A Semantics for Procedure Local Heaps and its Abstractions

by

Noam Rinetzky\textsuperscript{1} Jörg Bauer\textsuperscript{2} Thomas Reps\textsuperscript{3}
Mooly Sagiv\textsuperscript{1} Reinhard Wilhelm\textsuperscript{2}

\textsuperscript{1} School of Comp. Sci.; Tel Aviv Univ.; Tel Aviv 69978; Israel.
\{	exttt{maon,msagiv}@post.tau.ac.il}
\textsuperscript{2} Informatik; Univ. des Saarlandes; Saarbrücken, Germany.
\{	exttt{joba,wilhelm}@cs.uni-sb.de}
\textsuperscript{3} Comp. Sci. Dept.; Univ. of Wisconsin; Madison, WI 53706; USA.
\texttt{reps@cs.wisc.edu}
A Semantics for Procedure Local Heaps and its Abstractions

Noam Rinetzky¹† Jörg Bauer²† Thomas Reps³§ Mooly Sagiv¹§ Reinhard Wilhelm²

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.

---

* A shorter version of this paper, [37], was presented at POPL05.
† 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 ................................................. 3  
  1.1 Store-based vs. Storeless Semantics .......................... 3  
  1.2 Main Results ............................................. 3  
  1.3 Outline .................................................. 4  

2 Preliminaries .............................................. 4
  2.1 Syntax of $E_\text{Algol}$ ..................................... 4  
  2.2 Running Example .......................................... 5  
  2.3 Global-Heap Store-Based Semantics .......................... 5  
  2.4 Observable Properties ..................................... 7  

3 Cutpoints and their Use ................................... 8

4 The Localized-Heap Storeless Semantics and its Properties 10
  4.1 The Localized-Heap Storeless Semantics .................. 10  
    4.1.1 Memory States ........................................ 10  
    4.1.2 Inference Rules ...................................... 12  
  4.2 Properties of the Semantics ................................ 17  
    4.2.1 Semantic Equivalence ................................. 17  
    4.2.2 Standard Properties ................................. 18  
    4.2.3 Modularity .......................................... 18  
  4.3 Assertion Language ...................................... 19  

5 Abstract Interpretation ..................................... 21
  5.1 The May-Alias Abstraction .................................. 21  
  5.2 Interprocedural Shape Analysis with Local-Heaps .......... 22  
    5.2.1 Representing $LSL$ Memory States by $3$-Valued Logical Structures .................. 23  
    5.2.2 Abstract Interpretation ............................. 27  
    5.2.3 Discussion .......................................... 28  

6 Related Work ............................................. 28
  6.1 Storeless Semantics ...................................... 28  
  6.2 Interprocedural Shape Analysis .......................... 28  
  6.3 Local Reasoning ......................................... 29  
  6.4 Encapsulation .......................................... 29  
  6.5 Rule of Adaptation ...................................... 30  

7 Conclusions .............................................. 30  

A Additional Code ........................................... 35  

B The May-Alias Abstraction .................................. 36

C Proofs ..................................................... 36  
  C.1 Properties of the $G_{SB}$ Semantics .................... 37  
  C.2 Properties of the $L_{SL}$ Semantics .................... 38  
  C.3 Context-Aware Equivalence .............................. 43
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 relate 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, $\text{LSL}$, 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 $\text{LSL}$, the non-standard concrete storeless semantics. The second part (Sec. 5) concerns abstract-interpretation of this semantics.

$\text{LSL}$ 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, 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. 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 LSL simulates a standard store-based semantics and identify a class of observations for which 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 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 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:

- 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 LSL.

- Using an abstraction of 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 LSL. Sec. 4 defines LSL semantics for EAlgol and states its properties. Sec. 5 presents the two aforementioned abstractions of 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 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 $\varepsilon$ denotes a sequence of $z$’s. We define the syntactic domains $x, y \in \text{VarId}$, $f \in \text{FieldId}$, $p \in \text{FuncId}$, $t \in \text{TypeId}$, and $lb \in \text{Labels}$ 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 ($\text{Sll}$); three list-manipulating functions: create ($\text{crt}$), destructive append ($\text{app}$), and destructive reverse ($\text{reverse}$); and a main function.

The program allocates three acyclic linked lists. It then destructively appends the list pointed-to by $t_2$ to the tails of the lists pointed-to by $t_1$ and $t_3$. As a result, at program point $lb_z$, just before $\text{reverse}$ 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 $\text{reverse}$, which is the core of our running example, (destructively) reverses the list passed as an argument. As a result, at $lb_y$, $\text{reverse}$’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 $\text{GSB}$ 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.\footnote{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., $x = x.z$. Such statements require additional source-to-source transformations and the introduction of temporary variables.} In addition, we assume that before a function terminates it assign a null value to every pointer variable that is not a formal parameter.\footnote{These conventions simplify the definition of both $\text{GSB}$ semantics and $\text{LSL}$; 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.}

Fig. 3 defines the semantic domains. $\text{Loc}$ is an unbounded set of memory locations. A memory state for a function $p$, $\sigma^p_G \in \Sigma^p_G$, keeps track of the allocated memory locations, $L$, an environment mapping $p$’s local variables to values, $\rho$, and a mapping from fields of allocated locations to values, $h$. Due to our simplifying assumptions, a value is either a memory location or null.

The meaning of statements is described by a transition relation $\trianglerighteq_G \subseteq (\sigma_G \times \text{stms}) \times \sigma_G$. Fig. 4 shows the axioms for assignments. The inference rule for function calls is given in Fig. 5. All other statements are
record Sll := { Sll n; int d }
Sll reverse(Sll h):=
  \begin{aligned}
  \text{lb} & = \text{lb} \\
  p & = h; \\
  \text{while} (p \neq \text{null}) \text{ do} q = p.n; p.n = t; t = p; p = q \text{ od}; \\
  \text{ret} & = t
  \end{aligned}
int main():=

Sll x,y,z,t1,t2,t3;
  t1 = \text{crt}(3); t2 = \text{crt}(2); t3 = \text{crt}(3); 
  x = \text{app}(t1,t2); 
  z = \text{app}(t3,t2); 
  t1 = \text{null}; t2 = \text{null}; t3 = \text{null}; 
\text{lb}: y = \text{reverse}(x); \text{lb}: 
  \text{ret} = 0

Figure 2: The running example. The code of functions \text{crt} and \text{app} appears in App. A.

\begin{align*}
l & \in \text{Loc} \\
v & \in \text{Val} = \text{Loc} \cup \{\text{null}\} \\
\rho & \in \text{Env}_p = \text{V}_p \rightarrow \text{Val} \\
h & \in \text{Heap}_G = \text{Loc} \times \text{FieldId} \rightarrow \text{Val} \\
\sigma_G, \langle L, \rho, h \rangle & \in \Sigma^L_G = 2^{\text{Loc}} \times \text{Env}_p \times \text{Heap}_G
\end{align*}

Figure 3: Semantic domains of the GSB semantics.

handled as usual using a two-level store semantics for pointer languages.

**Example 2.1** The memory state at lb\textsubscript{c}, the call-site to reverse, is depicted graphically in Fig. 6 (labeled \sigma\textsubscript{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 \sigma\textsubscript{e}G. The heap of \sigma\textsubscript{e}G is identical to the one of \sigma\textsubscript{c}G, but its environment only maps h, reverse’s formal parameter, to l\textsubscript{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\textsubscript{x}, is denoted by \sigma\textsubscript{x}G.

\begin{align*}
\langle x = \text{null}, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L, \rho[x \mapsto \text{null}], h \rangle \\
\langle x = y, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L, \rho[x \mapsto \rho(y)], h \rangle \\
\langle x = y.f, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L, \rho[x \mapsto h(\rho(y), f)], h \rangle \quad \rho(y) \neq \text{null} \\
\langle x.f = \text{null}, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L, \rho, h[(\rho(x), f) \mapsto \text{null}] \rangle \quad \rho(x) \neq \text{null} \\
\langle x.f = y, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L, \rho, h[(\rho(x), f) \mapsto \rho(y)] \rangle \quad \rho(x) \neq \text{null} \\
\langle x = \text{alloc} t, \langle L, \rho, h \rangle \rangle & \xrightarrow{G} \langle L \cup \{l\}, \rho[x \mapsto l], h \cup I(l) \rangle \quad l \notin L
\end{align*}

Figure 4: Axioms for atomic statements in the GSB semantics. \( I \) initializes all pointer fields at \( l \) to \( \text{null} \).
An Definition 2.3 (Access path)

The only means by which a program can observe a state. Note that the program cannot observe location names.

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 $\delta \in \Delta = \text{FieldId}^*$ is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by $\epsilon$.

Definition 2.3 (Access path) An access path $\alpha = (x, \delta) \in V_p \times \Delta$ of a function $p$ is a pair consisting of a local variable of $p$ and a field path. $\text{AccPath}_p$ denotes the set of all access paths of function $p$. $\text{AccPath}$ denotes the union of all access paths of all functions in a program.
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 \( \rho \), 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 \( (L, \rho, h) \), denoted by \([\alpha]_G\langle L, \rho, h \rangle\), is defined to be \( \hat{h}(\rho(x), \delta) \), where

\[
\hat{h} : \text{Val} \times \Delta \rightarrow \text{Val} \quad \text{such that}
\]

\[
\hat{h}(v, \delta) = \begin{cases} 
  v & \text{if } \delta = \epsilon \\
  \hat{h}(h(v, f), \delta') & \text{if } \delta = f\delta', \ v \in \text{Loc} \\
  \text{null} & \text{otherwise}
\end{cases}
\]

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 GSB semantics and \( \mathcal{LSL} \), our new semantics. Alternatively, we could have defined the value of such a path to be \( \bot \). 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 \( \alpha \) and \( \beta \) are equal in a given state \( \sigma_G \), 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 \([\alpha = \text{null}]_G(\sigma_G)\), if \([\alpha]_G(\sigma_G) = \text{null}\).

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. current access path). For example, in state \( \sigma^p_G \), at the entry to \text{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 \text{reverse}, we need to know that the list element pointed-to by \( h.n.n.n \) when the execution of \text{reverse} begins, is pointed-to by \( \text{retain} \) 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 pending access path that does not pass through any object that is reachable from one of \( p \)’s formal parameters.
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 \( (u_7 \text{ and } u_9) \) are heavily shaded. Note that \( u_{10} \) 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., \( x_2 \) and \( y.f1.f1 \). This is because \( u_{10} \) is also pointed-to by \( h \), zoo’s formal parameter.

For example, in memory state \( \sigma_c^G \), 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, 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 \( cpl \in 2^{F_p} \times \Delta \) 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, 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 \( \hat{a} \) where \( a \in CPLbs_p \) for cutpoint-labels for function \( p \).

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 \( \alpha = (cpl, \delta) \in CPLbs_p \times \Delta \) 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 ˆ{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 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 ⟨r, ϵ⟩ 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.

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 $\delta$ to $\alpha$. We say that a generalized access path $\alpha$ is a prefix of a generalized access path $\beta$, denoted by $\alpha \preceq \beta$, when there is a field path $\delta \in \Delta$, such that $\beta = \alpha.\delta$. We say that $\alpha$ is a proper prefix of $\beta$, 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 $\mathit{flat}$ functional, well-known from functional programming. $\mathit{flat} M$ returns the set of all elements of $M$, if $M$ is a set of sets. Formally, $\mathit{flat} M \overset{\text{def}}{=} \{ x \mid \exists A \in M : x \in A \}$.

