is equivalent to $\mathcal{GSB}$, in the sense that both behave equivalently w.r.t. termination, and that execution of statements preserves observational equivalence.

Theorem 4.6 (Equivalence) *Let p be a function. Let $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ be observationally equivalent states. Let st be an arbitrary statement in p. The following holds:*

$$\langle st, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma'_L \iff \langle st, \sigma_G \rangle \overset{G}{\rightsquigarrow} \sigma'_G.$$

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

We prove The. 4.6 by establishing a stronger property of the $\mathcal{LSL}$ semantics: the preservation of *Context-Aware Equivalence*. Informally, the *Context-Aware Equivalence* theorem shows that the cutpoints are, in a sense, the "store-based part" of $\mathcal{LSL}$: they are used to label and fix certain objects, something that is done automatically if we have locations. The theorem is formally stated and proved in App. C.3.

The following theorem states that $\mathcal{LSL}$ can be used to: (i) verify data-structure invariants that are expressed by access-path equalities at a program point; and (ii) assert the absence of *null*-valued pointer dereferences. Formally, a property is an invariant at a (labeled) statement if is satisfied in any memory-state that occurs just before the (labeled) statement is executed.

Corollary 4.7 *Let P be a program, p a function , lb a program point in p. For any $\alpha, \beta \in AccPath_p$, $[\![\alpha = \beta]\!]_L$ is an invariant of P at lb iff $[\![\alpha = \beta]\!]_G$ is an invariant of P at lb.*

The following theorem states that $\mathcal{LSL}$ can detect memory leaks[6] without investigating reachability from *roots* of pending access paths. A memory leak can occur only when a variable or a field is assigned `null`. The "leaked objects" are the ones that are not pointed-to only by suffixes of the nullified variable (or field).

Corollary 4.8 *A memory leak can occur only when a variable or a field is assigned* `null`*. Furthermore,*

- *Executing a statement* `x = null` *in a memory state $\langle \mathrm{CPL}, A \rangle$ leaks an object o iff $o \subseteq x.\Delta$.*
- *Executing a statement* `x.f = null` *in a memory state $\langle \mathrm{CPL}, A \rangle$ leaks an object o iff $o \subseteq [\langle x, \epsilon \rangle]_A.f.\Delta$.*

[6]By a memory leak we mean an object that is not pointed-to by any access path; i.e., neither by an access path of the current call nor by one of a pending call.

4.2.2 Standard Properties

The following theorems state that the $\mathcal{LSL}$ semantics has certain standard properties. The following lemma ensures that the $\mathcal{LSL}$ semantics preserves admissible states.

Lemma 4.9 (Admissibility) *Let* st *be a statement and* σ_L *an admissible state. If* $\langle \text{st}, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma'_L$ *then* σ'_L *is also an admissible state.*

Furthermore, $\mathcal{LSL}$ is a deterministic semantics; this holds because memory allocation is deterministic. (In contrast, most store-based semantics do not have a deterministic memory-allocation mechanism.)

Lemma 4.10 (Determinism) *Let* st *be a statement and* σ_L *an admissible state. If* $\langle \text{st}, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma'_L$ *and* $\langle \text{st}, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma''_L$*, then* $\sigma'_L = \sigma''_L$*.*

The following lemma states that $\mathcal{LSL}$ is fully abstract. To state this property, we use the notation $P[\cdot]$ for *program contexts*. The *denotation* $[\![st]\!]_L$ of a statement is defined to be the (partial) function $\lambda\sigma_L.\sigma'_L$ where $\langle \sigma_L, st \rangle \overset{L}{\rightsquigarrow} \sigma'_L$.

Lemma 4.11 (Full Abstraction) *Let* st_1 *and* st_2 *be two statements such that for all program contexts* $P[\cdot]$ *and all states* σ_L *the states* $[\![P[st_1]]\!]_L(\sigma_L)$ *and* $[\![P[st_2]]\!]_L(\sigma_L)$ *are observationally equivalent. Then* $[\![st_1]\!]_L = [\![st_2]\!]_L$*.*

4.2.3 Modularity

The following theorems state that $\mathcal{LSL}$ manipulates the heap in a "modular" way. Thanks to these properties, the *analysis* (see Sec. 5) can also be a modular.

The following theorem states that a function has no effect on the observable properties of the unreachable part of the heap.

Theorem 4.12 (Framed Execution) *Let* q *be a function. Let* $\sigma^c_L, \sigma^r_L \in \Sigma^q_L$ *be states of function* q *such that* $\langle \sigma^c_L, y = p(x_1, \ldots, x_k) \rangle \overset{L}{\rightsquigarrow} \sigma^r_L$*. Let* O^{passed}_c *be the objects in* σ^c_L *that are reachable from* $x_1, \ldots, x_k$*. Let* $\alpha, \beta, \gamma \in GAccPath_q \setminus y.\Delta$ *be arbitrary generalized access paths of function* q *that do not start with* y *and do not* ***pass through*** *objects in* O^{passed}_c*. The following properties hold:*

(i) $[\![\alpha = \beta]\!]_L(\sigma^c_L) \iff [\![\alpha = \beta]\!]_L(\sigma^r_L)$*, and*

(ii) $[\![\gamma = \texttt{null}]\!]_L(\sigma^c_L) \iff [\![\gamma = \texttt{null}]\!]_L(\sigma^r_L)$*.*

Note that the above theorem is also applicable for access paths that *point-to* objects in the part of the heap that the function can access, but do not pass through this part.[7]

The following theorem states that a function cannot observe its context, i.e., that the execution of the function body is not affected by the cutpoint-labels component of the state.

[7] Recall that an access path α *passes through* an object o if there exists a proper prefix $\alpha' < \alpha$ such that α' points-to o.

Theorem 4.13 (Context Indifference) *Let p be a function. Let $\sigma_L^1, \sigma_L^2 \in \Sigma_L^p$ be observationally equivalent states of p. Let st be an arbitrary statement in p. The following holds:*

$$\langle \sigma_L^1, st \rangle \overset{L}{\rightsquigarrow} \sigma_L^{1'} \iff \langle \sigma_L^2, st \rangle \overset{L}{\rightsquigarrow} \sigma_L^{2'}.$$

Furthermore, $\sigma_L^{1'}$ and $\sigma_L^{2'}$ are observationally equivalent.

The following theorem states that a function has a similar effect on contexts that differ only by the *contents* of the part of the heap that is not reachable from actual parameters. Practically speaking, this theorem justifies the reuse of the results of an analysis of a function invocation (see Sec. 5) in "similar" contexts.

Theorem 4.14 (Heap Modularity) *Let p be a function. For $i = 1, 2$, let q_i be a function, $\sigma_L^{c_i} \in \Sigma_L^{q_i}$ be a state of function q_i, $y^i = p(x_1^i, \ldots, x_k^i)$ be a statement in function q_i, and $\sigma_L^{e_i} = Call_{q_i}^{y^i=p(x_1^i,\ldots,x_k^i)}(\sigma_L^{c_i}) \in \Sigma_L^p$ be the state that results at the entry to function p when it is invoked at $\sigma_L^{c_i}$. If $\sigma_L^{e_1}$ and $\sigma_L^{e_2}$ are observationally equivalent then the following properties hold:*

(i) $\langle \sigma_L^{c_1}, y^1 = p(x_1^1, \ldots, x_k^1) \rangle \overset{L}{\rightsquigarrow} \sigma_L^{r_1} \iff \langle \sigma_L^{c_2}, y^2 = p(x_1^2, \ldots, x_k^2) \rangle \overset{L}{\rightsquigarrow} \sigma_L^{r_2}$, *and*

(ii) *if $CPL^{e_2} \subseteq CPL^{e_1}$ and $\langle \sigma_L^{e_1}, body\ of\ p \rangle \overset{L}{\rightsquigarrow} \sigma_L^{x_1}$, then*
$\sigma_{\mathbf{L}}^{\mathbf{r_2}} = \mathbf{gc}(Ret_{q_2}^{y^2=p(x_1^2,\ldots,x_k^2)}(\sigma_L^{c_2}, \sigma_{\mathbf{L}}^{\mathbf{x_1}}))$, *where* $\mathbf{gc}(\langle \text{CPL}, A \rangle) \overset{\text{def}}{=} \langle \text{CPL}, A \setminus \emptyset \rangle$.

We need to apply $\mathbf{gc}$ to the heap produced by $Ret_{q_2}^{y^2=p(x_1^2,\ldots,x_k^2)}(\sigma_L^{c_2}, \sigma_{\mathbf{L}}^{\mathbf{x_1}})$ because of the following technical reason: it is possible that some of the objects in A^{x_1} are reachable only from objects that are cutpoints when p is invoked at $\sigma_L^{c_1}$ but not when it is invoked at $\sigma_L^{c_2}$. Thus, some objects that are reachable (i.e., pointed-to by a current or a pending access path) at $\sigma_L^{r_1}$ might not be reachable at $\sigma_L^{r_2}$.

4.3 Assertion Language

In this section, we define $\mathbf{Assn_{AP}}$, a language of assertions over access paths. $\mathcal{LSL}$ preserves validity of invariants and of partial-correctness and total-correctness assertions expressed in $\mathbf{Assn_{AP}}$. Our aim, in this section, is to provide a syntactic characterization of $\mathbf{Assn_{AP}}$. In particular, we do not develop a proof system.

The definition of $\mathbf{Assn_{AP}}$ is similar to [42, Ch. 6]. First, we extend the syntactic category of $AccPath_q$ to include *access-path variables* over which we can quantify. As indicated by its name, an access-path variable ranges over access paths. The extended syntactic category $EAccPath_q$ of access paths for function q is defined in Fig. 13 (1). An extended access path e is either an access path of function q (α), an access-path variable (χ), or a concatenation of an extended access path with some field path ($e.\delta$).

The syntax of assertions in $\mathbf{Assn_{AP}}$ is defined in Fig. 13 (2). $\mathbf{Assn_{AP}}$ is a first-order language with 3 relation symbols (i.e., *atomic assertions*): al, $isNull$, and $\leq$.

The intended meaning of the *atomic assertions* is:

$al(e_1, e_2)$: the extended access paths e_1 and e_2 denote the same object (or `null`).

$isNull(e)$: the extended access path e denotes *null*.

$e_1 \leq e_2$: the extended access path e_1 is a prefix of the extended access path e_2.

The meaning of an assertion is defined using an *interpretation* for access-path variables. An interpretation $I\colon AccPathVar \to AccPath_q$ associates an access-path variable with a particular access path. We lift an interpretation I to $\hat{I}\colon EAccPath_q \to AccPath_q$ as follows:

$$\hat{I}(e) = \begin{cases} I(e) & e \in AccPathVar \\ e & e \in AccPath_q \\ \hat{I}(e_1).f & e = e_1.f. \end{cases}$$

We use $\hat{I}$ to specify when a state $\sigma_L \in \Sigma_L^q$ (resp. $\sigma_G \in \Sigma_G^q$) satisfies an assertion $ass \in AtmAssrt_q$ w.r.t. I, denoted by $\sigma_L \models_L^I ass$ (resp. $\sigma_G \models_G^I ass$). The satisfaction relation between states in $\mathcal{LSL}$ (resp. $\mathcal{GSB}$), interpretations and atomic assertions is specified in Fig. 13(3) (resp. Fig. 13(4)). Note that by interpreting access-path variables as access paths (and not, for example, as locations) we can use the *same* interpretation in the definition of the satisfaction relation for both $\mathcal{LSL}$ and $\mathcal{GSB}$. The satisfaction relation for non-atomic assertions is defined by structural induction in a standard manner (e.g., see [42, Ch. 6]).

Example 4.15

- For any interpretation, the assertion $\neg isNull(x.n)$ holds in any state in which the `n`-field of the object `x` points-to does not have a *null* value.
- The assertion $\langle x, \epsilon\rangle \leq \chi$ holds in any state, provided that the interpretation I maps χ to an access path that starts with x; i.e., there exists a field path $\delta \in \Delta$ such that $I(\chi) = x.\delta$.
- The assertion $ass = \langle x, \epsilon\rangle \leq \chi \wedge \neg isNull(\chi)$ is satisfied if χ is mapped to an access path that points-to an object that is reachable from `x`. For example, ass is satisfied in state σ_L^c, the state in which `reverse` is invoked (see Fig. 6), w.r.t. an interpretation that maps χ to $\langle x, nnnn\rangle$ but not w.r.t. an interpretation that maps χ to either $\langle x, nnnnnnn\rangle$ or to $\langle y, nnnn\rangle$.

The following lemma states that for the same interpretation, observationally equivalent states satisfy the same assertions.

Lemma 4.16 (Equivalence w.r.t. $\mathbf{Assn_{AP}}$) *Let p be a function. Let $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ be observationally equivalent states. For any assertion $ass \in Assrt_p$ and for any interpretation $I \in AccPathVar \to AccPath_p$, $\sigma_L \models_L^I ass \iff \sigma_G \models_G^I ass$.*

$$
\begin{array}{ll}
(1) & \begin{array}{lllll} e & \in & \mathit{EAccPath}_q & ::= & \alpha \mid \chi \mid e.f \\ \alpha & \in & \mathit{AccPath}_q & & \\ \chi & \in & \mathit{AccPathVar} & & \\ f & \in & \mathit{FieldId} & & \end{array} \\[1em]
(2) & \begin{array}{lllll} \mathit{atmAss} & \in & \mathit{AtmAssrt}_q & ::= & \mathit{al}(e_1, e_2) \mid \mathit{isNull}(e) \mid e_1 \leq e_2 \\ \mathit{ass} & \in & \mathit{Assrt}_q & ::= & \mathit{atmAss} \mid \neg \mathit{ass} \mid \mathit{ass}_1 \wedge \mathit{ass}_2 \mid \\ & & & & \exists \chi. \mathit{ass} \end{array} \\[1em]
(3) & \begin{array}{lll} \sigma_L \models^I_L \mathit{al}(e_1, e_2) & \iff & [\![\hat{I}(e_2) = \hat{I}(e_2)]\!]_L(\sigma_L) \\ \sigma_L \models^I_L \mathit{isNull}(e) & \iff & [\![\hat{I}(e) = \mathtt{null}]\!]_L(\sigma_L) \\ \sigma_L \models^I_L e_1 \leq e_2 & \iff & \hat{I}(e_2) \leq \hat{I}(e_2) \end{array} \\[1em]
(4) & \begin{array}{lll} \sigma_G \models^I_G \mathit{al}(e_1, e_2) & \iff & [\![\hat{I}(e_2) = \hat{I}(e_2)]\!]_G(\sigma_G) \\ \sigma_G \models^I_G \mathit{isNull}(e) & \iff & [\![\hat{I}(e) = \mathtt{null}]\!]_G(\sigma_G) \\ \sigma_G \models^I_G e_1 \leq e_2 & \iff & \hat{I}(e_2) \leq \hat{I}(e_2) \end{array}
\end{array}
$$

Figure 13: (1) The extended syntactic category $\mathit{EAccPath}_q$ of access paths for function q. (2) Syntax of assertions in $\mathbf{Assn_{AP}}$ for function q. We also use the symbols $<$, $\vee$, $\Longrightarrow$, and $\forall$ as shorthands in a standard manner. (3) The satisfaction relation between states, interpretations, and atomic assertions in the $\mathcal{LSL}$ semantics. (4) The satisfaction relation in the $\mathcal{GSB}$ semantics.

An immediate corollary of Lem. 4.16 is that we can strengthen Cor. 4.7 in the following manner:

Corollary 4.17 *Let P be a program, p a function, and lb a program point in p. A closed assertion $ass \in \mathit{Assrt}_p$ is an invariant of P at* lb *according to the $\mathcal{LSL}$ semantics iff ass is an invariant of P at* lb *according to the $\mathcal{GSB}$ semantics.*

Example 4.18 The following assertions are invariants of the running example at lb_c:

- $\neg \mathit{isNull}(x)$, expressing that `x` points-to an object, and
- $\exists \chi, \langle x, \epsilon \rangle < \chi \ \wedge \ \mathit{isNull}(\chi)$, expressing that `x` points-to an acyclic list (e.g., *null*-terminated).

Having defined the satisfaction relation between interpretations, states in $\mathcal{LSL}$ (resp. $\mathcal{GSB}$) and assertion in $\mathbf{Assn_{AP}}$, we define the notion of *validity* of partial (resp. total) correctness assertions in $\mathcal{LSL}$ (resp. $\mathcal{GSB}$) in a standard manner (see [42, Ch. 6]). The following theorem states that $\mathcal{LSL}$ preserves validity of partial and total-correctness assertions expressed in $\mathbf{Assn_{AP}}$.

Theorem 4.19 (Preservation-of-Correctness Assertion) *Let p be a function. Let $P, Q \in Assrt_q$ be arbitrary assertions in $\mathbf{Assn_{AP}}$ for function p. Let st be an arbitrary statement in p.*

(i) A partial correctness assertion $\{P\}\ st\ \{Q\}$ is valid in $\mathcal{LSL}$ if and only if it is valid in $\mathcal{GSB}$.

(ii) A total correctness assertion $[P]\ st\ [Q]$ is valid in $\mathcal{LSL}$ if and only if it is valid in $\mathcal{GSB}$.