**4.1.1 Memory States**

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.

A memory state for a function $p$ is a pair $(\text{CPL}_p, A_p)$ of a set of cutpoint-labels, (denoted by CPL) and a heap (denoted by $A_p$). A heap is a finite (but unbounded) set of objects. An object (denoted by $o$) is
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 \( \langle CPL_p \rangle \) and a representation of the heap \( \langle A_p \rangle \) 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 \( \langle CPL_p, A_p \rangle \) 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) \( A \) is prefix-closed, i.e., if \( \alpha, f \in \text{flat} A_p \), then \( \alpha \in \text{flat} A_p \); 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 \( \langle r, \delta \rangle \in \text{flat} A_p \), 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 \( LSL \) preserves admissibility of states (see Lem. 4.9), in the sequel, whenever we refer to an \( LSL \) state, we mean an admissible \( LSL \) 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 \( \langle x, e \rangle \) is in \( o \). Similarly, cutpoint-label \( cpl \) labels object \( o \) iff \( \langle cpl, e \rangle \) is in \( o \). The field \( \delta \) 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 \( \alpha \) 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 \( A' \)). It shows eight sets of generalized access paths. Each set represents one allocated list-element. At \( A' \), \( x.n.n.n \) 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 \( z \)'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 \( A'' \)) differs from \( A' \) in three ways: (i) there are only five objects in the heap; (ii) the set of cutpoint-labels contains \( \{h.n.n.n\} \), 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 \( \hat{h} \{h.n.n.n\} \).

4.1.2 Inference Rules

The meaning of statements is described by a transition relation \( L \Rightarrow \subseteq (\sigma_L \times \text{stmts}) \times \sigma_L \). 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 \( \sigma_L \) 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: \([\_], \text{rem}(\_, \_)\) and \(\text{add}(\_, \_)\), 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 \( \alpha \) points-to in heap \( A \). When \( \alpha \) 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 \( \text{rem}(A, a)\) 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 \( \text{rem} \) removes all the (generalized) access paths from the description of an object, that object is removed from the description of the heap. The function \( \text{add}(A, a, \alpha)\) yields a modified version of heap \( A \), where to every object \( o \in A \) reachable from \( \alpha \) by following some field path \( \delta \in \Delta \), the generalized access paths \( a.\delta \) are added.

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

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.

\[
\begin{align*}
: \ G\text{AccPath} \times \Delta & \rightarrow G\text{AccPath} \ \text{s.t.} \\
(r, \delta) \delta' & \equiv (r, \delta') \\
: \ 2^{G\text{AccPath}} \times \Delta & \rightarrow 2^{G\text{AccPath}} \ \text{s.t.} \\
\alpha.\delta & \equiv \{ \alpha.\delta \mid \alpha \in a \} \\
: \ 2^{G\text{AccPath}} \times 2^\Delta & \rightarrow 2^{G\text{AccPath}} \ \text{s.t.} \\
\alpha.D & \equiv \{ \alpha.\delta \mid \alpha \in a, \delta \in D \} \\
[] : \ G\text{AccPath} \times \text{Heap}_L & \rightarrow \text{Obj}_L \ \text{s.t.} \\
[\alpha]_A & \equiv \{ \beta \in a \mid a \in A, \alpha \in a \} \\
\text{rem} : \ \text{Heap}_L \times 2^{G\text{AccPath}} & \rightarrow \text{Heap}_L \ \text{s.t.} \\
\text{rem}(A, a) & \equiv (\text{map}(\lambda.o.o\setminus a.\Delta) \ A) \ \setminus \ \emptyset \\
\text{add} : \ \text{Heap}_L \times 2^{G\text{AccPath}} \times G\text{AccPath} & \rightarrow \text{Heap}_L \ \text{s.t.} \\
\text{add}(A, a, \alpha) & \equiv \text{map}(\lambda.o.o \cup a.\{ \delta \in \Delta \mid \alpha.\delta \in o \}) \ A
\end{align*}
\]

Figure 9: Helper functions.
Assigning \texttt{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 \texttt{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 \( \delta \), i.e., \((y, \delta)\) points-to \( o \). This is accomplished by applying \texttt{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 \( \delta \). 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.

A destructive update \( x.f = \texttt{null} \) (potentially) modifies the link structure of the heap. Thus, every \textit{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 \( \delta \) from \( y \), i.e., \((y, \delta)\in o\), is also reachable by traversing some (generalized) access path aliased with \( x \), followed by an \( f \)-field, and continuing with \( \delta \). As this is a place where cycles can be created, \texttt{add} does not necessarily return a right-regular heap. Therefore we apply the operator \( \cdot. \). \( A \) 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 \( A \).\footnote{The operator \( \cdot \) is similar to the \( \rho_{\text{rte}} \), operator in [14].}

The (deterministic) semantics of memory allocation \( x = \text{alloc} \ t \) 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 \texttt{null}.

### 4.1.2.3 Function Calls

The inference rule for function calls is defined in Fig. 11. The rule defines the program state \( \sigma_L^c \) that results from an invocation \( y=p(x_1, \ldots, x_k) \) at memory state \( \sigma_L^c \), assuming that the execution of the body of \( p \) at memory state \( \sigma_L^c \) results in memory state \( \sigma_L^q \). The heaps \( A^c \) and \( A^q \) are described by sets of generalized access paths starting at the caller’s variables and cutpoint-labels, whereas the heaps \( A^c' \) and \( A^q' \) 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 \( \text{Call}^q_y=p(x_1, \ldots, x_k) \) and \( \text{Ret}^q_y=p(x_1, \ldots, x_k) \), which are parameterized for each call statement in the program. \( \text{Call}^q_y=p(x_1, \ldots, x_k) \) computes the memory state \( \sigma_L^q \) that results at the entry of \( p \) when \( y=p(x_1, \ldots, x_k) \) is invoked by \( q \) in memory state \( \sigma_L^c \). The caller’s memory state after the invocation is restored by the function \( \text{Ret}^q_y=p(x_1, \ldots, x_k) \). This function computes the memory state of the caller at the return-site (\( \sigma_L^c \)) according to \( q \)'s memory state at the call-site (\( \sigma_L^q \)) and \( p \)'s memory state at the exit-site.
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 \( \mathcal{L} \mathcal{S} \mathcal{L} \) 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 \textit{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]. (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 \((x_1, \ldots, x_k)\), we first compute the set of objects that are \textit{pointed-to} by \( p \)’s actual parameters \((O_{\text{args}})\). Then, the auxiliary function \( \text{RObj} \) finds the part of the caller’s heap \((A^c)\) that is reachable from these objects \((O_{\text{passed}})\).

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 \( O_{\text{passed}} \). Thus, to compute the (possibly) updated description of objects in \( O_{\text{passed}} \) (as well as of objects that the callee allocates) it is sufficient to have a description of every object in \( O_{\text{passed}} \) (and of every object allocated by the callee) comprised of the (generalized) access paths that start at objects that separate \( O_{\text{passed}} \) from the rest of the caller’s heap: When the function returns, we just replace any (generalized) access paths \((r_p, \delta_p)\) in the description of every object in the heap of the callee \((A^c)\) that start at a “separating object” \( o' \), by access paths of the caller \((r_q, \delta_q)\) such that \((r_q, \delta_q)\) points-to \( o' \), but does not pass through \( O_{\text{passed}} \) (and thus cannot be modified). Technically, this is done as described below.

The auxiliary function \( \text{CPObj}_q \) (cf. Fig. 12) determines the cutpoints for this function invocation \((O_{\text{cp}})\). Cutpoints are the objects that “separate” \( O_{\text{passed}} \) 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 \( \text{CPObj}_q \), which is passed the caller’s memory state as well as the previously computed \( O_{\text{args}} \) and \( O_{\text{passed}} \), considers only objects in \( O_{\text{deep}} = O_{\text{passed}} \setminus O_{\text{args}} \) as possible cutpoints. Following the intuition of cutpoints as “separating objects”, an object \( o \in O_{\text{deep}} \) is qualified as a cutpoint if (and only if) one of the following holds:

\begin{itemize}
  \item \( o \) is pointed-to by a local variable of the caller \((O_{\text{vars}})\), or
  \item \( o \) is pointed-to by an object in the part of the caller’s heap that is not passed to the function \((O_{\text{fld}})\), or
  \item \( o \) separates the heap of the \textit{caller} from the heap of one of the pending calls, i.e., \( o \) is a cutpoint of the invocation of the caller \((O_{\text{cpi}})\).
\end{itemize}

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 \( \text{bind}_\text{args} \) 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 \( \text{bind}_\text{cp} \) 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., \( \text{bind}_\text{cp} \) maps a cutpoint to its label (see Sec. 3). To compute the label of a cutpoint \( o \), we apply \( \text{sub}(\text{bind}_\text{args}) \). The latter denotes a function that replaces every access path that starts with an actual parameter \((x_i, \delta)\) in the representation of \( o \) by an access path \((h_i, \delta)\) that starts with the corresponding formal parameter. (\text{sub} is defined in Fig. 12.) The \( \text{bind}_\text{call} \) combines the previous two mappings trivially as they have disjoint domains.

---

\(^4\)The same mechanism is used to compute the description of objects that the callee allocates.
\[
\begin{aligned}
\text{Call}_{q}^{y=p(x_1, \ldots, x_k)} &: \Sigma_L^q \rightarrow \Sigma_L^q \text{ s.t.} \\
\text{Call}_{q}^{y=p(x_1, \ldots, x_k)} (\langle \text{CPL}^c, A^c \rangle) &\overset{\text{def}}{=} \\
&\begin{cases}
O_c^{\text{args}} &= \{[x_i]_{A^c} \mid 1 \leq i < k, \ [x_i]_{A^c} \neq \emptyset\} \\
O_c^{\text{passed}} &= \text{RObjs}(A^c) \bigcup \text{O}_c^{\text{args}} \\
O_c^{\text{cpl}} &= \text{CPObjs}_q(\langle \text{CPL}^c, A^c \rangle) (\text{O}_c^{\text{args}}, O_c^{\text{passed}}) \\
\text{bind} &= \lambda o \in O_c^{\text{args}} \cup \{\langle h_i, c \rangle \mid 1 \leq i \leq k, x_i \in o\} \\
\text{bind}_c &= \lambda o \in O_c^{\text{cpl}} \cup \{\text{sub}(\text{bind}_c(o), \emptyset)\} \\
\text{bind}_c &= \lambda o \in O_c^{\text{cpl}} \cup O_c^{\text{cp}} \cup \{\text{bind}_c(o) \mid o \in O_c^{\text{args}}\} \\
\end{cases}
\end{aligned}
\]

\[
\begin{aligned}
\text{Ret}_{q}^{y=p(x_1, \ldots, x_k)} &: \Sigma_L^q \times \Sigma_L^q \rightarrow \Sigma_L^q \text{ s.t.} \\
\text{Ret}_{q}^{y=p(x_1, \ldots, x_k)} (\langle \text{CPL}^c, A^c \rangle, \langle \text{CPL}^x, A^x \rangle) &\overset{\text{def}}{=} \\
&\begin{cases}
\text{bind}_c &= \lambda o \in \text{range}(\text{bind}_c) \cup \{\langle \text{ret}, e \rangle\}. \\
&\begin{cases}
\{\langle y, e \rangle\} & a = \langle \text{ret}, e \rangle \\
\text{Bypass}(O_c^{\text{passed}}) \circ \text{bind}_c^{-1}(a) & \text{otherwise}
\end{cases}
\end{cases}
\end{aligned}
\]

\[
\begin{aligned}
\langle \text{body of } p, \sigma_L^c \rangle &\overset{L}{\rightarrow} \sigma_L^c \\
\langle y = p(x_1, \ldots, x_k), \sigma_L^c \rangle &\overset{L}{\rightarrow} \sigma_L^c
\end{aligned}
\]

where

\[
\begin{aligned}
\sigma_L^c &= \text{Call}_{q}^{y=p(x_1, \ldots, x_k)} (\sigma_L^c) \\
\sigma_L^c &= \text{Ret}_{q}^{y=p(x_1, \ldots, x_k)} (\sigma_L^c)
\end{aligned}
\]

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

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 (\( \text{CPL}^c \)) is computed by applying \( \text{bind}_c \) to every cutpoint. The heap component (\( A^c \)) is constructed by applying \( \text{bind}_c \) to every object in \( O_c^{\text{passed}} \). 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, \( \text{bind}_c \). This mapping is the inverse of \( \text{bind}_c \) (hence getting back to the caller’s representation of the object) composed with the function \( \text{Bypass}(O_c^{\text{passed}}) \), which filters out generalized access paths (of the caller) that pass through the heap that \( p \) had access to (\( O_c^{\text{passed}} \)). In addition, it also takes care of replacing access paths starting with special variable \( \text{ret} \) with the same access paths starting with result variable \( y \). Note that applying \( \text{bind}_c \) is well defined because \( \text{CPL}^c \) and \( \text{CPL}^x \) are equal (the callee cannot modify the set of objects that separate its own local-heap from the local-heap of some pending call\(^5\)).

The cutpoint-labels component of the state after the return of \( p \) is the same as before the invocation.

\(^5\)Note that in any transition \( \langle \sigma_L, sL \rangle \overset{L}{\rightarrow} \sigma'_L \), the cutpoint-labels component in \( \sigma_L \) and \( \sigma'_L \) is the same.
A body (A) of the heap at the call-site the passed objects (A the heap of some other (earlier) pending-call. The new heap is called CPL (function q 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′. 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′), and substituting the description of all the objects by applying sub(bind ret) to every object in A′.

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

\[
\begin{align*}
O_\text{deep} &= O_{\text{c passed}} \setminus O_{\text{c args}} \\
O_\text{vars} &= \{(x, e) \mid x \in V_q \} \\
O_\text{fld} &= \{o \in O_{\text{deep}} \mid \exists o' \in A' \setminus O_{\text{c passed}}, f \in \text{FieldId}, o'.f \subseteq o\} \\
O_\text{cpl} &= \{(cpl, e) \mid cpl \in CPL\} \\
\text{bind}_{\text{args}} &= \{x \mapsto h\} \\
\text{bind}_{\text{cpl}} &= \{z.n.n.n, x.n.n.n \mapsto \{h.n.n.n\}\} \\
\text{bind}_{\text{ret}} &= \{x \mapsto h, \{h.n.n.n\} \mapsto \{z.n.n.n\}, \{\text{ret} \} \mapsto \{y\}\}
\end{align*}
\]

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 \{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, \{h.n.n.n\}\}. When the execution of the function body ends, the cutpoint-anchored paths in the representation of every object in A′ (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 \{h.n.n.n\} n in the representation of the third element in the returned list is replaced by z.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 \( \alpha \) and \( \beta \) are equal in a given state \( \sigma_L \), denoted by \([\alpha = \beta]_L(\sigma_L)\), if \( \forall \alpha \in A. \alpha \alpha \beta \iff \alpha \beta \). An access path \( \alpha \) is equal to null in state \( \sigma_L \), denoted by \([\alpha = \text{null}]_L(\sigma_L)\), if \( \alpha \notin \text{flat} A \).

**Definition 4.5 (Observational equivalence)** Let \( p \) be a function. The states \( \sigma_L \in \Sigma^p_L \) and \( \sigma_G \in \Sigma^p_G \) are observationally equivalent if for all \( \alpha, \beta, \gamma \in \text{AccPath}_p \),

(i) \([\alpha = \beta]_L(\sigma_L) \iff [\alpha = \beta]_G(\sigma_G)\), and

(ii) \([\gamma = \text{null}]_L(\sigma_L) \iff [\gamma = \text{null}]_G(\sigma_G)\).

We also define observational equivalence between states in LSL in the same way.

4.2.1 Semantic Equivalence

The following theorem is the main theorem in the paper. It states that LSL is equivalent to 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 \( \sigma_L \in \Sigma^p_L \) and \( \sigma_G \in \Sigma^p_G \) be observationally equivalent states. Let \( \text{st} \) be an arbitrary statement in \( p \). The following holds:

\[
\langle \text{st}, \sigma_L \rangle \xrightarrow{\text{L}} \sigma_L' \iff \langle \text{st}, \sigma_G \rangle \xrightarrow{\text{G}} \sigma_G'.
\]

Furthermore, \( \sigma_L' \) and \( \sigma_G' \) are observationally equivalent.

We prove The. 4.6 by establishing a stronger property of the 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 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 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 , \( \text{lb} \) a program point in \( p \). For any \( \alpha, \beta \in \text{AccPath}_p \),

\([\alpha = \beta]_L \) is an invariant of \( P \) at \( \text{lb} \) iff \([\alpha = \beta]_G \) is an invariant of \( P \) at \( \text{lb} \).

The following theorem states that 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 \( x = \text{null} \) in a memory state \( \langle \text{CPL}, A \rangle \) leaks an object \( o \) iff \( o \subseteq x.\Delta \).
- Executing a statement \( x.\epsilon = \text{null} \) in a memory state \( \langle \text{CPL}, A \rangle \) leaks an object \( o \) iff \( o \subseteq [(x, \epsilon)]_A.\epsilon.\Delta \).

\(^6\)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 $\sigma_L$ an admissible state. If $\langle st, \sigma_L \rangle \xrightarrow{L} \sigma'_L$ then $\sigma'_L$ 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 $\sigma_L$ an admissible state. If $\langle st, \sigma_L \rangle \xrightarrow{L} \sigma'_L$ and $\langle st, \sigma_L \rangle \xrightarrow{L} \sigma''_L$, 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 $\langle \sigma_L, st \rangle \xrightarrow{L} \sigma'_L$.

**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 $\sigma_L$, 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 does not have an effect on the observable properties of the unreachable part of the heap.

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

- (i) $[\alpha = \beta]_L(\sigma^y_L) \iff [\alpha = \beta]_L(\sigma^x_L)$, and
- (ii) $[\gamma = \text{null}]_L(\sigma^y_L) \iff [\gamma = \text{null}]_L(\sigma^x_L)$.

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.

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

$\langle \sigma^1_L, st \rangle \xrightarrow{L} \sigma'^1_L \iff \langle \sigma^2_L, st \rangle \xrightarrow{L} \sigma'^2_L$.

Furthermore, $\sigma'^1_L$ and $\sigma'^2_L$ 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.

\(^7\)Recall that an access path $\alpha$ passes through an object $o$ if there exists a proper prefix $\alpha' < \alpha$ such that $\alpha'$ points-to $o$.  

18
Theorem 4.14 (Heap Modularity) Let $p$ be a function. For $i = 1, 2$, let $q_i$ be a function, $\sigma_L^{q_i} \in \Sigma_L^{q_i}$ be a state of function $q_i$, $y' = p(x_1, \ldots, x_k)$ be a statement in function $q_i$, and $\sigma_R^{q_i} = \text{Call}^{q_i} = p(x_1, \ldots, x_k)(\sigma_L^{q_i}) \in \Sigma_R^{q_i}$ be the state that results at the entry to function $p$ when it is invoked at $\sigma_L^{q_i}$. If $\sigma_L^{q_1}$ and $\sigma_L^{q_2}$ are observationally equivalent then the following properties hold:

(i) \( (\sigma_L^{q_1}, y^1 = p(x_1, \ldots, x_k)) \overset{L}{\rightarrow} \sigma_L^{q_1} \iff (\sigma_L^{q_2}, y^2 = p(x_1, \ldots, x_k)) \overset{L}{\rightarrow} \sigma_L^{q_2}, \) and

(ii) if $\text{CPL}^{q_i} \subseteq \text{CPL}^{q_j}$ and $(\sigma_L^{q_j}, \text{body of } p) \overset{L}{\rightarrow} \sigma_L^{q_j}$, then

\[ \sigma_L^{q_2} = \text{gc}(\text{Ret}^{q_2} = p(x_1, \ldots, x_k)(\sigma_L^{q_2}, \sigma_L^{q_2 \uparrow})) \]

where $\text{gc}(\langle \text{CPL}, A \rangle) \overset{\text{def}}{=} \langle \text{CPL}, A \setminus \emptyset \rangle$.

We need to apply gc to the heap produced by $\text{Ret}^{q_2} = p(x_1, \ldots, x_k)(\sigma_L^{q_2}, \sigma_L^{q_2 \uparrow})$ because of the following technical reason: it is possible that some of the objects in $A^{q_2}$ are reachable only from objects that are cutpoints when $p$ is invoked at $\sigma_L^{q_1}$ but not when it is invoked at $\sigma_L^{q_2}$. Thus, some objects that are reachable (i.e., pointed-to by a current or a pending access path) at $\sigma_L^{q_1}$ might not be reachable at $\sigma_L^{q_2}$.

4.3 Assertion Language

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

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

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

The intended meaning of the atomic assertions is:

- $\text{al}(e_1, e_2)$: the extended access paths $e_1$ and $e_2$ denote the same object (or $\text{null}$).
- $\text{isNull}(e)$: the extended access path $e$ denotes $\text{null}$.
- $e_1 \leq e_2$: 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 $I$: $\text{AccPathVar} \rightarrow \text{AccPath}_q$ associates an access-path variable with a particular access path. We lift an interpretation $I$ to $\hat{I}$: $\text{EAccPath}_q \rightarrow \text{AccPath}_q$ as follows:

\[ \hat{I}(e) = \begin{cases} I(e) & e \in \text{AccPathVar} \\ e & e \in \text{AccPath}_q \\ \hat{I}(e_1).f & e = e_1.f. \end{cases} \]

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 $\text{ass} \in \text{AtmAssrt}_q$ w.r.t. $I$, denoted by $\sigma_L \models_{\hat{I}} \text{ass}$ (resp. $\sigma_G \models_{\hat{G}} \text{ass}$). The satisfaction relation between states in $\text{LSL}$ (resp. $\text{GSB}$), 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 $\text{LSL}$ and $\text{GSB}$. 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 $\neg \text{isNull}(x, n)$ holds in any state in which the $n$-field of the object $x$ points-to does not have a $\text{null}$ value.
Figure 13: (1) The extended syntactic category $E\text{AccPath}_q$ of access paths for function $q$. (2) Syntax of assertions in $\text{Assn}_{\text{AP}}$ for function $q$. We also use the symbols $<, \lor, \Rightarrow$, and $\forall$ as shorthands in a standard manner. (3) The satisfaction relation between states, interpretations, and atomic assertions in the $\text{LSL}$ semantics. (4) The satisfaction relation in the $\text{GSB}$ semantics.