Example 4.20 The partial-correctness assertion $\{ass_1\}\mathtt{reverse}(\mathtt{x})\{ass_2\}$ where

$$\begin{array}{rcl} ass_1 & = & \exists\chi_1,\quad \langle x,\epsilon\rangle\leq\chi_1 \wedge al(\chi_2,\chi_1) \wedge al(\chi_3,\chi_2.n) \wedge \\ & & \qquad \forall\chi_1'\leq\chi_1\colon (\forall\chi_2' < \chi_2\colon \neg al(\chi_1',\chi_2')) \wedge \\ & & \qquad\qquad (\forall\chi_3' < \chi_3\colon \neg al(\chi_1',\chi_3')) \\ ass_2 & = & \exists\chi_4\colon \langle y,\epsilon\rangle\leq\chi_4 \wedge al(\chi_4,\chi_3) \wedge al(\chi_3.n,\chi_2) \end{array}$$

is valid in state σ_L^c, the state in which `reverse` is invoked (see Fig. 6). It asserts that the invocation of `reverse(x)` results in a reversed list. Note that the access-path variables χ_2 and χ_3 are bounded to the same access paths in the state before the call and in the state after the call. The precondition assumes that in the state before the call, χ_2 points-to the predecessor of the object that χ_3 points-to in `x`'s list, and, in addition, that no prefix of either χ_2 or χ_3 points-to an object in `x`'s list. The postcondition ensures that after the call, χ_2 points-to the successor of the object that χ_3 points-to in the returned list.

5 Abstract Interpretation

In this section, we use the $\mathcal{LSL}$ semantics to automatically compute a safe approximation to the set of possible program states using an iterative abstract-interpretation algorithm. The main idea is that every abstract state finitely represents a potentially infinite number of concrete $\mathcal{LSL}$ states. The program is interpreted according to an abstract semantics ($\overset{L^\sharp}{\rightsquigarrow}$) that over-approximates the concrete transition relation ($\overset{L}{\rightsquigarrow}$). Termination of the the abstract-interpretation algorithm is guaranteed by the finiteness of the set of abstract states.

The algorithm is *conservative*, it describes any memory state that can arise (at any program point) in any execution. This means that we can conservatively determine properties of the program such as the absence of null-dereferences, absence of garbage, and validity of invariants by checking these properties on the (generated) abstract states. However, because the description is *conservative*, the algorithm might represent concrete states that are infeasible according to the concrete semantics. This leads to incompleteness in the sense that we may fail to establish assertions that hold for every execution.

Neither Sec. 5.1 nor Sec. 5.2 gives the full details of the analyses. In particular, the abstract transfer functions are not defined. This paper focuses on the abstraction of $\mathcal{LSL}$ memory states. We plan to report on the shape-analysis algorithm in more details once its implementation is complete.

5.1 The May-Alias Abstraction

In this section, we show that Deutsch's abstract-interpretation algorithm [15] can be seen as an abstraction of the $\mathcal{LSL}$ semantics. Also, we provide insight into the clever interprocedural aspects of the analysis. App. B provides a more detailed description of [15] than the description provided in this section. It also gives the actual details of the Galois connection.

May-alias algorithms find an upper approximation for the sets of aliased access paths at every program point. The algorithm of [15] is interprocedural, flow-sensitive, and context-sensitive. It handles dynamically allocated memory, recursive functions, and recursive data structures. The algorithm computes (in polynomial time) a (bounded) representation of all the pairs of aliased access paths at every program point.

One of the most intricate aspects of the interprocedural analysis in [15] is the delayed propagation of the effect of destructive updates performed by an invoked function on pending access paths. The algorithm does not represent pending access paths explicitly. Instead, it tracks the effect of the function body on field paths that start at—what we call—cutpoints of the invocation. In particular, it represents (values of) current access paths and (values of) pending access path differently.

This simple observation suffices to see why the analysis of `revr`, a *recursive* function that (destructively) reverses a singly linked list (shown in Fig. 14, originally in [15]) manages to verify that reversing an acyclic list returns an acyclic list, whereas the analysis fails to verify this property for a list-reversal function that uses a loop, e.g., our running example.

The function `revr` reverses a list recursively by invoking itself with the tail (`t`) of the (original) list, which is not reversed yet, and a pointer to the already reversed part (`r`). The analysis handles the destructive update precisely because it can distinguish between the value of `t` in the current call and its values in pending calls by abstracting them differently. However, in the analysis of the loop-based `reverse` function in our running example (where variable `p` plays the same role as `t` in Fig. 14) , the analysis cannot distinguish between the value of `p` in the different iterations. Note that this loss of information is inherent in the may-alias analysis. In particular, it does not depend on the algorithm that abstracts the access paths.

5.2 Interprocedural Shape Analysis with Local-Heaps

In this section, we present a new interprocedural shape-analysis algorithm for programs that manipulate singly-linked lists. The algorithm finds a finite description of all the memory states that arise during program execution. Useful information regarding the program's behavior can be extracted from the computed descriptors. For example, an analysis of the running example successfully verifies that the program does not

```
Sll revr(Sll t, Sll r):=
  Sll tn;
  if (t == null) then
    ret = r
  else
    tn = t.n;
    l_d:  t.n = r;
    ret = revr(tn, t);
  fi
```

Figure 14: A function that *recursively* reverses a list.

reference null; does not create garbage; and that when `reverse` returns, the variables `z` and `y` point-to acyclic linked lists with a shared tail.

The algorithm is flow-sensitive and context-sensitive. It creates a *summary transformer* for each function p by tabulating input/output descriptors. The tabulation is restricted to input descriptors that occur at the entry to p. The algorithm is sound by construction: it is an abstract interpretation [11] of $\mathcal{LSL}$.

The algorithm is presented in terms of the 3-valued-logic framework for program analysis of [40]. This framework provides for the automatic generation of abstract interpreters (i.e., analysis algorithms) based on a specification of the programming language's concrete semantics. The most demanding task on the analysis designer is the choice of the memory-state properties that the analysis should track. Once the choice is made, the rest of the algorithm is synthesized in a provably-correct fashion. Technically, 3-valued logical structures are used to represent unbounded memory states. The tracked properties are encoded as predicates.

In this paper, we focus on the abstraction of $\mathcal{LSL}$ memory states. Due to lack of space, we do not give the full details of the analyses. In particular, the abstract transfer functions are not defined. Instead, we specify the analysis using the *best abstract transformer* [12]. A detailed description of the shape-analysis algorithm is given in [39].

5.2.1 Representing $\mathcal{LSL}$ Memory States by 3-Valued Logical Structures

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of $\frac{1}{2}$ (unknown) for cases in which predicates could have either value, 1 (true) or 0 (false). We say that 0 and 1 are *definite* values, whereas $\frac{1}{2}$ is an *indefinite* value. The information partial order on the set $\{0, \frac{1}{2}, 1\}$ is defined as $0 \sqsubseteq \frac{1}{2} \sqsupseteq 1$, and $0 \sqcup 1 = \frac{1}{2}$.

A 3-*valued logical structure* S is comprised of a set of individuals (nodes) called a universe, denoted by U^S, and an interpretation over that universe for a (finite) set of predicate symbols. The interpretation of a predicate symbol p in S is denoted by p^S. For every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \to \{0, \frac{1}{2}, 1\}$. A 2-valued structure is a 3-valued structure with an interpretation limited to $\{0, 1\}$. The set of *2-valued* logical structure is denoted by *2-Struct*, and the set of *3-valued* logical

$$
\boxed{\begin{array}{l}
\textit{to2VLS}\colon \Sigma_L \to \textit{2-Struct} \text{ s.t.} \\
\textit{to2VLS}(\langle \textit{CPL}, A\rangle) = S \text{ where } U^S = A \cup \textit{CPL} \text{ and} \\
\quad \begin{array}{lcl}
\mathit{isList}^S(v) & = & v \in A \\
\mathit{isLabel}^S(v) & = & v \in \textit{CPL} \\
x^S(v) & = & v \in A \text{ and } x \in v \\
n^S(v_1, v_2) & = & v_1 \in A, v_2 \in A \text{ and } v_1.n \subseteq v_2 \\
r_x^S(v_1) & = & \exists \alpha \in v_1 \text{ s.t. } \langle x, \epsilon\rangle \le \alpha \\
\mathit{ils}^S(v) & = & \exists \alpha.n \in v,\, \beta.n \in v \text{ s.t. } [\alpha]_A \neq [\beta]_A \\
c^S(v) & = & \exists \alpha \in v,\, \beta \in v \text{ s.t. } \alpha < \beta \\
\mathit{eq}^S(v_1, v_2) & = & v_1 = v_2 \\
\mathit{lbl}^S(v_1, v_2) & = & v_1 \in \textit{CPL},\, v_2 \in A \text{ and } \langle v_1, \epsilon\rangle \in v_2 \\
\mathit{cp}^S(v) & = & \exists r \in \textit{CPL} \text{ s.t. } \langle r, \epsilon\rangle \in v \\
r_{cp}^S(v) & = & \exists r \in \textit{CPL},\, \delta \in \Delta \text{ s.t. } \langle r, \delta\rangle \in v
\end{array}
\end{array}}
$$

Figure 15: The function *to2VLS* maps states in Σ_L to 2-valued logical structures.

structures is denoted by *3-Struct*.

To establish the Galois connection between the set of program states (ordered by set inclusion) and *3-Struct*, it suffices to show a *representation function* that maps a program state to its "most-precise representation" in *3-Struct* (e.g., see [31]). We define the function $\beta_{\mathit{shape}}\colon \Sigma_L \to$ *3-Struct*, which maps a local-heap to its most precise representation as a 3-valued logical structure. β_{shape} is a composition of two functions: (i) *to2VLS*: $\Sigma_L \to$ *2-Struct*, which maps a local-heap σ_L to an unbounded *2-valued* logical structure S, and (ii) *canonical abstraction*: *2-Struct* $\to$ *3-Struct* which conservatively bounds S (defined as usual in [40]). The Galois connection $(2^{\Sigma_L}, \alpha\colon 2^{\Sigma_L} \to 2^{\textit{3-Struct}}, \gamma\colon 2^{\textit{3-Struct}} \to 2^{\Sigma_L}, 2^{\textit{3-Struct}})$ is defined in a standard manner:

$$\alpha(\mathit{AA}) = \{\beta_{\mathit{shape}}(\sigma_L) \mid \sigma_L \in \mathit{AA}\} \text{ and } \gamma(\mathit{SS}) = \{\sigma_L \in \Sigma_L \mid \beta_{\mathit{shape}}(\sigma_L) \in \mathit{SS}\}.$$

5.2.1.1 Representing a Local-Heap by a 2-Valued Logical Structure. The function *to2VLS*, defined in Fig. 15, maps a local heap $\sigma_L = \langle \mathit{CPL}, A\rangle$ to a *2-valued* logical structure S. Every object $o \in A$ and every cutpoint-label $\mathit{cpl} \in \mathit{CPL}$ is represented by a unique node in U^S. Tracked properties of the memory state are recorded by the predicates given in Tab. 1. We denote the set of predicates used to represent a memory state by $\mathcal{P}$.

2-valued logical structures are depicted as directed graphs. A directed edge between nodes u_1 and u_2 that is labeled with binary predicate symbol p indicates that $p^S(u_1, u_2) = 1$. Also, for a unary predicate symbol p, we draw p inside a node u when $p^S(u) = 1$; conversely, when $p^S(u) = 0$ we do not draw p in u.

We explain the predicates' intended meanings through an example. In the example, we apply *to2VLS* to σ_L^e, the memory state at the entry point of `reverse` (shown in Fig. 6). The resulting 2-valued logical structure, denoted by S_e, is depicted in Fig. 16.

Predicate	Intended Meaning
$isList(v)$	Is v a list element?
$isLabel(v)$	Is v a cutpoint-label?
$x(v)$	Is v pointed-to by a (current) variable `x`?
$n(v_1, v_2)$	Does the `n`-field of v_1 point-to v_2?
$r_x(v)$	Is v_2 reachable from (current) variable `x` using `n`-fields?
$ils(v)$	Is v *locally* shared? i.e., is v pointed-to by more than one `n`-fields of objects in the *local-heap*?
$c(v)$	Does v reside on a directed cycle of `n`-fields?
$eq(v_1, v_2)$	Are v_1 and v_2 the same object or cutpoint-label?
$lbl(v_1, v_2)$	Is list element v_2 labeled by cutpoint-label v_1?
$cp(v)$	Is list element v a cutpoint?
$r_{cp}(v)$	Is the list element v reachable from a cutpoint using `n`-fields?

Table 1: The predicates used to represent states in Σ_L. There are separate predicates x and r_x for every program variable `x`.

The universe of S_e contains six nodes. The nodes u_0–u_3 represent the list elements. The node u_6 represents the cutpoint-label $\{\widehat{h.n.n.n}\}$.

- The predicates $isList$ and $isLabel$ record whether a node represents a list element or a cutpoint. We draw nodes u that represent list elements, i.e., $isList^S(u) = 1$, as rectangles, e.g., nodes u_0–u_3; and we draw nodes v that represent cutpoint-labels, i.e., $isLabel^S(v) = 1$, as circles, e.g., node u_6.

- The predicates ***h***, ***n***, r_h, ils, c, and eq are an adaptation to local-heaps of the standard predicates used in the analysis of singly linked lists [28, 40].

  - For each pointer variable `h`, there is a unary predicate ***h***. The value of $h^S(u)$ is 1 if variable `h` points-to the list element represented by u. The value of the h-predicate is depicted via an edge from the predicate name h to the node that represents the list element that `h` points-to. In Fig. 16, only node u_0 is pointed-to by a variable.

  - The pointed-to-by-a-field relation between list elements is represented by the binary predicate ***n***, i.e., $n^S(v_1, v_2) = 1$ if the `n`-field of the list element represented by v_1 points-to the list element represented by v_2. For example; the `n`-labeled edge from u_0 to u_{1a} indicates that u_{1a} represents the `n`-successor of the list element represented by u_0; u_{1b} represents the successor of u_{1a}, etc.

  - The unary predicate r_h holds for list elements that are reachable by an access path that starts at a local variable `h` of the *current* call. In σ_L^e, all the list elements are reachable from `h`. Thus, in S_e, the value of the predicate r_h is 1 for all the nodes that represent list elements.

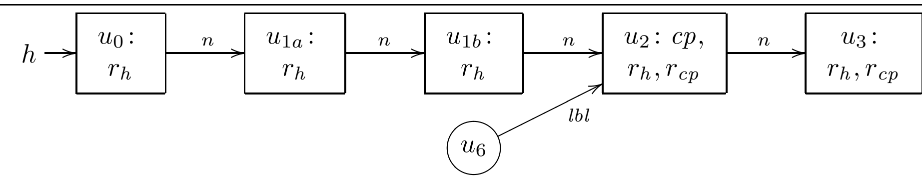
- The unary predicate ils captures *local-heap* sharing information. The predicate has the value 1 at a node u that represents a list element that is pointed-to by the `n`-fields of two or more list elements in the *local heap*. In σ_L^e, no list element is locally shared. Thus, the value of ils^{S_e} is 0 for all of the nodes in U^{S_e}. Note that the predicate records only *local* sharing. In particular, $ils^{S_e}(u_2) = 0$, although in a "global-view" of the heap, the list element represented by u_2 is the `n`-successor of two list elements: one in the local heap (represented by u_{1b}) and one not in the local heap (the third element in the list pointed-to by `z`).
- The unary predicate c holds at a node that resides on a cycle of `n`-fields. Because the list pointed-to by `h` is acyclic, $c^{S_e}(u) = 0$ for all the nodes.
- The binary predicate eq records the equality relation. It is not drawn in the pictures.

- The predicates lbl, cp, and r_{cp} record information that is special for the abstraction of an $\mathcal{LSL}$ state.
  - The binary predicate lbl relates a node that represents a cutpoint-label to the node that represents the corresponding cutpoint. For example, $lbl^S(u_6, u_2) = 1$, because u_6 represents the label of the cutpoint represented by u_2.
  - The unary predicate cp records the property that a list element is a cutpoint, e.g., $cp^{S_e}(u_2) = 1$ because u_2 represents the (only) cutpoint in S_e; for all other nodes u, $cp^{S_e}(u) = 0$.
  - The unary predicate r_{cp} records the property that a list element is reachable by a cutpoint-anchored path. For example, $r_{cp}^{S_e}(u_2) = 1$ and $r_{cp}^{S_e}(u_3) = 1$ because (only) u_2 and u_3 represent list elements that can be reached from the cutpoint (by the cutpoint-anchored paths $\langle\{\widehat{h.n.n.n}\}, \epsilon\rangle$ and $\langle\{\widehat{h.n.n.n}\}, n\rangle$, respectively). For all other nodes u, $r_{cp}^{S_e}(u) = 0$.

The predicates cp and r_{cp} are used to record information regarding cutpoint-anchored paths in a similar manner to the way h and r_h record information regarding access-paths. However, unlike local variables, the number of cutpoints is unbounded. Thus, we cannot have a predicate recording the reachable list-elements from every cutpoint. Instead, we use individuals to represent cutpoint-labels, and "mark" cutpoint objects with the cp predicate.

5.2.1.2 Canonical Abstraction. The main idea in canonical abstraction is to represent several list elements (or cutpoint-labels) by a single node, i.e., the mapping from list elements and cutpoint-labels to the universe of the *3-valued* logical structure is a surjective function, but not necessarily an injective function. A node that represents more than one list element (or more than one cutpoint-labels), is called a *summary* node.

Informally, the *3-valued* logical structure $S^\sharp$ that (conservatively) represents a memory-state σ_L is obtained by "merging" *all* the nodes in the *2-valued* logical structure $S =$ *to2VLS*(σ_L) that have the same values for *all* the unary predicates (and using these values for the unary predicates at the "merged" node). The value of a binary predicate



$$u_0 = \{\, h \,\}, \quad u_{1a} = \{\, h.n \,\}, \quad u_{1b} = \{h.n.n\}, \quad u_2 = \left\{ \begin{array}{c} h.n.n.n, \\ \{\widehat{h.n.n.n}\} \end{array} \right\}, \quad u_3 = \left\{ \begin{array}{c} h.n.n.n.n, \\ \{\widehat{h.n.n.n}\}.n \end{array} \right\},$$

$$u_6 = \{\widehat{h.n.n.n}\}$$

Figure 16: The 2-valued logical structure that results by applying *to2VLS* to σ_L^e, the memory state at the entry point of `reverse` (σ_L^e is shown in Fig. 6). We denote this structure by S_e.