- The assertion $(x, e) \preceq \chi$ holds in any state, provided that the interpretation $I$ maps $\chi$ 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 $\text{ass} = (x, e) \preceq \chi \land \neg \text{isNull}(\chi)$ is satisfied if $\chi$ is mapped to an access path that points-to an object that is reachable from $x$. For example, $\text{ass}$ is satisfied in state $\sigma^*_L$, the state in which $\text{reverse}$ is invoked (see Fig. 6), w.r.t. an interpretation that maps $\chi$ to $(x, \text{nnnnnn})$ but not w.r.t. an interpretation that maps $\chi$ to either $(x, \text{nnnnnnnn})$ or to $(y, \text{nnnn})$.

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

**Lemma 4.16 (Equivalence w.r.t. $\text{Assn}_{\text{AP}}$)** Let $p$ be a function. Let $\sigma_L \in \Sigma^p_L$ and $\sigma_G \in \Sigma^p_G$ be observationally equivalent states. For any assertion $\text{ass} \in \text{Assrt}_p$ and for any interpretation $I \in \text{AccPathVar} \rightarrow \text{EAccPath}_p$, $\sigma_L \models^I_L \text{ass} \iff \sigma_G \models^I_G \text{ass}$.

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 $\text{lb}$ a program point in $p$. A closed assertion $\text{ass} \in \text{Assrt}_p$ is an invariant of $P$ at $\text{lb}$ according to the $\text{LSL}$ semantics iff $\text{ass}$ is an invariant of $P$ at $\text{lb}$ according to the $\text{GSB}$ semantics.

**Example 4.18** The following assertions are invariants of the running example at $\text{lb}_c$:

- $\neg \text{isNull}(x)$, expressing that $x$ points-to an object, and
- $\exists \chi, (x, e) < \chi \land \text{isNull}(\chi)$, expressing that $x$ points-to an acyclic list (e.g., null-terminated).
Having defined the satisfaction relation between interpretations, states in \( LSL \) (resp. \( GSB \)) and assertion in \( Assn_{AP} \), we define the notion of validity of partial (resp. total) correctness assertions in \( LSL \) (resp. \( GSB \)) in a standard manner (see [42, Ch. 6]). The following theorem states that \( LSL \) preserves validity of partial and total-correctness assertions expressed in \( Assn_{AP} \).

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

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

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

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

\[
ass_1 = \exists \chi_1, \ (x, e) \leq \chi_1 \land al(\chi_2, \chi_1) \land al(\chi_3, \chi_2, n) \land \\
\forall \chi_1 \leq \chi_1 : (\forall \chi_2 < \chi_2 : ¬al(\chi_1', \chi_2')) \land \\
(\forall \chi_3 < \chi_3 : ¬al(\chi_3', \chi_3'))
\]

\[
ass_2 = \exists \chi_4 : (y, e) \leq \chi_4 \land al(\chi_4, \chi_3) \land al(\chi_3, n, \chi_2)
\]

is valid in state \( p^\top \), 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 \( \chi_2 \) and \( \chi_3 \) 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, \( \chi_2 \) points-to the predecessor of the object that \( \chi_3 \) points-to in \( x \)'s list, and, in addition, that no prefix of either \( \chi_2 \) or \( \chi_3 \) points-to an object in \( x \)'s list. The postcondition ensures that after the call, \( \chi_2 \) points-to the successor of the object that \( \chi_3 \) points-to in the returned list.

5 Abstract Interpretation

In this section, we use the \( LSL \) 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 \( LSL \) states. The program is interpreted according to an abstract semantics \((\cdot)^a\) that over-approximates the concrete transition relation \((\cdot)^L\). Termination of 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 \( LSL \) 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 \( LSL \) 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—outpoints 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 \texttt{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 \texttt{revr} reverses a list recursively by invoking itself with the tail (\texttt{t}) of the (original) list, which is not reversed yet, and a pointer to the already reversed part (\texttt{r}). The analysis handles the destructive update precisely because it can distinguish between the value of \texttt{t} in the current call and its values in pending calls by abstracting them differently. However, in the analysis of the loop-based \texttt{reverse} function in our running example (where variable \texttt{p} plays the same role as \texttt{t} in Fig. 14), the analysis cannot distinguish between the value of \texttt{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 reference null; does not create garbage; and that when \texttt{reverse} returns, the variables \texttt{z} and \texttt{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 \texttt{p} by tabulating input/output descriptors. The tabulation is restricted to input descriptors that occur at the entry to \texttt{p}. The algorithm is sound by construction: it is an abstract interpretation [11] of \textsc{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 \( LSL \) 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 \( LSL \) 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 \( 0 \sqsubseteq \frac{1}{2} \sqsubseteq 1 \), 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 \( p^S : (U^S)^k \rightarrow \{0, \frac{1}{2}, 1\} \). 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 \( 2 \text{-Struct} \), and the set of 3-valued logical structures is denoted by \( 3 \text{-Struct} \).

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

\[
\alpha(AA) = \{ \beta_{\text{shape}}(\sigma_L) \mid \sigma_L \in AA \} \quad \text{and} \quad \gamma(SS) = \{ \sigma_L \in \Sigma_L \mid \beta_{\text{shape}}(\sigma_L) \in SS \}.
\]

#### 5.2.1.1 Representing a Local-Heap by a 2-Valued Logical Structure

The function \( \text{to}2\text{VLS} \), defined in Fig. 15, maps a local heap \( \sigma_L = (CPL, A) \) to a 2-valued logical structure \( S \). Every object \( a \in A \) and every cutpoint-label \( \text{cpl} \in CPL \) 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 \( \mathcal{P} \).
The predicates represents the cutpoint-label $\hat{S}$ to $\sigma_p$ that is labeled with binary predicate symbol $x$. The universe of $S_v$ to $2VLS$.

We explain the predicates' intended meanings through an example. In the example, we apply to2VLS to $\sigma_e^v$, the memory state at the entry point of $\tau_1 \tau_1 \tau_1 \tau_1$ (shown in Fig. 6). The resulting 2-valued logical structure, denoted by $S_e$, is depicted in Fig. 16.

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 $\{h.n.n.n\}$.

- 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 $u_{1\alpha}$ indicates that $u_{1\alpha}$ represents the $n$-successor of the list element represented by $u_0$; $u_{1\beta}$ represents the successor of $u_{1\alpha}$, 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 $\sigma_e^v$, 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 $\sigma_e^v$, no list element is locally shared. Thus, the value of $ils^S(u)$ is 0 for all of the nodes in $U^S_e$. Note that the predicate records only $local$ sharing. In particular, $ils^S(u_2) = 0$, although in a “global-view” of the heap, the list element represented by $u_2$ is the $n$-successor of two list elements:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$isList(v)$</td>
<td>Is $v$ a list element?</td>
</tr>
<tr>
<td>$isLabel(v)$</td>
<td>Is $v$ a cutpoint-label?</td>
</tr>
<tr>
<td>$x(v)$</td>
<td>Is $v$ pointed-to by a (current) variable $x$?</td>
</tr>
<tr>
<td>$n(v_1, v_2)$</td>
<td>Does the $n$-field of $v_1$ point-to $v_2$?</td>
</tr>
<tr>
<td>$r_x(v)$</td>
<td>Is $v_2$ reachable from (current) variable $x$ using $n$-fields?</td>
</tr>
<tr>
<td>$ils(v)$</td>
<td>Is $v$ locally shared? i.e., is $v$ pointed-to by more than one $n$-fields?</td>
</tr>
<tr>
<td>$c(v)$</td>
<td>Does $v$ reside on a directed cycle of $n$-fields?</td>
</tr>
<tr>
<td>$eq(v_1, v_2)$</td>
<td>Are $v_1$ and $v_2$ the same object or cutpoint-label?</td>
</tr>
<tr>
<td>$ilb(v_1, v_2)$</td>
<td>Is list element $v_2$ labeled by cutpoint-label $v_1$?</td>
</tr>
<tr>
<td>$cp(v)$</td>
<td>Is list element $v$ a cutpoint?</td>
</tr>
<tr>
<td>$r_{cp}(v)$</td>
<td>Is the list element $v$ reachable from a cutpoint using $n$-fields?</td>
</tr>
</tbody>
</table>

Table 1: The predicates used to represent states in $\Sigma_s$. There are separate predicates $x$ and $r_x$ for every program variable $x$.
one in the local heap (represented by u₁b) 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(σ₁,c) = 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_c record information that is special for the abstraction of an 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(u₆, u₂) = 1, because u₆ represents the label of the cutpoint represented by u₂.

  - The unary predicate cp records the property that a list element is a cutpoint, e.g., cp(u₈, u₂) = 1 because u₂ represents the only cutpoint in S_c; for all other nodes u, cp(u) = 0.

  - The unary predicate r_c records the property that a list element is reachable by a cutpoint-anchored path. For example, r_c(u₂) = 1 and r_c(u₃) = 1 because (only) u₂ and u₃ represent list elements that can be reached from the cutpoint (by the cutpoint-anchored paths ⟨{h.n.n.n}, c⟩ and ⟨{h.n.n.n}, n⟩, respectively). For all other nodes u, r_c(u) = 0.

The predicates cp and r_c 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 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 p_S(u₁, u₂) is set to a definite value (0 or 1) only when the predicate p_S(u₁, u₂) has this value for all the nodes u₁ and u₂ in U_S that are “merged” into u₁ and u₂, respectively.

Formally, a 3-valued logical structure S is a canonical abstraction of a 2-valued logical structure S if there exists a surjective function f: U_S → U_S such that (i) For all u₁, u₂ ∈ U_S, the value of the predicate p_S(u₁, u₂) is determined by the mapping f, and (ii) The resulting structure S satisfies the properties of a canonical abstraction.
Figure 17: Representative 3-valued logical structures that arise during the analysis of the running example at \( lb \), the call-site to \( reverse \) (first row, first column); \( lb \), the entry to \( reverse \) (first row, second column); \( lb \), \( reverse \)'s exit point (second row, second column); and \( lb \), the return-site from \( reverse \) (first row, second column).

\( U^S, f(u_1) = f(u_2) \) iff for all unary predicates \( p \in P \), \( p^S(u_1) = p^S(u_2) \), and (ii) For all predicates \( p \in P \) of arity \( k \) and for all \( k \)-tuples \( u_1^1, u_2^1, \ldots, u_k^1 \in U^{S^s} \),

\[
p^S(u_1^1, u_2^1, \ldots, u_k^1) = \bigcup_{u_1^{*}, \ldots, u_k^{*} \in U^S} p^S(u_1, u_2, \ldots, u_k).
\]

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

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 \( S^p_2 \), depicted in Fig. 17 (first row, second column), (conservatively) represents the memory state \( \sigma_L \), 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 \( \left( \frac{1}{2} \right) \) 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 \( S^p_2 \) contains five nodes. The mapping \( f: U^S_e \rightarrow U^{S^p}_e \) induced by the canonical abstraction is \( f(u_0) = u_0^p, f(u_{1a}) = f(u_{1b}) = u_1^p, f(u_2) = u_2^p, f(u_3) = u_3^p, \) and \( f(u_6) = u_6^p \). 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 \( U^{S^p} \).

The only summary node in \( S^p_2 \) is \( u_7^p \), which represents both \( u_{1a} \) and \( u_{1b} \). This is recorded by the predicate \( eq \), which has an indefinite value at \( u_7^p \), i.e., \( eq^{S^p}(u_7^p, u_7^p) =\frac{1}{2} \). The value of \( n^{S^p}(u_0^p, u_7^p) \) is indefinite because the \( n \)-field of the first list element (represented by \( u_0^p \)) points to the second list element (represented by \( u_1^p \)) but not to the third list element (also represented by \( u_1^p \)).