$p^{S^\sharp}(u_1^\sharp, u_2^\sharp)$ is set to a *definite* value (0 or 1) only when the predicate $p^S(u_1, u_2)$ has this value for all the nodes u_1 and u_2 in U^S that are "merged" into $u_1^\sharp$ and $u_2^\sharp$, respectively.

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

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

We say that a node $u^\sharp \in U^{S^\sharp}$ **represents** node $u \in U$, when $f(u) = u^\sharp$.

By definition [40, Def 3.4.1] every *2-valued* logical structure has a *3-valued* logical structure that is its canonical abstraction.

> **Example 5.1** The *3-valued* logical structure $S_e^\sharp$, depicted in Fig. 17 (first row, second column), (conservatively) represents the memory state σ_L^e, represented by S_e.
>
> *3-valued* logical structures are also drawn as directed graphs. Definite values 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.
>
> The universe of S_e contains 6 nodes. The only nodes that have the same values for all the unary predicates are u_{1a} and u_{1b}. Thus, the universe of $S_e^\sharp$ contains five nodes. The mapping $f\colon U^{S_e} \to U^{S_e^\sharp}$ induced by the canonical abstraction is $f(u_0) = u_0^\sharp$, $f(u_{1a}) = f(u_{1b}) = u_1^\sharp$, $f(u_2) = u_2^\sharp$, $f(u_3) = u_3^\sharp$, and $f(u_6) = u_6^\sharp$. Note that the value of every unary predicate is the same for a node $u \in U^S$ and for the node that represents u in $U^{S^\sharp}$.

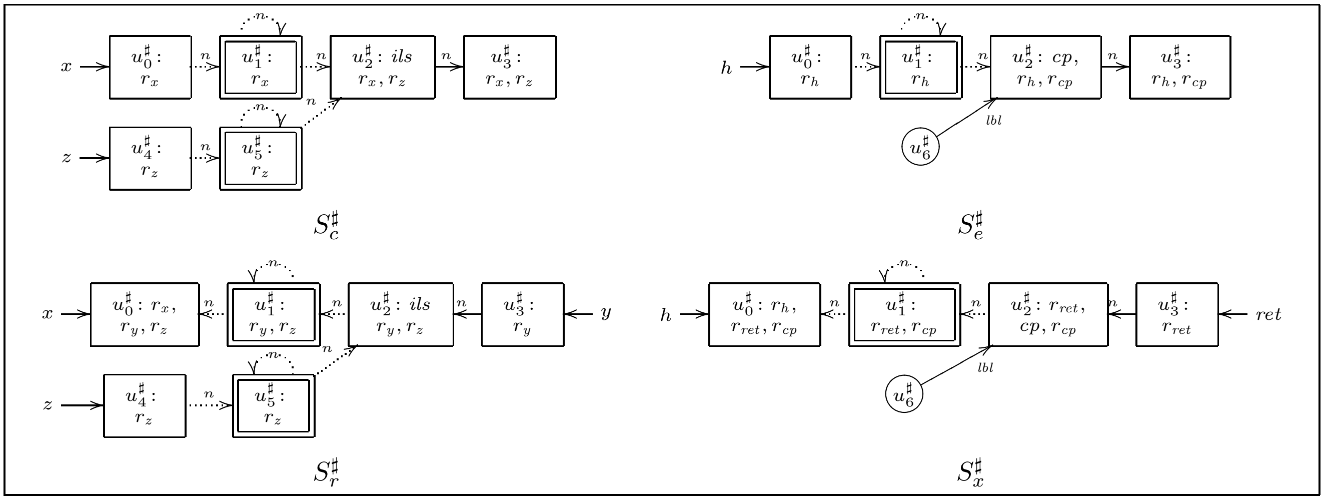


Figure 17: Representative *3-valued* logical structures that arise during the analysis of the running example at lb_c, the call-site to `reverse` (first row, first column); lb_e, the entry to `reverse` (first row, second column); lb_x, `reverse`'s exit point (second row, second column); and lb_r, the return-site from `reverse` (second row, first column).

> The only summary node in $S_e^\sharp$ is $u_1^\sharp$, which represents both u_{1a} and u_{1b}. This is recorded by the predicate eq, which has an indefinite value at $u_1^\sharp$, i.e., $eq^{S_e^\sharp}(u_1^\sharp, u_1^\sharp) = \frac{1}{2}$. The value of $n^{S_e^\sharp}(u_0^\sharp, u_1^\sharp)$ is indefinite because the `n`-field of the first list element (represented by $u_0^\sharp$) points-to the second list element (represented by $u_1^\sharp$) but not to the third list element (also represented by $u_1^\sharp$).
>
> We see that any memory state represented by $S_e^\sharp$ contains one cutpoint label (the node $u_6^\sharp$ is not a summary node). The cutpoint is represented by $u_2^\sharp$. This is recorded in two ways: (i) the value of the predicate $lbl^{S_e^\sharp}(u_6^\sharp, u_2^\sharp) = 1$ and (ii) $u_2^\sharp$ represents a list element that is labeled, as indicated by the value of the unary predicate $cp^{S_e^\sharp}(u_2^\sharp) = 1$.
>
> We also see that in any memory state represented by $S_e^\sharp$ there is no garbage (i.e., all the list elements are reachable from `h`, as indicated by the fact that the value of the predicate r_h is 1 (true) at all of them); the list pointed-to by `h` is acyclic (the value of the predicate c is 0 (false) at all the nodes); and the only cutpoint object is the list element that precedes the last element in the list. However, we no longer know the number of elements in the list.

5.2.2 Abstract Interpretation

The specification of the abstract interpretation is given by "abstract" inference rules in the same style as the natural semantics. The abstract inference rules operate on *3-*

$$\langle st, S\rangle \overset{L}{\rightsquigarrow}^{\sharp} \{\beta_{shape}(\sigma'_L) \mid \sigma_L \in \gamma(S), \langle st, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L\}$$

Figure 18: A specification of the abstract inference rules for atomic statements.

$$\frac{\langle body\ of\ p, XS_p\rangle \overset{L}{\rightsquigarrow}^{\sharp} XS'_p}{\langle y = p(x_1, \ldots, x_k), XS_q\rangle \overset{L}{\rightsquigarrow}^{\sharp} XS'_q}$$

where

$$\{Call^p_q(\sigma^c_L) \mid \sigma^c_L \in \gamma(XS_q)\} \subseteq \gamma(XS_p)$$

$$\left\{ Ret^p_q(\sigma^c_L, \sigma^x_L) \;\middle|\; \begin{array}{c} \sigma^c_L \in \gamma(XS_q), \\ \sigma^x_L \in \gamma(XS'_p), \\ compatible(\sigma^c_L, \sigma^x_L) \end{array} \right\} \subseteq \gamma(XS'_q)$$

Figure 19: A specification of the abstract inference rules for function calls. The functions $Call_q^{y=p(x_1,\ldots,x_k)}$ and $Ret_q^{y=p(x_1,\ldots,x_k)}$ are defined in Fig. 11. Note that we apply $Ret_q^{y=p(x_1,\ldots,x_k)}$ only for *compatible* pairs of memory states. Memory states σ^c_L and σ^x_L are compatible when the sharing pattern that results from the invocation of p at σ^c_L matches the description of the context in σ^x_L, the state of p at the exit-site. Formally, $compatible(\sigma^c_L, \sigma^x_L) \iff (CPL^e = CPL^x \wedge \forall h, h' \in F_p.[\![h = h']\!]_L(\sigma^e_L) \iff [\![h = h']\!]_L(\sigma^x_L) \wedge \forall h \in F_p.[\![h = \texttt{null}]\!]_L(\sigma^e_L) \iff [\![h = \texttt{null}]\!]_L(\sigma^x_L))$, where $\sigma^e_L = Call_q^{y=p(x_1,ldots,x_k)}(\sigma^c_L)$.

valued logical structures. Fig. 18 and Fig. 19 shows the specification of the abstract inference rules for atomic statements and function-calls respectively. These rules are declarative in the style of the best abstract transformer [12]: every abstract inference rule emulates a corresponding concrete inference rule using represented states (see Fig. 20).

> **Example 5.2** Fig. 17 shows an application of the function-call inference rule from Fig. 19 to the running example. The logical structures are: $S^\sharp_c$, which arises at lb_c, the call-site to `reverse`; $S^\sharp_e$, which arises lb_e, the entry to `reverse`; $S^\sharp_x$ which arises at lb_x, the exit-point of `reverse`; and $S^\sharp_r$, the structure *computed* at the return-site.
>
> In $S^\sharp_x$, the list pointed-to by `ret` is reversed. As a result, $u^\sharp_0$ is now reachable from the cutpoint at the exit-site. Therefore, even though the list-element pointed-to by`z` is not explicitly represented in $S^\sharp_x$, the inference rule allows us to conclude that at $S^\sharp_r$, the return-site's logical structure, $u^\sharp_0$

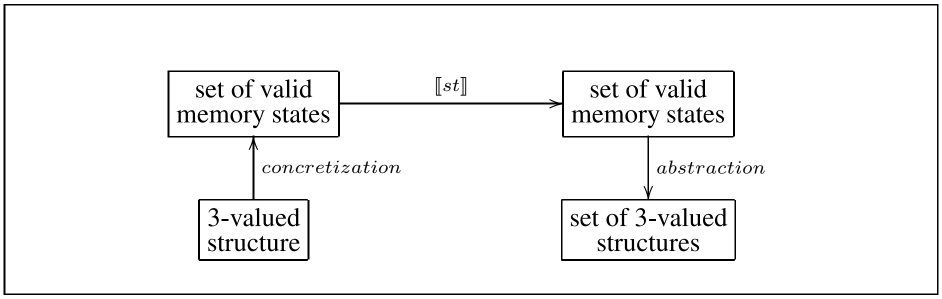


Figure 20: The best abstract semantics of a statement `st` with respect to 3-valued structures. $[\![st]\!]$ is the operational semantics of `st` applied pointwise to every admissible memory state. Conceptually, the most precise (also called *best*) conservative effect of a program statement on a 3-valued logical structure S is defined in three stages: (i) find each admissible memory state σ_L represented by S (*concretization*); (ii) apply the statement's concrete operational semantics to every such state σ_L, and (iii) abstract each of the resulting memory states by a 3-valued structure (*abstraction*).

becomes reachable from `z`. Similarly, $u_3^\sharp$ is no longer reachable from `z`. To conclude, definite values of many of the tracked properties of `z` can be established after the function call returns.

5.2.3 Discussion

In our abstraction, when a program state is mapped to a *2-valued* logical structure, no information is tracked regarding the contents of their labels. Furthermore, we do not differentiate between different cutpoints. This may lead to a significant loss of precision when multiple cutpoints arise. For example, passing two lists with shared tails will be handled very conservatively.

Nevertheless, even with this simple abstraction, our abstract domain is precise enough to analyze the singly-linked-list-manipulating programs analyzed in [23, 38] and verify that they do not dereference null-valued pointers, do not create garbage, and do not create cyclic lists. Moreover, we can handle programs not handled before by [23, 38]. For example, we can verify that a recursive function that destructively merges two acyclic lists, returns an acyclic list.

6 Related Work

6.1 Storeless Semantics

Storeless semantics was first introduced by Jonkers [24]. The original work does not handle procedure calls. Intraprocedural storeless semantics is also used in [5] to develop a logic that allows to express regular properties of unbounded data structures.

A storeless semantics that handles function-calls is defined in [14]. The semantics is used to develop a may-alias algorithm. In contrast to $\mathcal{LSL}$, in [14] pending access paths are explicitly represented.

The interprocedural may-alias algorithm of [15] uses a storeless representation of the heap. The algorithm is polynomial and can handle function calls, dynamic memory allocation and destructive updates. The algorithm is *not* shown to be an abstract interpretation of [14]. One can define a Galois connection between memory states in $\mathcal{LSL}$ with the abstract domain of [15]; see [36].

6.2 Interprocedural Shape Analysis

The original motivation for our work comes from our attempt to apply interprocedural shape analysis (e.g., [40]) to heap-manipulating programs in a modular fashion. In [35, Chap. 6] this objective was achieved, but based on a weaker technique: (i) a procedure operates on the part of the heap that is reachable from the actual parameters, where the heap is considered as an *undirected* graph; and (ii) pending access paths that point-to objects in the passed part of the heap are represented. In this paper, the heap is treated as a directed graph and pending access paths are not represented. In addition, [35] does not handle recursive procedures.

Interprocedural shape analysis has been studied in [23,38]. In [38], the main idea is to make the runtime stack an explicit data structure and abstract it as a linked list. In this method, the entire heap and run-time stack are represented at every program point. As a result, the abstraction may lose information about properties of the heap, *for parts of the heap that cannot be affected by the procedure at all*. In [23], procedures are considered as transformers from the (entire) program heap before the call, to the (entire) program heap after the call. Every heap-allocated object is represented at every program point; on the other hand, only the values of the local variables of the current procedure are represented, which means that the irrelevant parts of the heap are summarized to a single summary node during the analysis of an invoked procedure.

A modular interprocedural shape-analysis algorithm is presented in [6]. A procedure is analyzed only in the part of the heap that is reachable from its parameters. The algorithm is able to relate the memory states at the procedure-entry with the memory states at the procedure-exit by labeling *every* abstract node. However, the mapping is determined by the sharing within the part of the heap that is passed to the procedure, and not by the sharing pattern with the context—which is what is needed.

6.3 Local Reasoning

Local reasoning [22, 34] provides a way of proving properties of a procedure independent of its calling contexts by using the "frame rule", which allows proofs to be carried out in a local fashion. The main idea is to partition the heap into disjoint parts using the $*$ operator,[8] and reason about each part separately. Inferring the effect of a procedure on a heap described by $P * Q$[9] by (only) reasoning about its effect on a heap P is possible, as long as there is no need to reason about the *contents* of the heap described

[8] Mutual references between the different parts of the heap are permitted.

[9] $P * Q$ asserts that the heap can be partitioned into two disjoint parts, one satisfying P and one satisfying Q.

by Q. Although the partitioning of the heap according to the formulae $P * Q$ is not deterministic, the frame rule remains sound in any partition.

In some sense, the approach used in this paper is in the spirit of local reasoning. Our semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap. However, while the frame rule allows for an arbitrary partitioning of the heap, in our semantics, an invoked procedure operates on the subheap reachable from the actual parameters. In particular, the partitioning of the heap according to the $\mathcal{LSL}$ semantics is deterministic. (However, in the *analysis*, when the distinction between several cutpoints is lost, the analysis has to take into account every possible matching between the cutpoints at the entry-site and the cutpoint at the exit-site.) As in local reasoning, the procedure can manipulate references to objects outside the part of the heap that it may dereference.

Local reasoning relies on user-supplied specifications, e.g., loop invariants, for the reasoning. 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.

6.4 Encapsulation

Another relevant body of work is that concerning *encapsulation* (also known as *confinement* or *ownership*) [1–4, 7, 8, 17, 20, 27, 29, 33]. These works allow modular reasoning about heap-manipulating (object-oriented) programs. The common aspect of this work, as described in [32], is that they all place various restrictions on the types of data structures that a program is allowed to manipulate—in particular, on the sharing patterns permitted in the manipulated data structures. For programs that adhere to their restrictions, they provide a "frame rule". This is done, for example, in [30].

In contrast, in our work, the $\mathcal{LSL}$ semantics does not place *any* restriction on the data structures that the program uses. Also, the shape-analysis algorithm for list-manipulating programs, described in Sec. 5.2, does not place any restriction on the sharing between different lists. However, we expect that the analysis would benefit when analyzing encapsulated programs, because we anticipate that encapsulated programs would have few cutpoints.

6.5 Rule of Adaptation

The first proof rule for procedure calls, the *rule of adaptation*, was given in [18]. It allows to reuse a proof of a procedure body in different invocations of the procedure. Later work, e.g., [16, 21], simplified the use of this rule by providing a *rule of invariance*, also known as a *frame axiom*. It enables one to prove that any predicate that does not refer to variables changed by the execution of a procedure can be assumed to remain true during the execution of a call to that procedure [16]. However, [16, 16, 18, 21] do not handle heap-manipulating programs. [19] gives proof rules for heap-manipulating programs. These rules are only valid for programs that use tree-like data structures, i.e., programs that do not use sharing.

In this work we do not provide a proof system per-se. However, abstract-interpretation can be seen as a mechanism for automatic program verification [10]. As discussed in

Sec. 6.4, the $\mathcal{LSL}$ semantics does not place any restrictions on the sharing in the data structures that the program manipulates. The semantics (re)constructs the caller's heap at the return-site by (re)using the description at the call-site of that part of the heap that is not passed to the procedure. The "mimicry" of this behavior by the static-analysis algorithm can be seen as a utilization of a (reachability-based) frame rule that is "built into" the semantics. The reuse of the results of an analysis of a procedure-body for different calling-contexts with similar sharing patterns can be seen as a utilization of (a limited form of) an "adaptation rule".

7 Conclusions

In this paper, we develop $\mathcal{LSL}$, a storeless semantics for languages with dynamic memory allocation, destructive updating and procedure calls. Our storeless semantics is unique in that called procedures are only passed *parts* of the heap. We characterize the manner in which the semantics is equivalent with the standard store-based semantics. This allows us to identify a class of assertions for which the non-standard concrete semantics is equivalent to the standard store-based semantics (c.f. Cor. 4.7, Cor. 4.17, and The. 4.19). In addition, $\mathcal{LSL}$ is fully abstract, i.e., whenever two code blocks are indistinguishable in every program context, the two code blocks have the same semantics (cf. Lem. 4.11).

The development of a storeless semantics that does not represent all the heap has been challenging. Intuitively, storeless semantics means that memory locations are not explicitly represented. Instead, every dynamically allocated object O is represented by the set of pointer-access paths whose R-value equals O's l-value. In languages with destructive updates, a procedure can modify the R-value of access paths that start at variables of pending calls (i.e., pending access paths). Thus, existing storeless semantics [13, 41] represent access paths that start from pending variables, although these variables cannot be accessed by the procedure. In contrast, our semantics only represents access paths that start from visible variables. This means that a procedure has a local view that only includes objects that are reachable from the procedure's parameters.

Our main insight is that the side-effects of a procedure invocation on R-values of pending access paths can be delayed to the procedure return—even though the memory cells do not have unique identifiers, e.g., locations. The main idea is to track the effect of destructive updates on access paths that start with the set of objects that separate the part of the heap the procedure can reach from the rest of the heap (objects that we call the *cutpoints* of the invocation). A similar observation regarding the uniform effect of a procedure on pending access paths was made by [13, 26] for pointer analysis. We believe we are the first ones to use it in semantics. We believe that $\mathcal{LSL}$ can be used to justify formally previous analyses that rely on this observation by showing that these analyses are an abstract interpretation of $\mathcal{LSL}$. In App. B, we show the first stage of such a proof: establishing a Galois connection between the concrete program states and the analysis's abstract domain.

$\mathcal{LSL}$ was designed with its precise and efficient abstractions in mind: information about the context provided by the rest of the heap is isolated to the sharing patterns of the cutpoints—which are expressible in a context-independent manner. An analysis benefits from the fact that the heap is localized: the behavior of a procedure only depends on the part of the heap that is reachable from actual parameters, and on the sharing patterns that create cutpoints. Furthermore, analysis results can be reused for different contexts that have similar sharing patterns.

Using an abstraction of the non-standard concrete semantics, we present a new interprocedural shape-analysis algorithm for programs that manipulate dynamically allocated storage. Our approach is markedly different from previous works that analyze a function invocation in the calling context [23, 38]. The new algorithm can prove properties of programs that were not automatically verified before, (e.g., to establish that a recursive, destructive merge of two acyclic singly-linked lists returns an acyclic singly-linked list—see Fig. 21). In particular, it provides a way to establish properties with fewer program-specific instrumentation predicates. We believe that the modular treatment of the heap will allow the implementation of these abstractions to scale better on larger pieces of code. The approach also provides insights into an existing may-analysis algorithm [15].

Two design choices were made during the development of the new shape-analysis algorithm: One is to use a "storeless" semantics. The other is to concentrate on a superset of a program's footprint, based on reachability, rather than the actual footprint. While the ideas underlying our approach apply also to store-*based* semantics, the choice of a storeless semantics was a natural one to make (see Sec. 1.2). We specified the semantics using an equivalence relation of pointer access-paths (and not, for example, by logical structures as done in [40]) because the naming scheme we use for cutpoints (cutpoint-labels) fits naturally with the explicit manipulation of access paths done in this type of semantics. The decision to concentrate on a superset of a program's footprint (inferable via static analysis), was a pragmatic choice for the present study. In future work, we plan to investigate the use of user-supplied assertions about preserved portions of the heap.

The notion of a *cutpoint* seems to be an important concept both in storeless semantics and in store-based semantics. For instance, garbage collection of local heaps becomes unsound unless cutpoints are considered as part of the root set. Our storeless semantics takes sets of access paths as *cutpoint-labels*. This provides a context-independent representation for the cutpoints of the invocation.

In some sense, the approach used in this paper is in the spirit of local reasoning [22, 34], which provides a way to prove properties of a procedure independent of its calling contexts. In local reasoning, the "frame rule" allows proofs to be carried out in a local fashion: the main idea is to partition the heap into disjoint parts and reason about the parts separately. Our semantics resembles the frame rule in the sense that the effect of a procedure call on a large heap can be obtained from its effect on a subheap.

Limitations. The non-standard concrete semantics assumes that programs do not perform either pointer-arithmetic or casts between pointers and integers. This prevents us from handling assembly programs and non-ANSI standard C programs. It may be possible to generalize our approach to handle such features by checking that the procedure does not refer beyond the local heap. The details of such generalizations are beyond the scope of this paper.

Another limitation of our approach is that not all program properties are preserved in the non-standard concrete semantics. For instance, the property that an object is pointed to by a field of an object from outside the local heap is not preserved. We remark that our semantics preserves the following properties: (i) the values computed by arbitrary code blocks and program expressions; (ii) partial correctness for program properties expressed in the assertion language we define (see Sec. 4.3), in particular, the absence of null-dereferences and the maintenance of data-structure invariants; (iii) infinite executions and total correctness for program properties expressed using the aforementioned assertion language; and (iv) the absence of garbage.

Acknowledgments. We are grateful for the helpful comments of E. Yahav, G. Yorsh, and the anonymous referees of the POPL paper [37].

References

[1] P. S. Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241, 1997.

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

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

[4] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-03)*, volume 38:1 of *ACM SIGPLAN Notices*, pages 213–223, New York, January 15–17 2003. ACM Press.

[5] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 55–65. ACM Press, 2003.

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

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

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.

[9] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In L.D. Zuck, P.C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *Proceedings of the Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, pages 20–24, Courant Institute, NYU, New York, N.Y., USA, January 9–11 2003. LNCS 2575, Springer, Berlin.

[11] 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.*, pages 238–252, New York, NY, 1977. ACM Press.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[13] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, LIX, Ecole Polytechnique, F-91128, Palaiseau, France, 1992.

[14] A. Deutsch. A storeless model for aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE International Conference on Computer Languages*, pages 2–13, Washington, DC, 1992. IEEE Press.

[15] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[16] D. Gries and G. Levin. Assignment and procedure call proof rules. *Trans. on Prog. Lang. and Syst.*, pages 564–579, October 1980.

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

[18] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. *Lecture Notes in Mathematics*, 188:102–116, 1971.

[19] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

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

[21] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I: A logical basis and its implementation. *Acta Inf.*, 4:145 – 182, 1974.

[22] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages*, pages 14–26, 2001.

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

[24] H.B.M. Jonkers. Abstract storage structures. In de Bakker and van Vllet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[25] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

[26] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 235–248. ACM Press, 1992.

[27] K. Rustan M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 246–257. ACM Press, 2002.

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

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

[30] P. Muller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in jml. In *ECOOP 2001 Workshop on Formal Techniques for Java Programs*, 2001.

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

[32] 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. available at http://www.cs.uu.nl/∼dave/iwaco/index.html.

[33] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[34] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74, 2002.

[35] N. Rinetzky. Interprocedural shape analysis. Master's thesis, Technion Israel Institute of Technology, Haifa, Israel, 2001.

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

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

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

[39] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Tech. Rep. 26, Tel Aviv Uni., November 2004. Available at "*http://www.math.tau.ac.il/∼maon*".

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

[41] A. Venet. Automatic analysis of pointer aliasing for untyped programs. *Science of Computer Programming*, 35(2):223–248, 1999.

[42] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

```
typedef struct List{
  struct List* n; int d;
} *L;

L merge(L p, L q) {
  L r;
  if (p == NULL) return q;
  if (q == NULL) return p;
  if (p->d < q->d ) {
    r = merge(p->n,q);
    p->n = r;
    return p;
  } else {
    r = merge(p,q->n);
    q->n = r;
    return q;
  }
}
```

Figure 21: A recursive C procedure that merges two singly linked lists using destructive updates.

A Additional Code

Fig. 21 shows the code for the `merge` function. Fig. 22 shows the code for the functions `crt` and `app` used in the running example.

```
Sll crt(int k) :=
  Sll p,q;
  int t;
  if (k==0) then
    ret = null
  else
    p = alloc Sll;
    p.d = k;
    t = k-1;
    q = crt(t);
    p.n = q;
    ret = p
  fi
```

(a)

```
Sll app(Sll p,
        Sll q) :=
  Sll t1,t2;
  if (p==null) then
    ret = q
  else
    t1 = p.n;
    t2 = app(t1,q);
    p.n = t2;
    ret = p
  fi
```

(b)

Figure 22: (a) `crt` creates a list with k elements; (b) `app` destructively appends list q at the tail of list p;

B The May-Alias Abstraction

In this section we define a Galois connection between sets of $\mathcal{LSL}$ states and the abstract domain of [15] for may-alias analysis.

The algorithm of [15] computes (in polynomial time) at every program point l a set of *symbolic alias pairs* (*SAP*s). The computed set represents (in a bounded way) any pair of *current* access paths that are aliased at l. As explained in Sec. 5.1, The algorithm does not represent pending access paths explicitly. Instead, at the call-site, the algorithm *generalizes* any *SAP* representing an alias with an access path that starts at an actual parameter. The generalized *SAP* contains: (i) a representation of the access path that starts at the actual parameter, where the root of the access path (i.e., the actual parameter) is substituted by its corresponding formal parameter, and (ii) a name of a *generic object*. A generic object represent—what we call—a cutpoint of the invocation. The name of the generic object is determined uniquely by the access path it is aliased with. We denote by *APG* the set of access paths enriched with generic object names (i.e., *APG* contains access paths that start at a variable or a generic object name.).

In our terminology, generic objects are an abstraction of the cutpoints of the invocation, and the name of the generic object is an abstraction of the cutpoint-label based on its content. The use of generic object names in the analysis of a return statement is an approximation of the way cutpoint-labels are used in $\mathcal{LSL}$.

The actual representation of symbolic alias pairs (*SAP*s) is immaterial for the definition of the Galois connection. All we rely on is that the set $UR = 2^{SAP}$ of all symbolic alias relation forms a lattice ordered by $\sqsubseteq_{SAP}$ and equipped with a join operator $\sqcup_{SAP}$.[10] We make use of the function $Factor : AccPath \times AccPath \rightarrow SAP$, defined in [15], which maps a pair of unbounded (aliased) access paths, possibly starting with a generic object name, to its most precise representation by a *SAP*. We also make use of the function $makeGenericName : AccPath \rightarrow APG$, also defined in [15], which maps an access path that starts with a formal parameter to the generic object name it determines.

To establish the Galois connection between the set of program states (ordered by set inclusion) and *UR*, it suffices to show a *representation function* that maps a program state to its "most precise representation" in *UR* (e.g., see [31]). The function $\beta^p_{may}\colon \Sigma^p_L \rightarrow SAP$, defined in Fig. 23 is a representation function. It is parameterized for every function p in the program by the set of the p's local variables (V_p) and formal parameters (F_p).

The function β^p_{may} is defined as a composition of two functions: (i) $toPairs^p\colon \Sigma^p_L \rightarrow 2^{APG\times APG}$, which maps a program state of function p, $\sigma^p_L \in \Sigma^p_L$, to pairs of (unbounded) access paths enriched with object names; and (ii) $boundPairs\colon 2^{APG\times APG} \rightarrow UR$, which bounds the resulting set by mapping it to a (bounded) set of symbolic access pairs.

The function *toPairs* converts a program state to a (bounded) alias relation in two steps: (i) it creates the equivalence relation (*AP*) by pairing any two generalized access paths that belong to the same equivalence class; (ii) it "recovers" the generic object names out of any generalized access path that starts with a cutpoint or a formal pa-

[10] In [15], The set *UR* is actually parameterized by the numeric lattice used in the analysis. Since the parameterization is not relevant for our purposes, we ignore this issue.

$$
\begin{array}{l}
\beta^p_{may} \colon \Sigma^p_L \to \mathit{UR} \ \text{ s.t.} \\
\beta^p_{may} = \mathit{boundPairs} \circ \mathit{toPairs}^p \\
\\
\mathit{toPairs}^p \colon \Sigma^p_L \to 2^{\mathit{APG} \times \mathit{APG}} \ \text{ s.t.} \\
\mathit{toPairs}^p(\langle \mathit{CPL}, A \rangle) = \ \text{Let} \\
\quad \mathit{AP} = \{\langle \alpha, \beta \rangle \mid \exists a \in A, \ \text{ s.t. } \alpha \in a \text{ and } \beta \in a\} \\
\quad \mathit{in} \bigcup_{\langle \alpha, \beta \rangle \in AP} \left\{ \langle \alpha', \beta' \rangle \,\middle|\, \begin{array}{l} \alpha' \in \mathit{generic}^p(\mathit{CPL}, \alpha), \\ \beta' \in \mathit{generic}^p(\mathit{CPL}, \beta) \end{array} \right\} \\
\quad \text{Where} \\
\quad \mathit{generic}^p \colon 2^{\mathit{CPL}} \times \mathit{GAccPath} \to 2^{\mathit{APG}} \ \text{ s.t.} \\
\quad \mathit{generic}^p(\mathit{CPL}, \langle r, \delta \rangle) = \\
\qquad \left\{ \begin{array}{ll} \{\langle r, \delta \rangle\} & r \in V_p \setminus F_p \\ \{\langle r, \delta \rangle, \langle \mathit{makeGenericName}(\langle r, \epsilon \rangle), \delta \rangle\} & r \in F_p \\ \{\langle \mathit{makeGenericName}(\alpha), \delta \rangle \mid \alpha \in r\} & r \in \mathit{CPL} \end{array} \right. \\
\\
\mathit{boundPairs} \colon 2^{\mathit{APG} \times \mathit{APG}} \to \mathit{UR} \ \text{ s.t.} \\
\quad \mathit{boundPairs}(\mathit{AliasRel}) = \\
\qquad \bigsqcup_{\mathit{SAP}} \{\mathit{Factor}(\langle \alpha, \beta \rangle) \mid \langle \alpha, \beta \rangle \in \mathit{AliasRel}\}
\end{array}
$$

Figure 23: β^p_{may} is a representation function that maps a memory state of function p to its most precise representation as sets of symbolic access path.

rameter by invoking *generic*. The special treatment for formal parameters is required because [15] considers objects pointed-to by actual parameters as (trivial) cutpoints, where we do not. A bounded representation is achieved by applying *Factor* pointwise and taking the least upper bound of the resulting set of symbolic access paths.[11]

C Proofs

In this section we prove our main theorem, The. 4.6 (preservation of observational equivalence). In Sec. C.1, we prove some properties of the $\mathcal{GSB}$ semantics. In Sec. C.2, we state, and prove, additional properties of the $\mathcal{LSL}$ semantics. In Sec. C.3, we define the notion of context-aware equivalence between states in $\mathcal{LSL}$ and states in $\mathcal{GSB}$, and prove a stronger theorem than the equivalence theorem, i.e., the preservation of *context-aware equivalence*.

As in Sec. 4.2, we assume, A and *CPL* with a certain index (resp. prime) to be the heap, resp. cutpoint-labels component of a state σ_L with the same index (resp. prime). Similarly, we assume L, ρ, and h with a certain index (resp. prime) to be the set of allocated locations, resp. environment, resp. heap of a state σ_G with the same index

[11] In [15], special care needs to be taken in case the analysis is parameterized by a lattice with infinite chains. In particular, $\sqsubseteq_{SAP}$ is not necessarily bounded. For simplicity, we assume this is not the case.

(resp. prime). In addition, we use the notations $[\![\alpha \neq \beta]\!]_L(\sigma_L)$, $[\![\alpha \neq \texttt{null}]\!]_L(\sigma_L)$, $[\![\alpha \neq \beta]\!]_G(\sigma_G)$, and $[\![\alpha \neq \texttt{null}]\!]_G(\sigma_G)$ as shorthand for $\neg[\![\alpha = \beta]\!]_L(\sigma_L)$, resp. $\neg[\![\alpha = \texttt{null}]\!]_L(\sigma_L)$, resp. $\neg[\![\alpha = \beta]\!]_G(\sigma_G)$, resp. $\neg[\![\alpha = \texttt{null}]\!]_G(\sigma_G)$.

C.1 Properties of the $\mathcal{GSB}$ Semantics

In this section, we introduce the notions of *heap paths* and *generalized heap paths*. We also prove some properties of the $\mathcal{GSB}$ semantics which are used in the proof of The. 4.6.

Definition C.1 (Heap path) *A* ***heap path*** $\zeta = \langle l, \delta \rangle \in Loc \times \Delta$ *is a pair consisting of a location and a field path.* $HeapPath$ *denotes the set* $Loc \times \Delta$.

Definition C.2 (Generalized heap path) *A* ***generalized heap path*** $\zeta \in AccPath_p \cup HeapPath$ *of a function* p *is an access path of* p *or a heap path.* $GHeapPath_p$ *denotes the set of all generalized heap paths of function* p. $GHeapPath$ *denotes the union of all generalized heap paths of all functions in a program.*

Definition C.3 (Generalized heap path value) *The value of a generalized heap path* ζ *in memory state* $\langle L, \rho, h \rangle$ *of function* p *is defined to be:*

$$[\![\zeta]\!]_G \langle L, \rho, h \rangle = \begin{cases} \hat{h}(\rho(x), \delta) & \zeta = \langle x, \delta \rangle,\ x \in V_p \\ \hat{h}(l, \delta) & \zeta = \langle l, \delta \rangle,\ l \in L \end{cases}$$

where $\hat{h}$ is as defined in Def. 2.4. Note that the above definition generalizes Def. 2.4 (value of an access path in the $\mathcal{GSB}$ semantics). The following definition generalizes Def. 2.5 (equality of access paths in the $\mathcal{GSB}$ semantics).