We see that any memory state represented by \( S^p_2 \) contains one cutpoint label (the node \( u_0^p \) is not a summary node). The cutpoint is represented by \( u_2^p \). This is recorded in two ways: (i) the value of the predicate \( lb^{S^p}(u_2^p, u_2^p) = 1 \) and (ii) \( u_2^p \) represents a list element that is labeled, as indicated by the value of the unary predicate \( cp^{S^p}(u_2^p) = 1 \).
\begin{equation}
\langle st, S \rangle \xrightarrow{L^1} \{ \beta_\text{shape}(\sigma_L) \mid \sigma_L \in \gamma(S) \}, \langle st, \sigma_L \rangle \xrightarrow{L^\perp} \sigma_L'
\end{equation}

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

<table>
<thead>
<tr>
<th>$\langle \text{body of } p, XS_p \rangle \xrightarrow{L^2} XS_p'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle y = p(x_1, \ldots, x_k), XS_q \rangle \xrightarrow{L^1} XS_q'$</td>
</tr>
</tbody>
</table>

where

\[
\{ \text{Call}_q^y(\sigma_L^L) \mid \sigma_L^L \in \gamma(XS_q) \} \subseteq \gamma(XS_p) \\
\{ \text{Ret}_q^y(\sigma_L^L, \sigma_L^L) \mid \sigma_L^L \in \gamma(XS_q), \sigma_L^L \in \gamma(XS_p' \text{ commpatible}(\sigma_L^L, \sigma_L^L)) \} \subseteq \gamma(XS_q')
\]

Figure 19: A specification of the abstract inference rules for function calls. The functions $\text{Call}_q^y=p(x_1, \ldots, x_k)$ and $\text{Ret}_q^y=p(x_1, \ldots, x_k)$ are defined in Fig. 11. Note that we apply $\text{Ret}_q^y=p(x_1, \ldots, x_k)$ only for compatible pairs of memory states. Memory states $\sigma_L^L$ and $\sigma_L^L$ are compatible when the sharing pattern that results from the invocation of $p$ at $\sigma_L^L$ matches the description of the context in $\sigma_L^L$, the state of $p$ at the exit-site. Formally, $\text{compatible}(\sigma_L^L, \sigma_L^L) \iff (\text{CPL}^c = \text{CPL}^x \land \forall h, h' \in F_p, [h = h']_L(\sigma_L^L) \iff [h = \text{null}]_L(\sigma_L^L))$, where $\sigma_L^L = \text{Call}_q^y=p(x_1, \ldots, x_k)(\sigma_L^L)$.

We also see that in any memory state represented by $S_L^L$ 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-valued logical structures. Fig. 18 and Fig. 19 show 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: $S_L^L$, which arises at $lh_e$, the call-site to reverse; $S_L^L$, which arises at $lh_e$, the entry to reverse; $S_L^L$, which arises at $lh_e$, the exit-point of reverse; and $S_L^L$, the structure computed at the return-site.

In $S_L^L$, the list pointed-to by ret is reversed. As a result, $u_0^L$ 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 $S_L^L$, the inference rule allows us to conclude that at $S_L^L$, the return-site’s logical structure, $u_0^L$ becomes reachable from z. Similarly, $u_0^L$ 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.
set of valid memory states $\rightarrow$ set of valid memory states

concretization \[3\]-valued structure \[3\]-valued structures

abstraction

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 $\sigma_L$ represented by $S$ (concretization); (ii) apply the statement’s concrete operational semantics to every such state $\sigma_L$, and (iii) abstract each of the resulting memory states by a $3$-valued structure (abstraction).

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 $LSL$, 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 $LSL$ 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, and reason about each part separately. Inferring the effect of a procedure on a heap described by $P * Q$ 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 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 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 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.

---

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 $Q$.
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 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 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, 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 LSL can be used to justify formally previous analyses that rely on this observation by showing that these analyses are an abstract interpretation of 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.

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


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.

```c
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.

```c
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

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
```

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 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 (SAPs). 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 LSL.

The actual representation of symbolic alias pairs (SAPs) is immaterial for the definition of the Galois connection. All we rely on is that the set UR = 2SAP of all symbolic alias relation forms a lattice ordered by ⊑ and equipped with a join operator ⊔. We make use of the function Factor : AccPath × AccPath → 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 → 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 βmap : ΣL → 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 (Vp) and formal parameters (Fp).

The function βmap is defined as a composition of two functions: (i) toPairsp : ΣL → 2APG×APG, which maps a program state of function p, σL ∈ ΣL, to pairs of (unbounded) access paths enriched with object names; and (ii) boundPairs : 2APG×APG → 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 parameter 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.\footnote{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.}

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 GSB semantics. In Sec. C.2, we state, and prove, additional properties of the LSL semantics. In Sec. C.3, we define the notion of context-aware equivalence between states in LSL and states in GSB, and prove a stronger theorem than the equivalence theorem, i.e., the preservation of context-aware equivalence.

\footnote{In [15], special care needs to be taken in case the analysis is parameterized by a lattice with infinite chains. In particular, ⊑SAP is not necessarily bounded. For simplicity, we assume this is not the case.}
with the same index (resp. prime). Similarly, we assume 
\( L, \rho, h \) with a certain index (resp. prime) to be the set of allocated locations, resp. environment, resp. heap of a state 
representation as sets of symbolic access path.

Figure 23: \( \beta^p \) is a representation function that maps a memory state of function \( p \) to its most precise representation as sets of symbolic access path.

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 \( \sigma_L \) with the same index (resp. prime). Similarly, we assume \( L, \rho, h \) with a certain index (resp. prime) to be the set of allocated locations, resp. environment, resp. heap of a state \( \sigma_G \) with the same index (resp. prime). In addition, we use the notations \([\alpha \neq \beta]_L(\sigma_L), [\alpha \neq \text{null}]_L(\sigma_L), \) \([\alpha \neq \beta]_G(\sigma_G), \) and \([\alpha \neq \text{null}]_G(\sigma_G)\) as shorthand for \( \neg[\alpha = \beta]_L(\sigma_L), \) resp. \( \neg[\alpha = \text{null}]_L(\sigma_L), \) resp. \( \neg[\alpha = \beta]_G(\sigma_G), \) resp. \( \neg[\alpha = \text{null}]_G(\sigma_G).\)

C.1 Properties of the GSB Semantics

In this section, we introduce the notions of heap paths and generalized heap paths. We also prove some properties of the GSB semantics which are used in the proof of Thm. 4.6.

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

Definition C.2 (Generalized heap path) A generalized heap path \( \zeta \in \text{AccPath}_p \cup \text{HeapPath} \) of function \( p \) is an access path of \( p \) or a heap path. GHeapPath 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 \( \zeta \) in memory state \( \langle L, \rho, h \rangle \) of function \( p \) is defined to be:

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

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 GSB semantics). The following definition generalizes Def. 2.5 (equality of access paths in the GSB semantics).
Definition C.4 (Generalized heap path equality) Generalized heap paths $\zeta_1$ and $\zeta_2$ are equal in a given state $\sigma_G$, 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 $\zeta$ is equal to null in a given state $\sigma_G$, denoted by $[\zeta = \text{null}]_G(\sigma_G)$, if $[\zeta]_G(\sigma_G) = \text{null}$.

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 $\sigma_G^c, \sigma_G^e$ be states in $\Sigma_G$ such that $(y = p(x_1, \ldots, x_k), \sigma_G^c) \xrightarrow{G} \sigma_G^e$. Let $L^{reach} \subseteq L^e$ be the location in $L^e$ that are reachable from the actual arguments at $\sigma_G^e$, i.e.,

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

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

1. $[\zeta]_G(\sigma_G^c) = [\zeta]_G(\sigma_G^e)$, and
2. for any $f \in \text{FieldId}$, $\langle [r, \delta f] \rangle_G(\sigma_G^c) = \langle [r, \delta f] \rangle_G(\sigma_G^e)$.

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 $\delta \in \Delta$, $\langle [l, \delta] \rangle_G(\sigma_G^c)$ is defined because $L^c \subseteq L^e$.

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 $\sigma_G \in \Sigma_G$ be a GSB state for function $q$.

1. For a generalized heap path $\alpha = \langle r,\delta_0\delta_1 \rangle \in G\text{HeapPath}_q$, it holds that $[[\alpha]]_G(\sigma_G) = [[\langle r,\delta_0 \rangle]_G(\sigma_G),\delta_1]]_G(\sigma_G)$.
2. For any (generalized) access path $\alpha \in G\text{HeapPath}_q$
   
   a. if $[[\alpha = \text{null}]]_G(\sigma_G)$, then for any generalized heap path $\alpha'$ such that $\alpha \leq \alpha'$, $[[\alpha']]_G(\sigma_G)$.
   
   b. if $[[\alpha \neq \text{null}]]_G(\sigma_G)$, then for any generalized heap path $\alpha'$ such that $\alpha \leq \alpha'$, $[[\alpha']]_G(\sigma_G)$.

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

C.2 Properties of the LSL Semantics

In this section, we prove certain properties of the $\text{LSL}$ semantics. These properties are needed in the proof of the Context-Aware Equivalence theorem, however, they are mere technicalities of the definition of $\text{LSL}$ 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 $[\cdot]$) Let $\sigma_L = (CPL, A) \in \Sigma_L$ be an (admissible) memory state for function $q$. For any generalized access paths $\alpha, \beta \in G\text{AccPath}_q$ the following holds:

1. $[\alpha = \text{null}]]_L(\sigma_L) \iff [\alpha]_A = 0$
2. \( [\alpha = \beta]_L(\sigma_L) \iff [\alpha]_A = [\beta]_A \)

3. \( [\alpha = \text{null}]_L(\sigma_L) \implies (\forall \alpha', \alpha \leq \alpha' \implies [\alpha' = \text{null}]_L(\sigma_L)) \)

4. \( [\alpha \neq \text{null}]_L(\sigma_L) \implies (\forall \alpha', \alpha' \leq \alpha \implies [\alpha' \neq \text{null}]_L(\sigma_L)) \)

5. \( [\alpha]_A \in A \cup \{0\} \)

6. \( \forall cpl \in CPL : [(cpl, \epsilon)]_A \neq \emptyset \)

**Sketch of Proof:** 1-5 are immediate from the definitions of \( [\cdot]_L \), \( [\cdot]_A \), 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 \( \sigma^c_L = \langle CPL^c, A^c \rangle \in \Sigma^c_L \) be an admissible memory state for function \( q \) in which the statement \( y = p(x_1, \ldots, x_k) \) is executed. Let \( \langle CPL^c, A^c \rangle, \langle CPL^r, A^r \rangle, \langle CPL', A' \rangle, O^c_{\text{args}}, O^c_{\text{passed}}, O^c_r, O^c_{\text{cre}}, \text{bind}_{\text{args}}, \text{bind}_{\text{call}}, \text{bind}_{\text{ret}}, \text{bind}_{\text{path}} \) be as defined in Fig. 11. Let \( \alpha \in \text{GAccPath}_q \) be an arbitrary generalized access path of function \( q \). The following holds.

1. \( \emptyset \notin O^c_{\text{passed}} \).

2. \( (A^c \setminus O^c_{\text{passed}}) \cap \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x = \emptyset. \)

3. If \( [\alpha]_{A^c} \neq \emptyset \) and \( [\alpha]_{A^c} \not\subset (A^c \setminus O^c_{\text{passed}}), \) then \( [\alpha]_{A^c} \in \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x. \)

4. \( \forall a, a' \in \text{dom}(\text{bind}_{\text{ret}}) : a \neq a' \implies \forall \alpha \in \text{bind}_{\text{ret}}(a) : \forall \beta \in \text{bind}_{\text{ret}}(a') : \alpha \not\leq \beta \land \beta \not\leq \alpha. \)

5. \( \forall a \in \text{dom}(\text{bind}_{\text{ret}}) : \forall \alpha, \beta \in \text{bind}_{\text{ret}}(a) : \alpha \neq \beta \implies \alpha \not\leq \beta \land \beta \not\leq \alpha. \)

6. \( \forall a \in \text{range}(\text{bind}_{\text{ret}}) : \forall \alpha \in a. \forall \alpha' < \alpha, [\alpha']_{A^c} \notin O^c_{\text{passed}}. \)

7. \( \emptyset \notin \text{range}(\text{bind}_{\text{ret}}) \)

**Proof:** We only sketch the proof of 3. The other properties are derived immediately from the definition of the \( \mathcal{L}S\mathcal{L} \) semantics.

1. \( [\alpha]_{A^c} \neq \emptyset \), \( [\alpha]_{A^c} \not\subset (A^c \setminus O^c_{\text{passed}}) \)

2. \( A^x = (A^c \setminus O^c_{\text{passed}}) \cup \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x \)

3. \( (A^c \setminus O^c_{\text{passed}}) \cap \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x = \emptyset \)

4. \( [\alpha]_{A^c} \in \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x \)

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 $\text{sub}(\text{bind}_{\text{ret}})$ 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 $\sigma_L^c, q, y = p(x_1, \ldots, x_k), \langle \text{CPL}^c, A^c \rangle, \langle \text{CPL}^x, A^x \rangle, \langle \text{CPL}^*, A^* \rangle, O^\text{args}, O^\text{pass} \rangle, O^c_{\text{cp}}, O^c_{\text{ret}}, \text{bind}_{\text{args}}, \text{bind}_{\text{cp}}, \text{bind}_{\text{call}}, \text{bind}_{\text{ret}},$ and $\alpha \in \text{GAccPath}_q$ be as in Lem. C.8. The following holds:

1. For any $cpl \in \text{CPL}^c$, the following holds:
   
   (a) $\emptyset \neq cpl \subseteq F_q \times \Delta$, and
   
   (b) for any $\alpha' \in cpl$ and any $\delta \in \Delta$, $[\alpha', \delta] = (cpl, \delta)]_L(\sigma_L^c)$.

2. If $[\alpha]_{A^c} \not\in (A^c \setminus O^\text{pass})$, then for any generalized access path $\alpha'$ such that $\alpha \leq \alpha'$, $[\alpha']_{A^c} \in \{\emptyset\} \cup \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^x$.

3. If $[\alpha]_{A^c} \in (A^c \setminus O^\text{pass})$, then for every generalized access path $\alpha' \leq \alpha$ it holds that $[\alpha']_{A^c} \not\in O^\text{pass}$.

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

5. (a) If $(y, e) \leq o$, then $[(y, \delta) = \text{null}]_L(\sigma_L^c) \iff [(\text{ret}, \delta) = \text{null}]_L(\sigma_L^c)$.

(b) If $(y, e) \not\leq o$ and $\forall \alpha' \leq o, [\alpha']_{A^c} \not\in O^\text{pass}$, then $[\alpha = \text{null}]_L(\sigma_L^c) \iff [\alpha = \text{null}]_L(\sigma_L^c)$.

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

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

   (a) $\alpha_0 = (y, e)$

   (b) $[\alpha_0]_{A^c} \in O^\text{args}$ and $\alpha_0 \in \text{Bypass}(O^\text{pass})$ $[\alpha_0]_{A^c}$