Definition C.4 (Generalized heap path equality) *Generalized heap paths* ζ_1 *and* ζ_2 *are* ***equal*** *in a given state* σ_G*, denoted by* $[\![\zeta_1 = \zeta_2]\!]_G(\sigma_G)$*, if they have the same value in that state, i.e.,* $[\![\zeta_1]\!]_G(\sigma_G) = [\![\zeta_2]\!]_G(\sigma_G)$*. A generalized heap path* ζ *is* ***equal to null*** *in a given state* σ_G*, denoted by* $[\![\zeta = \texttt{null}]\!]_G(\sigma_G)$*, if* $[\![\zeta]\!]_G(\sigma_G) = null$*.*

The following lemma states that a function invocation cannot modify fields of objects that are allocated, but which are not reachable from an actual parameter, when a function is invoked.

Lemma C.5 (Unreachable locations not modified) *Let σ_G^c, σ_G^r be states in Σ_G such that $\langle y = p(x_1, \ldots, x_k), \sigma_G^c\rangle \overset{G}{\rightsquigarrow} \sigma_G^r$. Let $L^{reach} \subseteq L^c$ be the location in L^c that are reachable from the actual arguments at σ_G^c, i.e.,*

$$L^{reach} = \bigcup_{1 \le i \le k} \{l \in L^c \mid \exists \delta \in \Delta, l = [\![\langle x_i, \delta\rangle]\!]_G(\sigma_G^c)\}.$$

For any generalized access path $\zeta = \langle r, \delta\rangle \in GHeapPath$ such that $[\![\zeta]\!]_G(\sigma_G^c) \in L^c \setminus L^{reach}$ the following holds:

1. *$[\![\zeta]\!]_G(\sigma_G^c) = [\![\zeta]\!]_G(\sigma_G^r)$, and*
2. *for any $f \in FieldId$, $[\![\langle r, \delta f\rangle]\!]_G(\sigma_G^c) = [\![\langle r, \delta f\rangle]\!]_G(\sigma_G^r)$.*

Sketch of Proof: The lemma states that a function cannot modify the content of locations it cannot access (reach). The proof is by induction on the derivation tree. We track the set of reachable locations from every variable of the invoked function and prove that a variable cannot point-to (and thus potentially modify) locations that are allocated when the function is invoked and are not reachable from any actual parameter. Note that for any $l \in L^{reach}$ and any $\delta \in \Delta$, $[\![\langle l, \delta\rangle]\!]_G(\sigma_G^r)$ is defined because $L^c \subseteq L^r$.

The following lemma formally states that any access path that extends a null valued access path has a null value. Similarly, any prefix of a non null valued access path points to a location.

Lemma C.6 (Null valued access paths) *Let $\sigma_G \in \Sigma_G^q$ be a $\mathcal{GSB}$ state for function q,*

1. *For a generalized heap path $\alpha = \langle r, \delta_0\delta_1\rangle \in GHeapPath_q$, it holds that $[\![\alpha]\!]_G(\sigma_G) = [\![\langle [\![\langle r, \delta_0\rangle]\!]_G(\sigma_G), \delta_1\rangle]\!]_G(\sigma_G)$.*
2. *For any (generalized) access path $\alpha \in GHeapPath_q$,*
   - *a. if $[\![\alpha = \mathtt{null}]\!]_G(\sigma_G)$, then for any generalized heap path α' such that $\alpha \le \alpha'$, $[\![\alpha' = \mathtt{null}]\!]_G(\sigma_G)$,*
   - *b. if $[\![\alpha \ne \mathtt{null}]\!]_G(\sigma_G)$, then for any generalized heap path α' such that $\alpha' \le \alpha$, $[\![\alpha' \ne \mathtt{null}]\!]_G(\sigma_G)$.*

Proof: Immediate from Def. C.3 (generalized heap-path value).

C.2 Properties of the $\mathcal{LSL}$ Semantics

In this section, we prove certain properties of the $\mathcal{LSL}$ semantics. These properties are needed in the proof of the Context-Aware Equivalence theorem, however, they are mere technicalities of the definition of $\mathcal{LSL}$ as given in Fig. 10 and Fig. 11.

The following lemma establishes some of the properties of the $[\cdot]_.$ function defined in Fig. 9. In particular, it states certain properties related to equality of access-paths.

Lemma C.7 (Properties of $[\cdot]_{\cdot}$) *Let $\sigma_L = \langle \text{CPL}, A \rangle \in \Sigma_L^q$ be an (admissible) memory state for function q. For any generalized access paths $\alpha, \beta \in GAccPath_q$ the following holds:*

1. $[\![\alpha = \texttt{null}]\!]_L(\sigma_L) \iff [\alpha]_A = \emptyset$
2. $[\![\alpha = \beta]\!]_L(\sigma_L) \iff [\alpha]_A = [\beta]_A$
3. $[\![\alpha = \texttt{null}]\!]_L(\sigma_L) \implies (\forall \alpha', \alpha \leq \alpha' \implies [\![\alpha' = \texttt{null}]\!]_L(\sigma_L))$
4. $[\![\alpha \neq \texttt{null}]\!]_L(\sigma_L) \implies (\forall \alpha', \alpha' \leq \alpha \implies [\![\alpha' \neq \texttt{null}]\!]_L(\sigma_L))$
5. $[\alpha]_A \in A \cup \{\emptyset\}$
6. $\forall cpl \in CPL : [\langle cpl, \epsilon \rangle]_A \neq \emptyset$

Sketch of Proof: 1-5 are immediate from the definitions of: $[\![\cdot = \cdot]\!]_L$, $[\cdot]_{\cdot}$, and admissibility. Lem. C.7(6) is proven using an induction on the derivation tree where the key observation is that objects never lose their labels, i.e., an access path of the form $\langle cpl, \epsilon \rangle$, where $cpl \in CPL$, is never removed from the description of an object.

The following lemma states certain properties of the sets of objects and the various mappings defined in the function call inference rule (see Fig. 11).

Lemma C.8 (Properties of the function call inference rule) *Let $\sigma_L^c = \langle \text{CPL}^c, A^c \rangle \in \Sigma_L^q$ be an admissible memory state for function q in which the statement $y = p(x_1, \dots, x_k)$ is executed. Let $\langle \text{CPL}^e, A^e \rangle$, $\langle \text{CPL}^x, A^x \rangle$, $\langle \text{CPL}^r, A^r \rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, and $bind_{ret}$ be as defined in Fig. 11. Let $\alpha \in GAccPath_q$ be an arbitrary generalized access path of function q. The following holds.*

1. $\emptyset \notin O_c^{passed}$.
2. $(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x = \emptyset$.
3. *If* $[\alpha]_{A^r} \neq \emptyset$ *and* $[\alpha]_{A^r} \notin (A^c \setminus O_c^{passed})$, *then* $[\alpha]_{A^r} \in map(sub(bind_{ret}))\ A^x$.
4. $\forall a, a' \in dom(bind_{ret}) : a \neq a' \implies \forall \alpha \in bind_{ret}(a) : \forall \beta \in bind_{ret}(a') : \alpha \not\leq \beta \wedge \beta \not\leq \alpha$.
5. $\forall a \in dom(bind_{ret}) : \forall \alpha, \beta \in bind_{ret}(a) : \alpha \neq \beta \implies \alpha \not\leq \beta \wedge \beta \not\leq \alpha$.
6. $\forall a \in range(bind_{ret}) : \forall \alpha \in a. \forall \alpha' < \alpha, [\alpha']_{A^c} \notin O_c^{passed}$.
7. $\emptyset \notin range(bind_{ret})$

Proof: We only sketch the proof of 3. The other properties are derived immediately from the definition of the $\mathcal{LSL}$ semantics.

1.	$[\alpha]_{A^r} \neq \emptyset$, $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$	Assumption
2.	$A^r = (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x$	See $Fig.\ 11$.
3.	$(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x = \emptyset$	$Lem.\ C.8(2)$
4.	$[\alpha]_{A^r} \in map(sub(bind_{ret}))\ A^x$	$1-3$

The following lemma establishes certain properties of the memory states that occur during a function invocation at the call-site, at the entry-site, at the exit-site, and at the return-site. Informally, it states the following properties:

1. Properties of cutpoint-labels:

   (a) Cutpoint labels are never empty.

   (b) At the entry state, every access path in a cutpoint-label points to the corresponding cutpoint.

2. When a function returns, an access path can point to one of the following: to an object which was not passed to the function, to an object that was in the invoked function local heap, or to null.

3. When a function returns, an access path that points to an object which was not in the callee local heap does not point to such an object when the function has been invoked.

4. The function $sub(bind_{ret})$ is injective for all objects that are reachable at the return site. Furthermore, it maps all unreachable objects to the empty set.

5. When a function returns, every access path that points to the invoked function's local heap, has a unique prefix which starts either with the return value, an object pointed-to by an actual parameter, or a cutpoint of that invocation.

Lemma C.9 (Properties of function calls) *Let σ_L^c, q, $y = p(x_1, \ldots, x_k)$, $\langle \text{CPL}^e, A^e \rangle$, $\langle \text{CPL}^x, A^x \rangle$, $\langle \text{CPL}^r, A^r \rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, $bind_{ret}$, and $\alpha \in GAccPath_q$ be as in Lem. C.8. The following holds:*

1. *For any $cpl \in CPL^e$, the following holds:*

   (a) $\emptyset \neq cpl \subseteq F_q \times \Delta$, and

   (b) for any $\alpha' \in cpl$ and any $\delta \in \Delta$, $[\![\alpha'.\delta = \langle cpl, \delta \rangle]\!]_L(\sigma_L^e)$.

2. *If $[\alpha]_{A^r} \notin (A^c \setminus O_c^{passed})$, then for any generalized access path α' such that $\alpha \leq \alpha'$, $[\alpha']_{A^r} \in \{\emptyset\} \cup map(sub(bind_{ret}))\ A^x$.*

3. *If $[\alpha]_{A^r} \in (A^c \setminus O_c^{passed})$, then for every generalized access path $\alpha' \leq \alpha$ it holds that $[\alpha']_{A^c} \notin O_c^{passed}$.*

4. *For any $o, o' \in A^x$, if $o \neq o'$, then either $sub(bind_{ret})\ o \neq sub(bind_{ret})\ o'$ or $sub(bind_{ret})\ o = sub(bind_{ret})\ o' = \emptyset$.*

5. *(a) If $\langle y, \epsilon \rangle \leq \alpha$, then $[\![\langle y, \delta \rangle = \texttt{null}]\!]_L(\sigma_L^r) \iff [\![\langle ret, \delta \rangle = \texttt{null}]\!]_L(\sigma_L^x)$*

(b) *If* $\langle y, \epsilon\rangle \not\leq \alpha$ *and* $\forall \alpha' \leq \alpha.[\alpha']_{A^c} \notin O_c^{passed}$*, then* $[\![\alpha = \mathtt{null}]\!]_L(\sigma_L^r)) \iff [\![\alpha = \mathtt{null}]\!]_L(\sigma_L^c)$.

(c) *If* $[\alpha]_{A^c} \in O_c^{args} \cup O_c^{cp}$ *and* $\alpha \in Bypass(O_c^{passed})\ [\alpha]_{A^c}$ *then there exists a (unique)* $r_1^\alpha \in dom(bind_{ret})$ *such that* $\alpha \in bind_{ret}\ r_1^\alpha$*. Furthermore for any* $\delta \in \Delta$, $[\![\alpha.\delta = \mathtt{null}]\!]_L(\sigma_L^r) \iff \forall \alpha_p \in r_1^\alpha, [\![\alpha_p.\delta = \mathtt{null}]\!]_L(\sigma_L^x)$

6. *For any* $o \in A^x$ *such that* $[\alpha]_{A^r} = sub(bind_{ret})\ o$ *and* $[\alpha]_{A^r} \neq \emptyset$*, there exists a unique* $\alpha_0 \leq \alpha$ *such that* $\alpha_0 \in \mathit{flat}\,(range(bind_{ret}))$*. Furthermore, one (and only one) of the following holds:*

(a) $\alpha_0 = \langle y, \epsilon\rangle$

(b) $[\alpha_0]_{A^c} \in O_c^{args}$ *and* $\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$

(c) $[\alpha_0]_{A^c} \in O_c^{cp}$ *and* $\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$

and, in addition, $r_1^\alpha.\delta_1^\alpha \subseteq o$ *where* $\alpha = \alpha_0.\delta_1^\alpha$, $r_1^\alpha.\delta_1^\alpha \neq \emptyset$ *and*

$$r_1^\alpha = \begin{cases} \{\langle ret, \epsilon\rangle\} & (\textit{if case } 6a \textit{ holds}) \\ bind_{args}\ [\alpha_0]_{A^c} & (\textit{if case } 6b \textit{ holds}) \\ bind_{cp}\ [\alpha_0]_{A^c} & (\textit{if case } 6c \textit{ holds}) \end{cases}$$

Proof:

Properties 2–5 are immediate. We prove properties 1 and 6.

1.

(i) By definition, $CPL^e = map(sub(bind_{args}))\ O_c^{cp}$. To show that $\emptyset \notin CPL^e \subseteq 2^{F_q\times\Delta}$, we show that $\forall o \in O_c^{passed}.\emptyset \neq sub(bind_{args})\ o \subseteq F_q \times \Delta$. This proves (i), because $O_c^{cp} \subseteq O_c^{passed}$. Recall that

$$O_c^{passed} \quad = \quad RObjs(A^c)O_c^{args} = \{o \in A^c \mid o' \in O_c^{args}, \delta \in \Delta, o'.\delta \subseteq o\}$$

Thus,

5. $o \in O_c^{passed} \iff \exists o' \in O_c^{args}, \exists \delta \in \Delta.\ o'.\delta \subseteq o.$

By definition (see Fig. 11),

6. $\emptyset \notin O_c^{args}$, and

7. $bind_{args} = \lambda o \in O_c^{args}.\{\langle h_i, \epsilon\rangle \mid 1 \leq i \leq k,\ x_i \in o\}$.

Thus, for any $o \in O_c^{passed}$,

8. $$\begin{aligned} sub(bind_{args})(o) &= \mathit{flat}\,\{bind_{args}(a).\delta \mid a \in dom(bind_{args}), \delta \in \Delta, a.\delta \subseteq o\} \\ &= \mathit{flat}\,\{bind_{args}(o').\delta \mid o' \in O_c^{args}, \delta \in \Delta, o'.\delta \subseteq o\} \end{aligned}$$

which gives (by 5, 6) that $\emptyset \notin sub(bind_{args})(o)$, and (by 5, 7) that $sub(bind_{args})(o) \subseteq F_q \times \Delta$.

To prove (ii), we recall (see Fig. 11) that

9. $CPL^e \quad = \quad map(sub(bind_{args}))\ O_c^{cp}$,

10. $O_c^{cp} \quad \subseteq \quad O_c^{passed}$,

11. $bind_{cp} \quad = \quad \lambda o \in O_c^{cp}.\{\langle sub(bind_{args})\ o, \epsilon\rangle\}$, and

12. $bind_{call} \quad = \quad \lambda o \in O_c^{args} \cup O_c^{cp}.\begin{cases} bind_{args}(o) & o \in O_c^{args} \\ bind_{cp}(o) & o \in O_c^{cp} \end{cases}$,

13. $A^e \quad = \quad map(sub(bind_{call}))\ O_c^{passed}$.

Thus, for any $cpl \in CPL^e$ and for any $o \in A^e$,

14.	$\langle cpl, \delta\rangle \in o$	$\iff$	13
15.	$\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge\ \langle cpl, \delta\rangle \in sub(bind_{call}) o'$	$\iff$	
16.	$\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge$ $\langle cpl, \delta\rangle \in flat\left\{ bind_{call}(a_1).\delta_1 \,\middle\vert\, \begin{array}{c} a_1 \in dom(bind_{call}), \\ \delta_1 \in \Delta,\ a_1.\delta_1 \subseteq o' \end{array} \right\}$	$\iff$	Def. of sub
17.	$\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge$ $\exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge\ o''.\delta \subseteq o'$	$\iff$	$9, 7, 11, 12$
18.	$\exists o' \in O_c^{passed} : o = sub(bind_{call})\ o' \ \wedge$ $\exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge$ $\forall o''' \in O_c^{args} : \forall \delta' \in \Delta : o'''.\delta' \in o'' \implies o'''.\delta'.\delta \in o'$	$\iff$	Admissibility of σ_L^c $O_c^{passed} = RObjs(A^c) O_c^{args}$
19.	$\exists o' \in O_c^{passed} : \exists o'' \in O_c^{cp} : \langle cpl, \delta\rangle = \langle sub(bind_{args})\ o'', \delta\rangle \ \wedge$ $sub(bind_{args})\ o''.\delta \subseteq sub(bind_{args})\ o'$	$\iff$	18, 8, Def. of $bind_{args}$
20.	$\forall \alpha' \in cpl : \alpha'.\delta \in o$	$\iff$	19

6.

21.	$[\alpha]_{A^r} \neq \emptyset, o \in A^x$, $[\alpha]_{A^r} = sub(bind_{ret})\ o$	Assumptions
22.	$\alpha \in sub(bind_{ret})\ o$	21, Def. of $[\cdot]_.$
23.	$\alpha \in flat\ \{bind_{ret}(a).\delta_1^\alpha \mid a \in dom(bind_{ret}),\ \delta_1^\alpha \in \Delta, a.\delta_1^\alpha \subseteq o\}$	22, Def. of sub
24.	$\exists a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon a.\delta_1^\alpha \subseteq o \wedge \alpha \in (bind_{ret}\ a).\delta_1^\alpha$	23, Def. of $flat$
25.	$\exists a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon \exists \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	24, Def. of $\cdot.\cdot$
26.	$\exists! a \in dom(bind_{ret})\colon \exists \delta_1^\alpha \in \Delta\colon \exists \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	25, $Lem.\ C.8(4)$, $\alpha_0 \leq \alpha$
27.	$\exists! a \in dom(bind_{ret})\colon \exists! \delta_1^\alpha \in \Delta\colon \exists! \alpha_0 \in bind_{ret}(a)\colon a.\delta_1^\alpha \subseteq o \wedge \alpha = \alpha_0.\delta_1^\alpha$	26, $Lem.\ C.8(5)$, $\alpha_0 \leq \alpha$
28.	Let $a, \alpha_0, \delta_1^\alpha$ be the unique values satisfying 27. We continue with a case analysis of the possible values of $a \in dom(bind_{ret})$	
29.	$\alpha_0 \in flat\,(range(bind_{ret})) \wedge \forall \alpha' \leq \alpha\colon \alpha' \in flat\,(range(bind_{ret})) \implies \alpha' = \alpha_0$	$27 - 28$
30.	$dom(bind_{ret}) = \{\{\langle ret, \epsilon\rangle\}\} \cup map(bind_{args})\ O_c^{args} \cup map(bind_{cp})\ O_c^{cp}$	Def. of $bind_{ret}$
31.	Assume $a = \{\langle ret, \epsilon\rangle\}$ (Case 6a)	
32.	$bind_{ret}(\{\langle ret, \epsilon\rangle\}) = \{\langle y, \epsilon\rangle\}$	Def. of $bind_{ret}$
33.	$\alpha_0 = \langle ret, \epsilon\rangle$	29, 32
34.	$\{\langle ret, \epsilon\rangle\}.\delta_2 \subseteq o$	27, 28, 33
35.	Assume $a \in map(bind_{args})\ O_c^{args}$ (Case 6b)	
36.	$\exists o' \in O_c^{args}, a = bind_{args}\ o'$	35
37.	$\alpha_0 \in bind_{ret}(a)$	28
38.	$\exists o' \in O_c^{args}, a = bind_{args}\ o'$, $\alpha_0 \in Bypass(O_c^{passed})\ o'$	$35 - 37$, Def. of $bind_{ret}$
39.	$a = bind_{args}([\alpha_0]_{A^c})$, $\alpha_0 \in Bypass(O_c^{passed})\ [\alpha_0]_{A^c}$	38, Def. of $bind_{ret}$ and $Bypass$
40.	$bind_{args}\ [\alpha_0]_{A^c}.\delta_1^\alpha \subseteq o$	28, 39
41.	Assume $a \in map(bind_{args})\ O_c^{args}$ (Case 6c)	
42.	proof analogous to case 6b	

Note that, by Lem. C.8(7), $r_\alpha^1 \neq \emptyset$.

In the following, we sketch the proofs of additional properties of $\mathcal{LSL}$ which are stated in Sec. 4.2.

Sketch of Proof (The. 4.12):

(i) For access paths that in memory state σ_L^c point to an object which is not in O_c^{passed}, the proof is immediate from admissibility of σ_L^r and the fact that $A^c \setminus O_c^{passed} \subseteq A^r$.

For accesses paths α, β that in memory state σ_L^c point-to (the same) object in O_c^{passed}, but do not *pass through* any object in O_c^{passed}, the proof follows from Lem. C.9(5c).

(ii) For access paths that are equal *null* in σ_L^c, the proof is immediate from Lem. C.9(5b).

Sketch of Proof (The. 4.13):

The proof is done by induction on the shape of the derivation tree. The base case is immediate because in every statement in both σ_L^1 and σ_L^2:

- the same set of access paths that start with a variable, are added / removed from the description of every object, and
- the side-conditions for executing a statement involve only access paths that start with a variable.

The induction step for (non-atomic) intraprocedural statements is also immediate because of the aforementioned nature of side-conditions in the $\mathcal{LSL}$ semantics. To see why the induction step holds for a function call, we observe that in both σ_L^1 and σ_L^2:

- the same objects are reachable from the actual parameters, and
- at function return, the update of access paths that start with a variable, is done using the same cutpoints.

C.3 Context-Aware Equivalence

In this section, we state and prove the context-aware equivalence theorem (The. C.15). The. 4.6 is an immediate corollary of The. C.15.

Definition C.10 (Renaming function) *Given an $\mathcal{LSL}$ state $\langle \text{CPL}, A\rangle$ of function p, and a $\mathcal{GSB}$ state $\langle L, \rho, h\rangle$ of function p, a function $f\colon CPL \to L$ is a* ***renaming function*** *if it is total and injective. We lift f to $\hat{f}\colon GAccPath_p \to GHeapPath_p$ as follows:*

$$\hat{f}(\langle r, \delta\rangle) = \begin{cases} \langle r, \delta\rangle & : \quad r \in V_p \\ \langle f(r), \delta\rangle & : \quad \textit{otherwise} \end{cases}$$

Definition C.11 (Context-Aware Equivalence) *Let p be a function. The **states** $\sigma_L = \langle \text{CPL}, A\rangle \in \Sigma_L^p$ and $\sigma_G = \langle L, \rho, h\rangle \in \Sigma_G^p$ are **context-aware equivalent w.r.t. a renaming function** $f\colon CPL \to L$, denoted by $\sigma_L \propto_f \sigma_G$, if for all $\alpha, \beta, \gamma \in GAccPath_p$,*

1. $[\![\alpha = \beta]\!]_L(\sigma_L) \iff [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G)$,
2. $[\![\gamma = \texttt{null}]\!]_L(\sigma_L) \iff [\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_G)$.

*The states σ_L and σ_G are **context-aware equivalent** if there exists a renaming function f s.t. $\sigma_L \propto_f \sigma_G$.*

The following lemma is rather technical. It states that any extension of a renamed access path points to the same location as the renamed extended access path.

Lemma C.12 *Let $\sigma_L \in \Sigma_L^q$ and $\sigma_G \in \Sigma_G^q$ be context-aware equivalent states w.r.t a renaming function f. For any $\alpha, \alpha_0 \in GAccPath_q$ and any $\delta \in \Delta$ such that $\alpha = \alpha_0.\delta$, $[\![\hat{f}(\alpha)]\!]_G(\sigma_G) = [\![\hat{f}(\alpha_0).\delta]\!]_G(\sigma_G) = [\![\langle [\![\hat{f}(\alpha_0)]\!]_G(\sigma_G), \delta\rangle]\!]_G(\sigma_G)$.*

Proof: Immediate from the definition of $\hat{f}$, the definition of $\cdot.\cdot$, and Lem. C.6(1).

The following lemma shows that context-aware equivalence at the call-site, implies context aware equivalence at the entry-site. Furthermore, it defines an appropriate renaming function (f_e). Property 1 shows that f_e is indeed a renaming function and Property 2 proves that the entry states are context aware equivalent with respect to f_e. Properties 3–5 establish certain properties of f_e.

Lemma C.13 (Context-aware equivalence of invoked functions)
Let $\sigma_L^c = \langle \text{CPL}^c, A^c\rangle \in \Sigma_L^q$ and $\sigma_G^c = \langle L^c, \rho^c, h^c\rangle \in \Sigma_G^q$ be context-aware equivalent states w.r.t. a renaming function f, i.e., $\sigma_L^c \propto_f \sigma_G^c$. Let $y = p(x_1, \ldots, x_k)$ be a call to function p whose formal parameters are $h_1, \ldots, h_k$. Let $\sigma_L^e = \langle \text{CPL}^e, A^e\rangle$, $\sigma_L^x = \langle \text{CPL}^e, A^x\rangle$, $\sigma_L^r = \langle \text{CPL}^c, A^r\rangle$, O_c^{args}, O_c^{passed}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, and $bind_{call}$ be as defined in Fig. 11. Let $\sigma_G^e = \langle L_e, \rho_e, h_e\rangle$, $\sigma_G^x = \langle L_x, \rho_x, h_x\rangle$, and $\sigma_G^r = \langle L_r, \rho_r, h_r\rangle$ be as defined in Fig. 5. Let L^{reach} be as defined in Lem. C.5. Let $f_e\colon CPL^e \to L^e$ such that $f_e(cpl) = [\![\alpha]\!]_G(\sigma_G^e)$ where $\alpha \in cpl$. The following holds:

1. *f_e is a renaming function.*
2. *$\sigma_L^e \propto_{f_e} \sigma_G^e$.*
3. (a) *For any $o \in O_c^{args}$, for any $\alpha_q \in Bypass(O_c^{passed})\ o$, for any $\alpha_p \in bind_{args}\ o$, $[\![\hat{f}(\alpha_q)]\!]_G(\sigma_G^c) = [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma_G^e)$.*
   (b) *For any $o \in O_c^{cp}$, for any $\alpha_q \in Bypass(O_c^{passed})\ o$, for any $\alpha_p \in bind_{cp}\ o$, $[\![\hat{f}(\alpha_q)]\!]_G(\sigma_G^c) = [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma_G^e)$.*
4. *For any $\alpha \in GAccPath_q$, $[\alpha]_{A^c} \in O_c^{passed} \iff [\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \in L^{reach}$*
5. *For any $\alpha_p \in (\{h_1, \ldots, h_k\} \cup CPL^e) \times \{\epsilon\}$, $[\![\hat{f}_e(\alpha_p)]\!]_G(\sigma_G^e) = [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma_G^x)$.*

Proof: 1.
The function f_e is a total function from CPL^e to L^e. It is well defined because, by construction of CPL^e:

- every $cpl \in CPL^e$ contains at least one generalized access path (see Lem. C.8(1)), and
- for every $\alpha_1, \alpha_2 \in cpl$, $[\![\alpha_1 = \alpha_2]\!]_L(\sigma^e_L)$.

The function f_e is injective because, by construction of CPL^e, for every $cpl_1, cpl_2 \in CPL^e$ such that $cpl_1 \neq cpl_2$ for every $\alpha_1 \in cpl_1$ and $\alpha_2 \in cpl_2$ $[\![\alpha_1 \neq \alpha_2]\!]_L(\sigma^e_L)$.
2.

1. σ^e_L and σ^e_G are observationally equivalent: For any access paths $\langle h_i, \delta\rangle$ and $\langle h_j, \delta'\rangle$, where $1 \leq i, j \leq k$ and $\delta, \delta' \in \Delta$, $[\![\langle h_i, \delta\rangle = \langle h_j, \delta'\rangle]\!]_L(\sigma^e_L) \iff [\![\langle x_i, \delta\rangle = \langle x_j, \delta'\rangle]\!]_L(\sigma^c_L) \iff [\![\langle x_i, \delta\rangle = \langle x_j, \delta'\rangle]\!]_G(\sigma^c_G) \iff [\![\langle h_i, \delta\rangle = \langle h_j, \delta'\rangle]\!]_G(\sigma^e_G)$. The proof for the preservation of equality with `null` of access paths that start at a formal variable is analogous. All other variables $x \in V_p \setminus F_p$ are equal to `null` at function entry by definition.

2. σ^e_L and σ^e_G are context-aware equivalent w.r.t f_e. We prove this by case analysis.

   - Assume $\alpha = \langle h_i, \delta\rangle$ and $\beta = \langle h_j, \delta'\rangle$. Then $[\![\alpha = \beta]\!]_L(\sigma^e_L) \iff [\![\hat{f}_e(\alpha) = \hat{f}_e(\beta)]\!]_G(\sigma^e_G)$, because σ^e_L and σ^e_G are observationally equivalent; and, by Def. C.10, $\alpha = \hat{f}_e(\alpha)$ and $\beta = \hat{f}_e(\beta)$.
   - Assume $\alpha = \langle cpl, \delta_\alpha\rangle$ and $\beta = \langle h, \delta_\beta\rangle$ for some $cpl \in CPL^e$, $\delta_\alpha, \delta_\beta \in \Delta$ and $h \in F_p$

$[\![\alpha = \beta]\!]_L(\sigma^e_L)$	$\iff$	
$\forall \alpha' \in cpl\colon [\![\alpha'.\delta_\alpha = \beta]\!]_L(\sigma^e_L)$	$\iff$	Lem. C.9(1), transitivity of $[\![\cdot = \cdot]\!]_L$
$\forall \alpha' \in cpl\colon [\![\alpha'.\delta_\alpha = \beta]\!]_G(\sigma^e_L)$	$\iff$	σ^e_L and σ^e_G are observationally equivalent
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\alpha'.\delta_\alpha) = \hat{f}_e(\beta)]\!]_G(\sigma^e_G)$	$\iff$	Def. C.10
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\alpha'.\delta_\alpha)]\!]_G(\sigma^e_G) = [\![\hat{f}_e(\beta)]\!]_G(\sigma^e_L)$	$\iff$	Def. of equality
$\forall \alpha' \in cpl\colon [\![\langle \hat{f}_e(\alpha'), \delta_\alpha\rangle]\!]_G(\sigma^e_G) = [\![\hat{f}_e(\beta)]\!]_G(\sigma^e_L)$	$\iff$	Lem. C.12
$[\![\langle f_e(cpl), \delta_\alpha\rangle]\!]_G(\sigma^e_G) = [\![\hat{f}_e(\beta)]\!]_G(\sigma^e_L)$	$\iff$	$f_e(cpl) = [\![\alpha']\!]_G(\sigma^e_L)$, $\alpha' \in cpl,\ cpl \neq \emptyset$
$\forall \alpha' \in cpl\colon [\![\hat{f}_e(\langle cpl, \delta_\alpha\rangle) = \hat{f}_e(\beta)]\!]_G(\sigma^e_L)$		

   - The proof for the preservation of equality with `null` and equality between two cutpoint-anchored paths is analogous.

3.
Immediate from the definition of f_e; the substitution of actual parameters by formal parameters as defined in Fig. 5 and Fig. 11; and the fact that the $h^e = h^c$.

4.

43. $\forall \alpha \in GAccPath_q:$

$$
\begin{array}{lll}
[\alpha]_{\sigma_L^c} \in O_c^{passed} & \iff & \text{Definition of } O_c^{passed}, \\
[\alpha]_{A^c} \neq \emptyset \land & & \text{(see proof for} \\
\quad \exists i, 1 \leq i \leq k \colon \exists \delta \in \Delta \colon [\![\alpha = \langle x_i, \delta \rangle]\!]_L(\sigma_L^c) & \iff & \text{Lem. C.14(1))} \\
[\![\alpha \neq \mathtt{null}]\!]_L(\sigma_L^c) \land & & \text{Lem. C.7(1)} \\
\quad \exists i, 1 \leq i \leq k \colon \exists \delta \in \Delta \colon [\![\alpha = \langle x_i, \delta \rangle]\!]_L(\sigma_L^c) & &
\end{array}
$$

44. $\forall \alpha \in GAccPath_q:$

$$
\begin{array}{lll}
[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \in L^{reach} & \iff & \text{By definition of } L^{reach} \\
[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \in L^c \ \land & & \\
\quad \exists i, 1 \leq i \leq kv \colon \exists \delta \in \Delta \colon [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_G(\sigma_G^c) & \iff & \text{Def. of equality with } \mathtt{null} \\
[\![\hat{f}(\alpha) \neq \mathtt{null}]\!]_G(\sigma_G^c) \ \land & & \\
\quad \exists i, 1 \leq i \leq k . \exists \delta \in \Delta . [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_G(\sigma_G^c) & &
\end{array}
$$

45. $\forall \alpha \in GAccPath_q \colon \forall i, 1 \leq i \leq k \colon \forall \delta \in \Delta :$

$$
\begin{array}{lll}
[\![\alpha \neq \mathtt{null}]\!]_L(\sigma_L^c) \ \land \ [\![\alpha = \langle x_i, \delta \rangle]\!]_L(\sigma_L^c) & \iff & \sigma_L^c \propto_f \sigma_G^c \\
[\![\hat{f}(\alpha) \neq \mathtt{null}]\!]_G(\sigma_G^c) \ \land \ [\![\hat{f}(\alpha) = \langle x_i, \delta \rangle]\!]_G(\sigma_G^c) & &
\end{array}
$$

46. $\forall \alpha \in GAccPath_q:$

$$[\alpha]_{\sigma_L^c} \in O_c^{passed} \iff [\![\hat{f}(\alpha)]\!]_G(A^c) \in L^{reach} \qquad 43-45$$

5.

Immediate from the following facts:

1. formal parameters are not assigned;
2. $L^e \subseteq L^x$; and
3. by definition of $[\![\cdot]\!]_G$, for any $cpl \in CPL^e$,

$$[\![\hat{f}_e(\langle cpl, \epsilon \rangle)]\!]_G(\sigma_G^e) = f_e(cpl) = [\![\hat{f}_e(\langle cpl, \epsilon \rangle)]\!]_G(\sigma_G^x).$$

The following lemma shows that context-aware equivalence at the call-site is preserved at the corresponding return-site for access paths that do not traverse the local heap of the invoked function (1–2). Furthermore, it asserts that if the exit-states are context-aware equivalent w.r.t f_e (as defined in the previous lemma), then the return states are also context-aware equivalent w.r.t. f_e (3). This is the main lemma used in the proof of The. C.15.

Lemma C.14 (Context-aware equivalence at return sites) *Let σ_L^c, σ_G^c, f, $y = p(x_1, \ldots, x_k)$, p, σ_L^e, σ_L^x, σ_L^r, O_c^{args}, O_c^{passed}, O_c^{cp}, $bind_{args}$, $bind_{cp}$, $bind_{call}$, σ_G^e, σ_G^x, L^{reach}, and f_e be as in Lem. C.13. The following holds,*

1. *$\forall \alpha \in GAccPath_q$ if $[\alpha]_{A^r} \neq \emptyset \wedge [\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ then (i) $[\alpha]_{A^r} = [\alpha]_{A^c}$ and (ii) $[\![\hat{f}(\alpha)]\!]_G(\sigma_G^r) = [\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \notin L^{reach}$.*
2. *For any $o \in O_c^{passed}$, for any $\alpha_q \in Bypass(O_c^{passed})\ o$, $[\![\hat{f}(\alpha_q)]\!]_G(\sigma_G^c) = [\![\hat{f}(\alpha_q)]\!]_G(\sigma_G^r)$.*
3. *If $\sigma_L^x \propto_{f_e} \sigma_G^x$ then $\sigma_L^r \propto_f \sigma_G^r$.*

Proof:

1.

47.	$[\alpha]_{A^r} \in A^c \setminus (O_c^{passed} \cup \{\emptyset\})$	Assumption
48.	$[\alpha]_{A^c} = [\alpha]_{A^r}$	Admissibility of σ_L^c
49.	$[\alpha]_{A^c} \notin RObjs(O_c^{args})$	$47-48$, $O_c^{passed} = RObjs(O_c^{args})$
50.	$\forall o \in A^c, \forall o' \in O_c^{args}, \forall \delta \in \Delta, o'.\delta \subseteq o \implies \alpha \notin o$	49, def. of $RObjs(O_c^{args})$
51.	$\forall o \in A^c, \forall i, 1 \leq i \leq k, \forall \delta \in \Delta,$ $([x_i]_{A^c} \neq \emptyset \wedge [x_i]_{A^c}.\delta \subseteq o) \implies \alpha \notin o$	50, def. of O_c^{args}
52.	$[\alpha]_{A^c} \neq \emptyset$	$47-48$
53.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\alpha]_{A^c} \neq [\langle x_i, \delta\rangle]_{A^c}$	$51, 52$
54.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\alpha \neq \langle x_i, \delta\rangle]\!]_L(\sigma_L^c)$	53, def. of equality
55.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\hat{f}(\alpha) \neq \langle x_i, \delta\rangle]\!]_G(\sigma_G^c)$	54, $\sigma_L^c \propto_f \sigma_G^c$, $\hat{f}(\langle x_i, \delta\rangle) = \langle x_i, \delta\rangle$
56.	$\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \neq [\![\langle x_i, \delta\rangle]\!]_G(\sigma_G^c)$	55, def. of $[\![\cdot = \cdot]\!]_G$
57.	$[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) \notin L^{reach}$	56, def. of L^{reach}
58.	$[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) = [\![\hat{f}(\alpha)]\!]_G(\sigma_G^r)$	57, Lem. C.5

2.
Immediate from the definition of *Bypass*, Lem. C.14(1), Lem. C.5(2), and the fact that a callee cannot modify the value of pending variables.

3

Let $\alpha = \langle r_\alpha, \delta_\alpha\rangle$, $\beta = \langle r_\beta, \delta_\beta\rangle$, and $\gamma = \langle r_\gamma, \delta_\gamma\rangle$ be any generalized access paths of function p. We show that

1. $[\![\alpha = \beta]\!]_L(\sigma_L^r) \implies [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^r)$ and
2. $[\![\gamma = \texttt{null}]\!]_L(\sigma_L^r) \implies [\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_G^r)$.

The proof of the other direction, (i.e., that $[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^r) \implies [\![\alpha = \beta]\!]_L(\sigma_L^r)$ and $[\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_G^r) \implies [\![\gamma = \texttt{null}]\!]_L(\sigma_L^r)$) is analogous, and it is not shown.

The proof is done by case analysis.

1. Proving $[\![\alpha = \beta]\!]_L(\sigma_L^r) \Rightarrow [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^r)$
   a. Assuming $[\![\alpha \neq \texttt{null}]\!]_L(\sigma_L^r)$ and $[\![\beta \neq \texttt{null}]\!]_L(\sigma_L)$ and
      1. $[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \in A^c \setminus O_c^{passed}$.
      2. $[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$.
      3. $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \in A^c \setminus O_c^{passed}$.
      4. $[\alpha]_{A^r} \notin A^c \setminus O_c^{passed}$ and $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$.
   b. Assuming $[\![\alpha = \texttt{null}]\!]_L(\sigma_L^r)$ and $[\![\beta = \texttt{null}]\!]_L(\sigma_L^r)$.
2. Proving $[\![\gamma = \texttt{null}]\!]_L(\sigma_L^r) \Rightarrow [\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_G^r)$
   a. $r_\gamma = y$.
   b. $r_\gamma \neq y$ and
      1. $\forall \gamma' \leq \gamma, [\gamma']_{A^r} \in (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$.
      2. $\exists \gamma' \leq \gamma, [\gamma']_{A^r} \notin (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$.

Case 1(a)1:

59.	$[\alpha]_{A^r} \neq \emptyset$	$[\![\alpha \neq \texttt{null}]\!]_L(\sigma_L^r), Lem.\ C.7(1)$
60.	$[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$	Assumption
61.	$[\alpha]_{A^c} = [\alpha]_{A^r}$	$59 - 61$, Lem. C.14(1)
62.	$[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) = [\![\hat{f}(\alpha)]\!]_G(\sigma_G^r)$	$59 - 61$, Lem. C.14(1)
63.	$[\beta]_{A^c} = [\beta]_{A^r} \neq \emptyset$	Analogous to $59 - 61$
64.	$[\![\hat{f}(\beta)]\!]_G(\sigma_G^c) = [\![\hat{f}(\beta)]\!]_G(\sigma_G^r)$	Analogous to 62
65.	$[\alpha]_{A^r} = [\beta]_{A^r}$	$[\![\alpha = \beta]\!]_L(\sigma_L^r), Lem.\ C.7(2)$
66.	$[\alpha]_{A^c} = [\beta]_{A^c}$	61, 63, 65
67.	$[\![\alpha = \beta]\!]_L(\sigma_L^c)$	66, $Lem.\ C.7(2)$
68.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^c)$	67, $\sigma_L^c \propto_f \sigma_G^c$
69.	$[\![\hat{f}(\alpha)]\!]_G(\sigma_G^c) = [\![\hat{f}(\beta)]\!]_G(\sigma_G^c)$	By def. of equality
70.	$[\![\hat{f}(\alpha)]\!]_G(\sigma_G^r) = [\![\hat{f}(\beta)]\!]_G(\sigma_G^r)$	62, 64, 69
71.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^r)$	70, def. of equality

Case 1(a)2: This case is impossible.

72.	$[\alpha]_{A^r} \in A^c \setminus O_c^{passed}$, $[\alpha]_{A^r} \neq \emptyset$	See $Case\ 1(a)1$.
73.	$[\beta]_{A^r} \neq \emptyset$	$[\![\beta \neq \texttt{null}]\!]_L(\sigma_L^r)$, $Lem.\ C.7(1)$
74.	$[\beta]_{A^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$	73, $[\beta]_{A^r} \notin A^c \setminus O_c^{passed}$ $Lem.\ C.9(2)$
75.	$(A^c \setminus O_c^{passed}) \cap map(sub(bind_{ret}))\ A^x \subseteq \{\emptyset\}$	$Lem.\ C.8(2)$
76.	$[\alpha]_{\sigma_L^r} = [\beta]_{\sigma_L^r}$	$[\![\alpha = \beta]\!]_L(\sigma_L^r)$, $Lem.\ C.7(2)$
77.	Contradiction	$72, 73, 74 - 76$

Case 1(a)3: This case is also impossible (see proof for Case 1(a)2).

Case 1(a)4:

78.	$[\alpha]_{A^r} \neq \emptyset$	$[\![\alpha \neq \mathtt{null}]\!]_L(\sigma^r_L)$, $Lem.\ C.7(1)$
79.	$[\alpha]_{A^r} \in (map(sub(bind_{ret}))\ A^x) \setminus \{\emptyset\}$	78, Assumption, $Lem.\ C.9(2)$
80.	$[\beta]_{A^r} \neq \emptyset$	$[\![\beta \neq \mathtt{null}]\!]_L(\sigma^r_L)$, $Lem.\ C.7(1)$
81.	$[\beta]_{A^r} \in (map(sub(bind_{ret}))\ A^x) \setminus \{\emptyset\}$	80, Assumption, $Lem.\ C.9(2)$
82.	$[\alpha]_{A^r} = [\beta]_{A^r}$	$[\![\alpha = \beta]\!]_L(\sigma^r_L)$, $Lem.\ C.7(2)$
83.	$\exists! o \in A^x, [\alpha]_{A^r} = [\beta]_{A^r} = sub(bind_{ret})\ o \neq \emptyset$	$78 - 82$, Def. of map, $Lem.\ C.9(4)$
84.	Let o be the unique object in A^x which satisfies 83. Let $\alpha_0, r^\alpha_1, \delta^\alpha_1$ be the unique values determined by $Lem.\ C.9(6)$ for o and α. Let $\beta_0, r^\beta_1, \delta^\beta_1$ be the unique values determined by $Lem.\ C.9(6)$ for o and β.	
85.	$\alpha_0 \neq \langle y, \epsilon\rangle \implies$ $\alpha_0 \in Bypass(O^{passed}_c)\ [\alpha_0]_{\sigma^c_L}$, $[\alpha_0]_{A^c} \in O^{args}_c \cup O^{cp}_c$, $[\alpha_0]_{A^c} \in O^{args}_c \implies r^\alpha_1 = bind_{args}\ [\alpha_0]_{A^c} \neq \emptyset$ $[\alpha_0]_{A^c} \in O^{cp}_c \implies r^\alpha_1 = bind_{cp}\ [\alpha_0]_{A^c} \neq \emptyset$	$Lem.\ C.9(6)$
86.	$[\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) = \begin{cases} [\![\hat{f}(\langle y, \epsilon\rangle)]\!]_G(\sigma^r_G) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) & \alpha_0 \neq \langle y, \epsilon\rangle \end{cases}$	$Lem.\ C.9(6)$
87.	$[\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) = \begin{cases} [\![\langle y, \epsilon\rangle]\!]_G(\sigma^r_G) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}(\alpha_0)]\!]_G(\sigma^c_G) & \alpha_0 \neq \langle y, \epsilon\rangle \end{cases}$	Def. of $\hat{f}$ 85, $O^{args}_c \cup O^{cp}_c \subseteq O^{passed}_c$, $Lem.\ C.14(2)$
88.	$[\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) = \begin{cases} [\![\langle ret, \epsilon\rangle]\!]_G(\sigma^x_G) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma^e_G) & \alpha_0 \neq \langle y, \epsilon\rangle,\ \alpha_p \in r^\alpha_1 \end{cases}$	Def. of $\mathcal{GSB}$ see Fig. 5 85, $Lem.\ C.13(3)$
89.	$[\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) = \begin{cases} [\![\hat{f}_e(\langle ret, \epsilon\rangle)]\!]_G(\sigma^x_G) & \alpha_0 = \langle y, \epsilon\rangle \\ [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma^x_G) & \alpha_0 \neq \langle y, \epsilon\rangle,\ \alpha_p \in r^\alpha_1 \end{cases}$	Def. of $\hat{f}_e$ 88, $Lem.\ C.13(5)$
90.	$\forall \alpha_p \in r^\alpha_1, [\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G) = [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma^x_G)$	89, $\alpha_0 = \langle y, \epsilon\rangle \Rightarrow r^\alpha_1 = \{\langle ret, \epsilon\rangle\}$
91.	$[\![\hat{f}(\alpha)]\!]_G(\sigma^r_G) = [\![\langle [\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G), \delta^\alpha_1\rangle]\!]_G(\sigma^r_G)$	$\alpha = \alpha_0.\delta^\alpha_1$, $Lem.\ C.6(1)$, $Lem.\ C.12$
92.	$\forall \beta_p \in r^\beta_1, [\![\hat{f}(\beta_0)]\!]_G(\sigma^r_G) = [\![\hat{f}_e(\beta_p)]\!]_G(\sigma^x_G)$	Analogous to $85 - 90$
93.	$[\![\hat{f}(\beta)]\!]_G(\sigma^r_G) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(\sigma^r_G), \delta^\beta_1\rangle]\!]_G(\sigma^r_G)$	Analogous to 91
94.	$r^\alpha_1.\delta^\alpha_1 \subseteq o$, $r^\beta_1.\delta^\beta_1 \subseteq o$	84, $Lem.\ C.9(6)$
95.	$\forall \alpha_p \in r^\alpha_1, \forall \beta_p \in r^\beta_1, [\![\alpha_p.\delta^\alpha_1 = \beta_p.\delta^\beta_2]\!]_L(\sigma^x_G)$	$94, Lem.\ C.7(2)$, Admissibility of σ^x_L
96.	$\sigma^x_L \propto_{f_e} \sigma^x_G$	Assumption
97.	$\forall \alpha_p \in r^\alpha_1, \forall \beta_p \in r^\beta_1, [\![\hat{f}_e(\alpha_p.\delta^\alpha_1) = \hat{f}_e(\beta_p.\delta^\beta_2)]\!]_G(\sigma^x_G)$	$95 - 96$
98.	$\forall \alpha_p \in r^\alpha_1, \forall \beta_p \in r^\beta_1, [\![\hat{f}_e(\alpha_p.\delta^\alpha_1)]\!]_G(\sigma^x_G) = [\![\hat{f}_e(\beta_p.\delta^\beta_2)]\!]_G(\sigma^x_G)$	97, Def. of equality
99.	$\forall \alpha_p \in r^\alpha_1, \forall \beta_p \in r^\beta_1,$ $[\![\langle [\![\hat{f}_e(\alpha_p)]\!]_G(\sigma^x_G), \delta^\alpha_1\rangle]\!]_G(\sigma^x_G) = [\![\langle [\![\hat{f}_e(\beta_p)]\!]_G(\sigma^x_G), \delta^\beta_1\rangle]\!]_G(\sigma^x_G)$	98
100.	$[\![\langle [\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G), \delta^\alpha_1\rangle]\!]_G(\sigma^x_G) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(\sigma^r_G), \delta^\beta_1\rangle]\!]_G(\sigma^x_G)$	$90, 92, 99,$ $85\ (r^\alpha_1 \neq \emptyset,\ r^\beta_1 \neq \emptyset)$
101.	$[\![\langle [\![\hat{f}(\alpha_0)]\!]_G(\sigma^r_G), \delta^\alpha_1\rangle]\!]_G(\sigma^r_G) = [\![\langle [\![\hat{f}(\beta_0)]\!]_G(\sigma^r_G), \delta^\beta_1\rangle]\!]_G(\sigma^r_G)$	100, $h^r = h^x$, Def. of $[\![\cdot]\!]_G$
102.	$[\![\hat{f}(\alpha)]\!]_G(\sigma^r_G) = [\![\hat{f}(\beta)]\!]_G(\sigma^r_G)$	$91, 93, 101$
103.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma^r_G)$	102, Def. of equality

Case 1b:

104.	$[\![\alpha = \texttt{null}]\!]_L(\sigma_L^r)$	Assumption
105.	$[\![\beta = \texttt{null}]\!]_L(\sigma_L^r)$	Assumption
106.	$[\![\hat{f}(\alpha) = \texttt{null}]\!]_G(\sigma_G^r)$	104, $\sigma_L^c \propto_f \sigma_G^c$, $Case\ 2$
107.	$[\![\hat{f}(\beta) = \texttt{null}]\!]_G(\sigma_G^r)$	105, $\sigma_L^c \propto_f \sigma_G^c$, $Case\ 2$
108.	$[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G^r)$	106, 107

Case 2a:

109.	$[\![\langle y, \delta_\gamma\rangle = \texttt{null}]\!]_L(\sigma_L^r)$	By assumption
110.	$[\![\langle ret, \delta_\gamma\rangle = \texttt{null}]\!]_L(\sigma_L^x)$	$Lem.\ C.9(5a)$
111.	$\sigma_L^x \propto_{f_e} \sigma_G^x$	$\sigma_L^c \propto_{f_e} \sigma_G^c$, the induction assumption
112.	$[\![\hat{f}_e(\langle ret, \delta_\gamma\rangle) = \texttt{null}]\!]_G(\sigma_G^x)$	110, 111
113.	$[\![\langle ret, \delta_\gamma\rangle = \texttt{null}]\!]_G(\sigma_G^x)$	112, Def. of $\hat{f}_e$
114.	$[\![\langle y, \delta_\gamma\rangle = \texttt{null}]\!]_G(\sigma_G^r)$	$h^r = h^c, \rho^r(y) = \rho^x(ret)$

Case 2(b)1:

115.	$\forall \gamma' \leq \gamma, [\gamma']_{A^r} \in (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$	Assumption
116.	$A^r = (A^c \setminus O_c^{passed}) \cup map(sub(bind_{ret}))\ A^x$	Def. of $\mathcal{LSL}$ (see Fig. 11).
117.	$\forall \gamma' \leq \gamma, [\gamma']_{\sigma_L^c} \notin O_c^{passed}$	116, $Lem.\ C.9(3), Lem.\ C.8(1)$
118.	$[\![\gamma = \texttt{null}]\!]_L(\sigma_L^c)$	115, 117, $\gamma \leq \gamma, Lem.\ C.9(5b)$
119.	$[\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_L^c)$	118, $\sigma_L^c \propto_f \sigma_G^c$
	We continue with case analysis w.r.t. the value of $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(\sigma_G^c)$	
•	Assume $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(\sigma_G^c) = \textit{null}$	
120.	$[\![\gamma = \texttt{null}]\!]_L(\sigma_L^r)$	$\forall \gamma' \leq \gamma [\![\gamma' \neq \texttt{null}]\!]_L(\sigma_L^r)$
121.	$r_\gamma \in V_q \setminus \{y\}$	120, $Lem.\ C.7(6)$, Assumption $(r_\gamma \neq y)$
122.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_L(\sigma_L^c)$	$120 - 121$, $Lem.\ C.9(5a)$
123.	$[\![\hat{f}(\langle r_\gamma, \epsilon\rangle) = \texttt{null}]\!]_G(\sigma_G^c)$	122, $\sigma_L^c \propto_f \sigma_G^c$
124.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_G(\sigma_G^c)$	Def. of $\hat{f}$
125.	$[\![\langle r_\gamma, \epsilon\rangle = \texttt{null}]\!]_G(\sigma_G^r)$	Def. of $\mathcal{GSB}$
126.	$[\![\gamma = \texttt{null}]\!]_G(\sigma_G^r)$	125, $\langle r_\gamma, \epsilon\rangle \leq \gamma$, $Lem.\ C.6(2a)$
•	Assume $[\![\langle r_\gamma, \epsilon\rangle]\!]_G(\sigma_G^c) \neq \textit{null}$	
127.	$\exists! s \in \mathit{FieldId},\ \exists! \gamma'.s \leq \gamma, [\![\hat{f}(\gamma') \neq \texttt{null}]\!]_G(\sigma_L^c) \wedge$ $\forall \gamma'', \gamma'.s \leq \gamma'', [\![\hat{f}(\gamma'') = \texttt{null}]\!]_G(\sigma_L^c)$	119, $Lem.\ C.6(2a - 2b)$, $Lem.\ C.12$
128.	Let f and γ' be the unique values satsfying 127	
129.	$[\![\hat{f}(\gamma')]\!]_G(\sigma_L^c) \notin L^{reach}$	117, 128, $Lem.\ C.13(4)$
130.	$[\![\hat{f}(\gamma')]\!]_G(\sigma_L^c) \in L^c \setminus L^{reach}$	129, $[\![\hat{f}(\gamma') \neq \texttt{null}]\!]_G(\sigma_L^c)$
131.	$[\![\hat{f}(\gamma').s]\!]_G(\sigma_L^c) = [\![\hat{f}(\gamma').s]\!]_G(\sigma_L^r)$	130, $Lem.\ C.5$, $Lem.\ C.12$
132.	$[\![\hat{f}(\gamma').s]\!]_G(\sigma_L^r) = \textit{null}$	$127 - 128$, 131, $Lem.\ C.12$
133.	$[\![\hat{f}(\gamma').s = \texttt{null}]\!]_G(\sigma_L^r)$	132, Def. of equality w. `null`
134.	$[\![\hat{f}(\gamma) = \texttt{null}]\!]_G(\sigma_L^r)$	128, 133, $Lem.\ C.6(2a)$, $Lem.\ C.12$

Case 2(b)2:

135.	$[\gamma]_{\sigma_L^r} = \emptyset$	$Lem.\ C.7(1)$ $(\llbracket \gamma = \mathtt{null} \rrbracket_L(\sigma_L^r))$
136.	$\exists \gamma' \leq \gamma, [\gamma']_{A^r} \notin (A^c \setminus O_c^{passed}) \cup \{\emptyset\}$	Assumption
137.	$\exists \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$, $\forall \gamma'', \gamma' \leq \gamma'' \implies [\gamma'']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \cup \{\emptyset\}$	$135 - 136,\ Lem.\ C.9(2)$
138.	$\exists \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \setminus \{\emptyset\}$, $\forall \gamma'', \gamma' \leq \gamma'' \implies [\gamma'']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \cup \{\emptyset\}$	$137,\ \llbracket \gamma = \mathtt{null} \rrbracket_L(\sigma_L^r)$, $Lem.\ C.7(1)$
139.	$\exists s \in FID, \exists \gamma', \gamma'.s \leq \gamma \wedge [\gamma']_{\sigma_L^r} \in map(sub(bind_{ret}))\ A^x \wedge$ $\llbracket \gamma' \neq \mathtt{null} \rrbracket_L(\sigma_L^r) \wedge \forall \gamma'', \gamma'.s \leq \gamma'' \implies \llbracket \gamma'' = \mathtt{null} \rrbracket_L(\sigma_L^r)$	$138,\ Lem.\ C.7(1,3,4)$
140.	$\exists s \in FID, \exists \gamma', \gamma'.s \leq \gamma, \exists o \in A^x,\ [\gamma']_{\sigma_L^r} = sub(bind_{ret})\ o \wedge$ $\llbracket \gamma' \neq \mathtt{null} \rrbracket_L(\sigma_L^r) \wedge \llbracket \gamma'.s = \mathtt{null} \rrbracket_L(\sigma_L^r)$	$139,$ Def. of map
141.	Let γ', s, o be the unique values satisfying 140	
142.	let $\gamma_0', r_1^{\gamma'}, \delta_1^{\gamma'}$ be the unique values determined by $Lem.\ C.9(6)$ for o and γ'.	
143.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \gamma_p.\delta_1^{\gamma'} s = \mathtt{null} \rrbracket_L(\sigma_L^x)$	$Lem.\ C.9(5c)$
144.	$\sigma_L^x \propto_{f_e} \sigma_G^x$	Assumption
145.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'} s) = \mathtt{null} \rrbracket_G(\sigma_L^x)$	$143 - 144$
146.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'} s) \rrbracket_G(\sigma_L^x) = \mathit{null}$	145
147.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \llbracket \hat{f}_e(\gamma_p.\delta_1^{\gamma'}) \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \mathit{null}$	$146,\ Lem.\ C.12$
148.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \llbracket \llbracket \hat{f}_e(\gamma_p) \rrbracket_G(\sigma_L^x).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \mathit{null}$	$147,\ Lem.\ C.12$
149.	$\forall \gamma_p \in r_1^{\gamma'}, \llbracket \hat{f}(\gamma_0) \rrbracket_G(\sigma_G^r) = \llbracket \hat{f}_e(\gamma_p) \rrbracket_G(\sigma_G^x),\ r_1^{\gamma'} \neq \emptyset$	See proof for $Case\ 1(a)4$
150.	$\llbracket \llbracket \llbracket \hat{f}(\gamma_0) \rrbracket_G(\sigma_L^r).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^x).s \rrbracket_G(\sigma_L^x) = \mathit{null}$	$148 - 149$
151.	$\llbracket \llbracket \llbracket \hat{f}(\gamma_0) \rrbracket_G(\sigma_L^r).\delta_1^{\gamma'} \rrbracket_G(\sigma_L^r).s \rrbracket_G(\sigma_L^r) = \mathit{null}$	$150,\ h^x = h^r$ (See Fig. 11)
152.	$\llbracket \gamma'.s \rrbracket_G(\sigma_L^r) = \mathit{null}$	$151,\ Lem.\ C.12$
153.	$\llbracket \gamma'.s = \mathtt{null} \rrbracket_G(\sigma_L^r)$	$152,$ Def. of $\llbracket \cdot \rrbracket_G$
154.	$\llbracket \gamma = \mathtt{null} \rrbracket_G(\sigma_L^r)$	$\gamma'.s \leq \gamma,\ Lem.\ C.6(2a)$

Theorem C.15 (Context-aware Equivalence Preservation) *Let p be a function. Let $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ be context-aware equivalent states w.r.t. to a renaming function f, i.e., $\sigma_L \propto_f \sigma_G$. Let st be an arbitrary statement in p. The following holds:*

1. *For any state $\sigma_L' \in \Sigma_L^p$ such that $\langle st, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma_L'$, there exists a state $\sigma_G' \in \Sigma_G^p$ such that:*
   - *$\langle st, \sigma_G \rangle \overset{G}{\rightsquigarrow} \sigma_G'$, and*
   - *$\sigma_L' \propto_f \sigma_G'$.*
2. *For any state $\sigma_G' \in \Sigma_G^p$ such that $\langle st, \sigma_G \rangle \overset{G}{\rightsquigarrow} \sigma_G'$, there exists a state $\sigma_L' \in \Sigma_L^p$ such that:*
   - *$\langle st, \sigma_L \rangle \overset{L}{\rightsquigarrow} \sigma_L'$, and*
   - *$\sigma_L' \propto_f \sigma_G'$.*

Proof: Let q be a function. Let $\sigma_L = \langle \mathit{CPL}, A\rangle \in \Sigma^q_L$ and $\sigma_G = \langle L, \rho, h\rangle \in \Sigma^q_G$ context-aware equivalent states w.r.t. to a renaming function f, i.e., $\sigma_L \propto_f \sigma_G$.

We prove (i) and (ii) simultaneously using an induction on the shape of the derivation tree. The proof is done by case analysis of the statement in the transition which labels the root of the derivation tree.

Base case: The transition which labels the root of the derivation tree contains an atomic statement, i.e., the derivation tree is a leaf. Thus, we only need to show that the states that result by executing the same (atomic) statement in σ_L and σ_G are also context-aware equivalent w.r.t. to f. The proof is done by a case analysis.

x=null The axiom for this statement has no side-condition, thus this statement is guaranteed to terminate in any state. In particular, it is true that $\exists \sigma'_L \in \Sigma^q_L,$ s.t. $\langle \mathtt{x} = \mathtt{null}, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L$ and $\exists \sigma'_G \in \Sigma^q_G,$ s.t. $\langle \mathtt{x} = \mathtt{null}, \sigma_G\rangle \overset{G}{\rightsquigarrow} \sigma'_G$. By definition, $\sigma'_L = \langle \mathit{CPL}, \mathit{rem}(A, \{x\})\rangle = \langle \mathit{CPL}, \{(\mathit{map}(\lambda o.o \setminus \{x\}.\Delta)\ A) \setminus \{\emptyset\}\rangle = \langle \mathit{CPL}, \{a \setminus x.\Delta \mid a \in A\} \setminus \{\emptyset\}\rangle$ and $\sigma'_G = \langle L, \rho[x \mapsto \mathit{null}], h\rangle$.

Let $\alpha = \langle r_\alpha, \delta_\alpha\rangle$, $\beta = \langle r_\beta, \delta_\beta\rangle$, and $\gamma = \langle r_\gamma, \delta_\gamma\rangle$ be generalized access paths of function q.

$$
\begin{array}{ll}
[\![\alpha = \beta]\!]_L(\sigma'_L) & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\} \setminus \{\emptyset\}, \alpha \in a' \iff \beta \in a' & \iff \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\}, \alpha \in a' \iff \beta \in a' & \iff \\
\left\{\begin{array}{ll}
r_\alpha \neq x, r_\beta \neq x, \forall a \in A, \alpha \in a \iff \beta \in a & \text{or} \\
r_\alpha = x, r_\beta \neq x, \forall a \in A, \beta \notin a & \text{or} \\
r_\alpha \neq x, r_\beta = x, \forall a \in A, \alpha \notin a & \text{or} \\
r_\alpha = x, r_\beta = x &
\end{array}\right. & \iff \\
\left\{\begin{array}{ll}
r_\alpha \neq x, r_\beta \neq x, [\![\alpha = \beta]\!]_L(\sigma_L) & \text{or} \\
r_\alpha = x, r_\beta \neq x, [\![\beta = \mathtt{null}]\!]_L(\sigma_L) & \text{or} \\
r_\alpha \neq x, r_\beta = x, [\![\alpha = \mathtt{null}]\!]_L(\sigma_L) & \text{or} \\
r_\alpha = x, r_\beta = x &
\end{array}\right. & \iff \quad (\sigma_L \propto_f \sigma_G) \\
\left\{\begin{array}{ll}
r_\alpha \neq x, r_\beta \neq x, [\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma_G) & \text{or} \\
r_\alpha = x, r_\beta \neq x, [\![\hat{f}(\beta) = \mathtt{null}]\!]_G(\sigma_G) & \text{or} \\
r_\alpha \neq x, r_\beta = x, [\![\hat{f}(\alpha) = \mathtt{null}]\!]_G(\sigma_G) & \text{or} \\
r_\alpha = x, r_\beta = x &
\end{array}\right. & \iff \\
[\![\hat{f}(\alpha)]\!]_G(\sigma'_G) = [\![\hat{f}(\beta)]\!]_G(\sigma'_G) & \iff \\
[\![\hat{f}(\alpha) = \hat{f}(\beta)]\!]_G(\sigma'_G) &
\end{array}
$$

$$
\begin{array}{ll}
[\![\gamma = \mathtt{null}]\!]_L(\sigma'_L) & \Longleftrightarrow \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\} \setminus \{\emptyset\}, \gamma \notin a' & \Longleftrightarrow \\
\forall a' \in \{a \setminus \{x\}.\Delta \mid a \in A\}, \gamma \notin a' & \Longleftrightarrow \\
\begin{cases} r_\gamma \neq x, \forall a \in A, \gamma \notin a \quad \text{or} \\ r_\gamma = x \end{cases} & \Longleftrightarrow \\
\begin{cases} r_\gamma \neq x, [\![\gamma = \mathtt{null}]\!]_L(\sigma_L) \quad \text{or} \\ r_\gamma = x \end{cases} & \Longleftrightarrow \quad (\sigma_L \propto_f \sigma_G) \\
\begin{cases} r_\gamma \neq x, [\![\hat{f}(\gamma) = \mathtt{null}]\!]_G(\sigma_G) \quad \text{or} \\ r_\gamma = x \end{cases} & \Longleftrightarrow \\
[\![\hat{f}(\gamma)]\!]_G(\sigma'_G) = \mathit{null} & \Longleftrightarrow \\
[\![\hat{f}(\gamma) = \mathtt{null}]\!]_G(\sigma'_G) &
\end{array}
$$

x=y Analogous to [x=null].

x=y.f Analogous to [x=null].

x.f=null Analogous to [x=null].

x.f=y Analogous to [x=null].

x=alloc t Analogous to [x=null].

Induction step (intraprocedural): The transition labeling the root of the derivation tree contains a non-atomic intraprocedural control statement. Thus, the induced derivation tree is not a leaf. The proof is done by a case analysis.

seq Assume that $st = st_1; st_2$ and that $\langle st_1; st_2, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L$.

155.	$\langle st_1; st_2, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L$	Assumption
156.	$\exists \sigma''_L.\langle st_1, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma''_L \wedge \langle st_2, \sigma''_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L$	Def. of [seq] in $\mathcal{LSL}$
157.	$\exists \sigma''_L.\exists \sigma''_G.\langle st_1, \sigma_L\rangle \overset{L}{\rightsquigarrow} \sigma''_L \;\wedge\; \langle st_2, \sigma''_L\rangle \overset{L}{\rightsquigarrow} \sigma'_L \;\wedge\; \langle st_1, \sigma_G\rangle \overset{G}{\rightsquigarrow} \sigma''_G \;\wedge\; \sigma''_L \propto_f \sigma''_G$	$\sigma_L \propto_f \sigma_G$ Induction assumption for st_1, σ_L, and σ_G
158.	$\exists \sigma''_G.\exists \sigma'_G.\langle st_1, \sigma_G\rangle \overset{G}{\rightsquigarrow} \sigma''_G \;\wedge\; \langle st_2, \sigma''_G\rangle \overset{G}{\rightsquigarrow} \sigma'_G \wedge \sigma'_L \propto_f \sigma'_G$	$\sigma''_L \propto_f \sigma''_G$ Induction assumption for st_2, σ''_L, and σ''_G
159.	$\exists \sigma'_G.\langle st_1; st_2, \sigma_G\rangle \overset{G}{\rightsquigarrow} \sigma'_G \;\wedge\; \sigma'_L \propto_f \sigma'_G$	Def. of [seq] in $\mathcal{GSB}$

The proof in the other direction is analogous.

if-tt Analogous to [seq].

if-ff Analogous to [seq].

while Analogous to [seq].

Induction step (interprocedural):

The transition labeling the root of the derivation tree contains a a function call. Thus, the induced derivation tree is not a leaf. Without loss of generality, assume that the invocation is `y=p(x`$_1$`,... ,x`$_k$`)`. To simplify notations, we assume that $\sigma_L^c = \sigma_L$ and that $\sigma_G^c = \sigma_G$.

Assume that $\langle y = p(x_1, \ldots, x_k), \sigma_L^c\rangle \overset{L}{\rightsquigarrow} \sigma_L^r$.

160.	$\langle y = p(x_1, \ldots, x_k), \sigma_L^c\rangle \overset{L}{\rightsquigarrow} \sigma_L^r$	Assumption
161.	$\exists \sigma_L^e, \sigma_L^x \in \Sigma_L^p, \sigma_L^r \in \Sigma_L^q, s.t.\ \langle \textit{body of}\, p, \sigma_L^e\rangle \overset{L}{\rightsquigarrow} \sigma_L^x;$ and σ_L^e, σ_L^x and σ_L^r are as defined in Fig. 11	Def. of function call in $\mathcal{LSL}$
162.	Let σ_G^e be that state that arise at the entry to p when p is invoked at σ_G^c	Such σ_G^e exists because in $\mathcal{GSB}$ there are no side-conditions for function calls
163.	Let f_e be that renaming function defined as in Lem. C.14 for σ_L^e and σ_G^e	
164.	$\sigma_L^e \propto_{f_e} \sigma_G^e$	162, 163, Lem. C.13(2)
165.	$\exists \sigma_G^x \in \Sigma_G^q. \langle \textit{body of}\, p, \sigma_G^e\rangle \overset{G}{\rightsquigarrow} \sigma_G^x \ \wedge\ \sigma_L^x \propto_{f_e} \sigma_G^x$	161, 164, Induction assumption for σ_L^e, σ_G^e, and f_e
166.	$\exists \sigma_G^r \in \Sigma_G^q. \langle y = p(x_1, \ldots, x_k), \sigma_G^c\rangle \overset{G}{\rightsquigarrow} \sigma_G^r\ s.t.$ σ_G^r is as defined in Fig. 5 for σ_G^c and σ_G^x	165
167.	$\sigma_L^r \propto_f \sigma_G^r$	164, 165, Lem. C.14(3)

The proof in the other direction is analogous.