   (c) $[\alpha_0]_{A^c} \in O^\text{cp}$ and $\alpha_0 \in \text{Bypass}(O^\text{pass})$ $[\alpha_0]_{A^c}$

and, in addition, $r^*_1, \delta^*_1 \subseteq o$ where $\alpha = \alpha_0, \delta^*_1, r^*_1, \delta^*_1 \neq \emptyset$ and

\[
\begin{align*}
  r^*_1 & = \{ (\text{ret}, e) \} & & \text{(if case 6a holds)} \\
  & \cup \text{bind}_{\text{args}} [\alpha_0]_{A^c} & & \text{(if case 6b holds)} \\
  & \cup \text{bind}_{\text{cp}} [\alpha_0]_{A^c} & & \text{(if case 6c holds)}
\end{align*}
\]

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

1. (i) By definition, $\text{CPL}^c = \text{map}(\text{sub}(\text{args})) O^\text{cp}$. To show that $\emptyset \not\subseteq \text{CPL}^c \subseteq 2^{F_q \times \Delta}$, we show that $\forall o \in \text{O}^\text{pass} \emptyset \neq \text{sub}(\text{args}) o \subseteq F_q \times \Delta$. This proves (i), because $O^\text{cp} \subseteq \text{O}^\text{pass}$. Recall that

\[
\text{O}^\text{pass} = \text{RObj}(A^c) O^\text{args} = \{ o \in A^c \mid o' \in O^\text{args}, \delta \in \Delta, o'.\delta \leq o \}
\]

Thus,

\[
5. o \in \text{O}^\text{pass} \iff \exists o' \in O^\text{args}, \exists \delta \in \Delta. o'.\delta \leq o.
\]
By definition (see Fig. 11),

6. \( \emptyset \not\in O_{c_{\text{args}}} \), and
7. \( \text{bind}_{\text{args}} = \lambda o \in O_{c_{\text{args}}} \cdot \{ \langle h_i, e \rangle \mid 1 \leq i \leq k, x_i \in o \} \).

Thus, for any \( o \in O_{\text{passed}} \),

8. \( \text{sub}(\text{bind}_{\text{args}})(o) = \text{flat} \{ \text{bind}_{\text{args}}(a).\delta \mid a \in \text{dom}(\text{bind}_{\text{args}}), \delta \in \Delta, a.\delta \subseteq o \} \)
   \( = \text{flat} \{ \text{bind}_{\text{args}}(o').\delta \mid o' \in O_{\text{args}}, \delta \in \Delta, o'.\delta \subseteq o \} \)

which gives (by 5, 6) that \( \emptyset \not\in \text{sub}(\text{bind}_{\text{args}})(o) \), and (by 5, 7) that \( \text{sub}(\text{bind}_{\text{args}})(o) \subseteq F_q \times \Delta \).

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

9. \( CPL^c = \text{map}(\text{sub}(\text{bind}_{\text{args}})) O_{c_{\text{pr}}} \)
10. \( O_{c_{\text{pr}}} \subseteq O_{c_{\text{passed}}} \)
11. \( \text{bind}_{\text{cp}} = \lambda o \in O_{c_{\text{pr}}} : \{ \text{sub}(\text{bind}_{\text{args}}) o, e \} \)
12. \( \text{bind}_{\text{call}} = \lambda o \in O_{c_{\text{args}}} \cup O_{c_{\text{cp}}} : \{ \text{bind}_{\text{args}}(o) o \in O_{c_{\text{args}}} \}
    \quad \text{bind}_{\text{cp}}(o) o \in O_{c_{\text{cp}}} \}
13. \( A^c = \text{map}(\text{sub}(\text{bind}_{\text{call}})) O_{c_{\text{passed}}} \)

Thus, for any \( cpl \in CPL^c \) and for any \( o \in A^c \),

14. \( \langle cpl, \delta \rangle \in o \)
15. \( \exists o' \in O_{c_{\text{passed}}} : o = \text{sub}(\text{bind}_{\text{call}}) o' \land \langle cpl, \delta \rangle \in \text{sub}(\text{bind}_{\text{call}}) o' \)
16. \( \langle cpl, \delta \rangle \in \text{flat} \{ \text{bind}_{\text{call}}(a_1).\delta_1 \mid a_1 \in \text{dom}(\text{bind}_{\text{call}}), \delta_1 \in \Delta, a_1.\delta_1 \subseteq o' \} \)
17. \( \exists o' \in O_{c_{\text{passed}}} : o = \text{sub}(\text{bind}_{\text{call}}) o' \land \langle cpl, \delta \rangle = (\text{sub}(\text{bind}_{\text{args}}) o''., \delta) \land o''., \delta \subseteq o' \)
18. \( \exists o'' \in O_{c_{\text{pr}}} : \langle cpl, \delta \rangle = (\text{sub}(\text{bind}_{\text{args}}) o''., \delta) \land \forall o''' \in O_{c_{\text{args}}} : \forall o' \in \Delta : o''' . o' \Rightarrow o''' . o' \in o' \)
19. \( \forall o' \in O_{c_{\text{passed}}} : \exists o'' \in O_{c_{\text{cp}}} : \langle cpl, \delta \rangle = (\text{sub}(\text{bind}_{\text{args}}) o'', \delta) \land \text{sub}(\text{bind}_{\text{args}}) o''.\delta \subseteq \text{sub}(\text{bind}_{\text{args}}) o' \)
20. \( \forall o' \in cpl : o'.\delta \subseteq o \)

\( \iff 9, 7, 11, 12 \) \( \iff \text{Admissibility of } \sigma^c_L \)
\( O_{c_{\text{passed}}} = R\text{Objs}(A^c)O_{c_{\text{args}}} \) \( \iff 18, 8, \text{Def. of } \text{bind}_{\text{args}} \)
6. \[ [\alpha]_{\mathcal{A}'} \neq \emptyset, o \in \mathcal{A}', [\alpha]_{\mathcal{A}'} = \text{sub}(\text{bind}_{\mathcal{D}}) o \]

21. \[ \alpha \in \text{sub}(\text{bind}_{\mathcal{D}}) o \]

22. \[ \alpha \in \text{flat} \{ \text{bind}_{\mathcal{D}}(a).\delta^o_1 | a \in \text{dom}(\text{bind}_{\mathcal{D}}), \delta^o_1 \in \Delta, a.\delta^o_1 \subseteq o \} \]

23. \[ \exists a \in \text{dom}(\text{bind}_{\mathcal{D}}): \delta^o_1 \in \Delta: a.\delta^o_1 \subseteq o \land \alpha = \alpha_0.\delta^o_1 \]

24. \[ \exists a \in \text{dom}(\text{bind}_{\mathcal{D}}): \exists \alpha_0 \in \text{bind}_{\mathcal{D}}(a): a.\delta^o_1 \subseteq o \land \alpha = \alpha_0.\delta^o_1 \]

25. \[ \exists a \in \text{dom}(\text{bind}_{\mathcal{D}}): \exists \alpha_0 \in \text{bind}_{\mathcal{D}}(a): a.\delta^o_1 \subseteq o \land \alpha = \alpha_0.\delta^o_1 \]

26. \[ \exists a \in \text{dom}(\text{bind}_{\mathcal{D}}): \exists \alpha \in \text{dom}(\text{bind}_{\mathcal{D}}): a.\delta^o_1 \subseteq o \land \alpha = \alpha_0.\delta^o_1 \]

27. \[ \exists a \in \text{dom}(\text{bind}_{\mathcal{D}}): \exists \alpha \in \Delta: \exists \alpha_0 \in \text{bind}_{\mathcal{D}}(a): a.\delta^o_1 \subseteq o \land \alpha = \alpha_0.\delta^o_1 \]

28. Let \( a, \alpha_0, \delta^o_1 \) be the unique values satisfying 27. We continue with a case analysis of the possible values of \( a \in \text{dom}(\text{bind}_{\mathcal{D}}) \)

29. \[ \alpha_0 \in \text{flat} (\text{range}(\text{bind}_{\mathcal{D}})) \land \alpha_0 \leq \alpha: \alpha' \in \text{flat} (\text{range}(\text{bind}_{\mathcal{D}})) \implies \alpha' = \alpha_0 \]

30. \[ \text{dom}(\text{bind}_{\mathcal{D}}) = \{ ((\text{ret}, \epsilon)) \} \cup \text{map}(\text{bind}_{\mathcal{D}}) O^\epsilon_{\mathcal{D}} \cup \text{map}(\text{bind}_{\mathcal{D}}) O^\epsilon_{\mathcal{D}} \]

31. Assume \( a = ((\text{ret}, \epsilon)) \) (Case 6a)

32. \[ \text{bind}_{\mathcal{D}}(((\text{ret}, \epsilon))) = ((\text{y}, \epsilon)) \]

33. \[ \alpha_0 = (\text{ret}, \epsilon) \]

34. \[ (\text{ret}, \epsilon).\delta_2 \subseteq o \]

35. Assume \( a \in \text{map}(\text{bind}_{\mathcal{D}}) O^\epsilon_{\mathcal{D}} \) (Case 6b)

36. \[ \exists \alpha' \in O^\epsilon_{\mathcal{D}}, a = \text{bind}_{\mathcal{D}} \alpha' \]

37. \[ \alpha_0 \in \text{bind}_{\mathcal{D}}(a) \]

38. \[ \exists \alpha' \in O^\epsilon_{\mathcal{D}}, a = \text{bind}_{\mathcal{D}} \alpha', \alpha_0 \in \text{Bypass}(O^\epsilon_{\mathcal{D}}) \alpha' \]

39. \[ a = \text{bind}_{\mathcal{D}}([\alpha_0]_{\mathcal{A'}}), \alpha_0 \in \text{Bypass}(O^\epsilon_{\mathcal{D}}) [\alpha_0]_{\mathcal{A'}} \]

40. \[ \text{bind}_{\mathcal{D}} [\alpha_0]_{\mathcal{A'}}, \delta^o_1 \subseteq o \]

41. Assume \( a \in \text{map}(\text{bind}_{\mathcal{D}}) O^\epsilon_{\mathcal{D}} \) (Case 6c)

42. proof analogous to case 6b

Note that, by Lem. C.8(7), \( r^1_{\alpha} \neq \emptyset \).

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

**Sketch of Proof (The. 4.12):**

(i) For access paths that in memory state \( \sigma^c_{\mathcal{L}} \) point to an object which is not in \( O^\epsilon_{\mathcal{D}} \), the proof is immediate from admissibility of \( \sigma^c_{\mathcal{L}} \) and the fact that \( \mathcal{A'} \setminus O^\epsilon_{\mathcal{D}} \subseteq \mathcal{A'} \).

For accesses paths \( \alpha, \beta \) that in memory state \( \sigma^c_{\mathcal{L}} \) point to (the same) object in \( O^\epsilon_{\mathcal{D}} \), but do not pass through any object in \( O^\epsilon_{\mathcal{D}} \), the proof follows from Lem. C.9(5c).

(ii) For access paths that are equal null in \( \sigma^c_{\mathcal{L}} \), 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 \( \sigma^c_{\mathcal{L}} \) and \( \sigma^c_{\mathcal{L}} \):

- 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 \( \mathcal{LSE} \) semantics. To see why the induction step holds for a function call, we observe that in both \( \sigma^c_{\mathcal{L}} \) and \( \sigma^c_{\mathcal{L}} \):

- 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 cut-points.

C.3 Context-Aware Equivalence

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

Definition C.10 (Renaming function) Given an LSL state \(\langle \text{CPL}, A \rangle\) of function \(p\), and a GSB state \(\langle L, \rho, h \rangle\) of function \(p\), a function \(f: \text{CPL} \rightarrow L\) is a renaming function if it is total and injective. We lift \(f\) to \(f^+: \text{GAccPath}_p \rightarrow \text{GHeapPath}_p\), as follows:

\[
f^+(r) = \begin{cases} r & r \in \text{V}_p \\ f(r) & \text{otherwise} \end{cases}
\]

Definition C.11 (Context-Aware Equivalence) Let \(p\) be a function. The states \(\sigma_L = \langle \text{CPL}, A \rangle \in \Sigma^q_L\) and \(\sigma_G = \langle L, \rho, h \rangle \in \Sigma^q_G\) are context-aware equivalent w.r.t. a renaming function \(f\) if it is total and injective. The states \(\sigma_L\) and \(\sigma_G\) 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 \(\sigma_L \in \Sigma^q_L\) and \(\sigma_G \in \Sigma^q_G\) be context-aware equivalent states w.r.t a renaming function \(f\). For any \(\alpha, \alpha_0 \in \text{GAccPath}_p\) and any \(\delta \in \Delta\) such that \(\alpha = \alpha_0 \cdot \delta\), \([\hat{f}(\alpha)]_G(\sigma_G) = [\hat{f}(\alpha_0 \cdot \delta)]_G(\sigma_G) = (\Gamma^q(\alpha_0))_G(\sigma_G),\) and \([\hat{f}(\alpha)]_G(\sigma_G) = \Gamma(\sigma_G) = \delta\).

Proof: Immediate from the definition of \(\hat{f}\), the definition of \(-\), 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 \(f_c\). Property 1 shows that \(f_c\) is indeed a renaming function and Property 2 proves that the entry states are context aware equivalent with respect to \(f_c\). Properties 3–5 establish certain properties of \(f_c\).

Lemma C.13 (Context-aware equivalence of invoked functions)

Let \(\sigma_L^c = \langle \text{CPL}^c, A^c \rangle \in \Sigma^q_L\) and \(\sigma_G^c = \langle L^c, \rho^c, h^c \rangle \in \Sigma^q_G\) be context-aware equivalent states w.r.t. a renaming function \(f_c\) i.e., \(\sigma_L^c \propto_{f_c} \sigma_G^c\). Let \(y = y(x_1, \ldots, x_k)\) be a call to function \(p\) whose formal parameters are \(x_1, \ldots, x_k\). Let \(\sigma_L^c = \langle \text{CPL}^c, A^c \rangle, \sigma_L^c = \langle \text{CPL}^c, A^c \rangle, \sigma_G^c = \langle \text{CPL}^c, A^c \rangle, O^c_{\text{args}}, O^c_{\text{call}}, O^c_{\text{bind}}, \text{bind}_{\text{args}}, \text{bind}_{\text{cp}}\), and \(\text{bind}_{\text{call}}\) be as defined in Fig. 11. Let \(\sigma_G^c = \langle L_{c e}, \rho_{c e}, h_{c e} \rangle, \sigma_G^c = \langle L_{c x}, \rho_{c x}, h_{c x} \rangle\), and \(\sigma_G^c = \langle L_{c r}, \rho_{c r}, h_{c r} \rangle\) be as defined in Fig. 5. Let \(L^{\text{reach}}\) be as defined in Lem. C.5. Let \(f_c: \text{CPL}^c \rightarrow L^c\) such that \(f_c(cpl) = \alpha\) where \(\alpha \in \text{cpl}\). The following holds:

1. \(f_c\) is a renaming function.
2. \(\sigma_L^c \propto_{f_c} \sigma_G^c\).
3. (a) For any \(\alpha \in O^c_{\text{args}}, \text{for any } \alpha_q \in \text{Bypass}(O^c_{\text{args}})\), for any \(\alpha_p \in \text{bind}_{\text{args}}\), \(\Gamma(\alpha_q)_G(\sigma_G^c) = \Gamma(\alpha_q)_G(\sigma_G^c)\).
   
   (b) For any \(\alpha \in O^c_{\text{cp}}, \text{for any } \alpha_q \in \text{Bypass}(O^c_{\text{cp}})\), for any \(\alpha_p \in \text{bind}_{\text{cp}}\), \(\Gamma(\alpha_q)_G(\sigma_G^c) = \Gamma(\alpha_q)_G(\sigma_G^c)\).

43
4. For any $\alpha \in \text{GAccPath}_q$, $[\alpha]_{A^q} \in O^\text{passed} \iff [\hat{f}(\alpha)]_G(\sigma_G) \in L^{\text{reach}}$

5. For any $\alpha_p \in \{(h_1, \ldots, h_k) \cup \text{CPL}^e\} \times \{\epsilon\}$, $[\hat{f}_e(\alpha_p)]_G(\sigma_G^\epsilon) = [\hat{f}_e(\alpha_p)]_G(\sigma_G^\epsilon)$.

Proof: 1.

The function $f_e$ is a total function from $\text{CPL}^e$ to $L^e$. It is well defined because, by construction of $\text{CPL}^e$:

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

The function $f_e$ is injective because, by construction of $\text{CPL}^e$, for every $cpl_1, cpl_2 \in \text{CPL}^e$ such that $cpl_1 \neq cpl_2$ for every $\alpha_1 \in cpl_1$ and $\alpha_2 \in cpl_2$ $[\alpha_1 \neq \alpha_2]_L(\sigma_L^\epsilon)$.

2. $\sigma_L^\epsilon$ and $\sigma_G^\epsilon$ are observationally equivalent: For any access paths $(h_i, \delta)$ and $(h_j, \delta')$, where $1 \leq i, j \leq k$ and $\delta, \delta' \in \Delta$, $[[\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon).$ 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. $\sigma_L^\epsilon$ and $\sigma_G^\epsilon$ are context-aware equivalent w.r.t $f_e$. We prove this by case analysis.

- Assume $\alpha = (h, \delta)$ and $\beta = (h_j, \delta')$. Then $[[\alpha = \beta]_L(\sigma_L^\epsilon) \iff [\hat{f}_e(\alpha)]_G(\sigma_G^\epsilon)$. because $\sigma_L^\epsilon$ and $\sigma_G^\epsilon$ are observationally equivalent; and, by Def. C.10, $\alpha = \hat{f}_e(\alpha)$ and $\beta = \hat{f}_e(\beta)$.

- Assume $\alpha = (cpl, \delta_\alpha)$ and $\beta = (h, \delta_\beta)$ for some $cpl \in \text{CPL}^e$, $\delta_\alpha, \delta_\beta \in \Delta$ and $h \in F_p$

\[ [\alpha = \beta]_L(\sigma_L^\epsilon) \iff \forall \alpha' \in cpl: [\alpha' \delta_\alpha = \beta]_L(\sigma_L^\epsilon) \iff \text{Lem. C.9(1), transitivity of $[\cdot]_L$.}$

\[ [\alpha = \beta]_L(\sigma_L^\epsilon) \iff \forall \alpha' \in cpl: [\alpha' \delta_\alpha = \beta]_G(\sigma_G^\epsilon) \iff \text{Def. C.10.}$

\[ [\alpha = \beta]_L(\sigma_L^\epsilon) \iff \forall \alpha' \in cpl: [[\hat{f}_e(\alpha') \delta_\alpha = \hat{f}_e(\beta)]_G(\sigma_G^\epsilon) \iff \text{Def. of equality}$

\[ [\alpha = \beta]_L(\sigma_L^\epsilon) \iff \forall \alpha' \in cpl: [[[\hat{f}_e(\alpha') \delta_\alpha = \hat{f}_e(\beta)]_G(\sigma_G^\epsilon) \iff \text{Lem. C.12}$

\[ [\alpha = \beta]_L(\sigma_L^\epsilon) \iff \forall \alpha' \in cpl: [\hat{f}_e((cpl, \delta_\alpha)) = \hat{f}_e(\beta)]_G(\sigma_G^\epsilon) \iff \text{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 $h^e = h^e$. 44
4. Immediate from the following facts:

43. \( \forall \alpha \in G\text{AccPath}_q \):

\[ [\alpha]_{G^L} \in O_{\text{passed}} \]

\[ [\alpha]_{G^L} \neq \emptyset \land \]

\[ \exists i, 1 \leq i \leq k : \exists \delta \in \Delta : \langle \alpha = \langle x_i, \delta \rangle \rangle_{L}(\sigma^L) \]

\[ \exists i, 1 \leq i \leq k : \exists \delta \in \Delta : \langle \alpha = \langle x_i, \delta \rangle \rangle_{L}(\sigma^L) \]

\[ \iff \]

Definition of \( O_{\text{passed}} \),

(see proof for Lem. C.14(1))

44. Lem. C.7(1)

45. Lem. C.14(1)

46. Lemma C.14 (Context-aware equivalence at return sites)

Let \( \sigma^L, \sigma^G, f, y = p(x_1, \ldots , x_k), p, \sigma^L, \sigma^G, O^{\text{passed}}, O^{\text{cp}} = \text{bindargs, bindcp, bindcall}, \sigma^G, \sigma^L, L^{\text{reach}}, \) and \( f_e \) be as in Lem. C.13. The following holds:

1. \( \forall \alpha \in G\text{AccPath}_q \) if \( [\alpha]_{A^L} \neq \emptyset \land [\alpha]_{A^V} \in A^C \setminus O_{\text{passed}} \) then (i) \( [\alpha]_{A^V} = [\alpha]_{A^V} \) and (ii) \( \hat{f}(\alpha)\rangle_{G}(\sigma^G) = [\hat{f}(\alpha)]_{G}(\sigma^G) \). 

2. For any \( o \in O_{\text{passed}} \), for any \( \alpha_q \in \text{Bypass}(O_{\text{passed}}) \) \( \alpha, \hat{f}(\alpha_q)\rangle_{G}(\sigma^G) = [\hat{f}(\alpha_q)]_{G}(\sigma^G) \).

3. If \( \sigma^L \propto f, \sigma^G \) then \( \sigma^L \propto f, \sigma^G \).

5. Immediate from the following facts:

1. formal parameters are not assigned;

2. \( L^e \subseteq L^r \); and

3. by definition of \([ ]_{G}\), for any \( cpl \in CPL^e \),

\[\hat{f}_e((cpl, \epsilon))\rangle_{G}(\sigma^G) = f_e(cpl) = [\hat{f}_e(cpl, \epsilon)]_{G}(\sigma^G)\].

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.
Proof:

1. 
   47. \([\alpha]_{A^c} \in A^c \setminus (O^\text{passed} \cup \{\emptyset\})\) 
   48. \([\alpha]_{A^c} = [\alpha]_{A^c}\) 
   49. \([\alpha]_{A^c} \not\in \text{RObj}_c(O^\text{args})\) 
   50. \(\forall o \in A^c, \forall o' \in O^\text{args}, \forall \delta \in \Delta, o', \delta \subseteq o \Rightarrow \alpha \not\in o\) 
   51. \(\forall o \in A^c, \forall i \leq k, \forall \delta \in \Delta, \forall i', \delta \subseteq o \Rightarrow \alpha \not\in o\) 
   52. \([\alpha]_{A^c} \not\in \emptyset\) 
   53. \(\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, \forall \delta \in \Delta, [\alpha]_{A^c} \not\in \{x_i, \delta\}_{A^c}\) 
   54. \(\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, \forall \delta \in \Delta, \forall \delta \not\in \{x_i, \delta\}_{A^c}\) 
   55. \(\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\hat{f}(\alpha) \not\in \{x_i, \delta\}]_{\text{reach}}\) 
   56. \(\forall i, 1 \leq i \leq k, \forall \delta \in \Delta, [\hat{f}(\alpha)]_{\text{reach}}(\sigma^*_G) \not\in \{x_i, \delta\}\) 
   57. \([\hat{f}(\alpha)]_{\text{reach}}(\sigma^*_G) \not\in \text{reach}\) 
   58. \([\hat{f}(\alpha)]_{\text{reach}}(\sigma^*_G) = [\hat{f}(\alpha)]_{\text{reach}}(\sigma^*_G)\) 

2. Immediate from the definition of \textit{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 
   \[ [\alpha = \beta]_{L}(\sigma^*_L) \Rightarrow [\hat{f}(\alpha) = \hat{f}(\beta)]_{G}(\sigma^*_G) \] 
   and 
   \[ [\gamma = \text{null1}]_{L}(\sigma^*_L) \Rightarrow [\hat{f}(\gamma) = \text{null1}]_{G}(\sigma^*_G). \]

The proof of the other direction, (i.e., \([\hat{f}(\alpha) = \hat{f}(\beta)]_{G}(\sigma^*_G) \Rightarrow [\alpha = \beta]_{L}(\sigma^*_L)\) and \([\hat{f}(\gamma) = \text{null1}]_{G}(\sigma^*_G) \Rightarrow [\gamma = \text{null1}]_{L}(\sigma^*_L)\) is analogous, and it is not shown.

The proof is done by case analysis.

1. Proving \([\alpha = \beta]_{L}(\sigma^*_L) \Rightarrow [\hat{f}(\alpha) = \hat{f}(\beta)]_{G}(\sigma^*_G)\) 
   a. Assuming \([\alpha \neq \text{null1}]_{L}(\sigma^*_L)\) and \([\beta \neq \text{null1}]_{L}(\sigma_L)\) and
      1. \([\alpha]_{A^c} \in A^c \setminus O^\text{passed} \text{ and } [\beta]_{A^c} \in A^c \setminus O^\text{passed}.\]
      2. \([\alpha]_{A^c} \in A^c \setminus O^\text{passed} \text{ and } [\beta]_{A^c} \not\in A^c \setminus O^\text{passed}.\]
      3. \([\alpha]_{A^c} \not\in A^c \setminus O^\text{passed} \text{ and } [\beta]_{A^c} \in A^c \setminus O^\text{passed}.\]
      4. \([\alpha]_{A^c} \not\in A^c \setminus O^\text{passed} \text{ and } [\beta]_{A^c} \not\in A^c \setminus O^\text{passed}.\]
   b. Assuming \([\alpha = \text{null1}]_{L}(\sigma^*_L)\) and \([\beta = \text{null1}]_{L}(\sigma^*_L)\).

2. Proving \([\gamma = \text{null1}]_{L}(\sigma^*_L) \Rightarrow [\hat{f}(\gamma) = \text{null1}]_{G}(\sigma^*_G)\) 
   a. \(r_\gamma = y.\)
   b. \(r_\gamma \neq y\) and
      1. \(\forall \gamma' \leq \gamma, [\gamma']_{A^c} \in (A^c \setminus O^\text{passed}) \cup \{\emptyset\}.\)
      2. \(\exists \gamma' \leq \gamma, [\gamma']_{A^c} \not\in (A^c \setminus O^\text{passed}) \cup \{\emptyset\}.\)
Case 1(a)1:

- $[\alpha]_{A^c} \neq \emptyset$  \[\alpha \neq \text{null}\] $L_2$, Lem. C.7(1)
- $[\alpha]_{A^c} \in A^c \setminus O^{\text{passed}}$ Assumption
- $[\alpha]_{A^c} = [\alpha]_{A^c}$ Analogous to 59 – 61
- $[f(\alpha)]_{O}(\sigma^c_G) = [f(\alpha)]_{G}(\sigma^c_G)$ By def. of equality

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

- $[\alpha]_{A^c} \in A^c \setminus O^{\text{passed}}$, $[\alpha]_{A^c} \neq \emptyset$ See Case 1(a)1.
- $[\beta]_{A^c} \neq \emptyset$ \[\beta \neq \text{null}\] $L_2$, Lem. C.7(1)
- $[\beta]_{A^c} \in \text{map(sub(bind_{rel}))} A^c \setminus \{\emptyset\}$ $L_3$, $[\beta]_{A^c} \notin A^c \setminus O^{\text{passed}}$
- $(A^c \setminus O^{\text{passed}}) \cap \text{map(sub(bind_{rel}))} A^c \subseteq \{\emptyset\}$ $L_4$, Lem. C.8(2)
- $[\alpha]_{\sigma^c_G} = [\beta]_{\sigma^c_G}$ $[\alpha = \beta]_L(\sigma^c_G)$, Lem. C.7(2)
- 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. \([\alpha]_{A^r} \neq \emptyset\) \[\alpha \neq \text{null}(L(\sigma^L_1)), \text{Lem. C.7}(1)\]
79. \([\alpha]_{A^r} \in (\text{map}(\text{sub}(\text{bind}_{ret})) \backslash A^r) \backslash \{\emptyset\}\) \[\text{78, Assumption, Lem. C.9}(2)\]
80. \([\beta]_{A^r} \neq \emptyset\) \[\beta \neq \text{null}(L(\sigma^L_1)), \text{Lem. C.7}(1)\]
81. \([\beta]_{A^r} \in (\text{map}(\text{sub}(\text{bind}_{ret})) \backslash A^r) \backslash \{\emptyset\}\) \[\text{80, Assumption, Lem. C.9}(2)\]
82. \([\alpha]_{A^r} = [\beta]_{A^r}\) \[\alpha = \beta \text{L}(\sigma^L_1), \text{Lem. C.7}(2)\]
83. \(\exists! o \in A^r, [\alpha]_{A^r} = [\beta]_{A^r} = \text{sub}(\text{bind}_{ret}) \ o \neq \emptyset\) \[78 - 82, \text{Def. of map, Lem. C.9}(4)\]

84. Let \(o\) be the unique object in \(A^r\) which satisfies 83.
Let \(\alpha_0, r^1_\beta, \delta^\alpha_0\) be the unique values determined by Lem. C.9(6) for \(o\) and \(\alpha\).
Let \(\beta_0, r^1_\beta, \delta^\alpha_0\) be the unique values determined by Lem. C.9(6) for \(o\) and \(\beta\).

85. \(\alpha_0 \neq (y, e) \implies \alpha_0 \in \text{Bypass}(O_{\text{passed}}) \ [\alpha_0]_{\sigma^L_1}, [\alpha_0]_{A^r} \in O_{\text{args}} \cup O_{\text{cp}}, \ [\alpha_0]_{A^r} \in O_{\text{args}} \implies r^1_\beta = \text{bind}_{\text{args}} \ [\alpha_0]_{A^r} \neq \emptyset \ [\alpha_0]_{A^r} \in O_{\text{cp}} \implies r^1_\beta = \text{bind}_{\text{cp}} \ [\alpha_0]_{A^r} \neq \emptyset.\) \[\text{Lem. C.9}(6)\]

86. \(\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f((y, e)) \rrbracket_{G}(\sigma^G_0) \ \alpha_0 = (y, e)\} \ [\text{Def. of } f]\)
87. \(\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket (y, e) \rrbracket_{G}(\sigma^G_0) \ \alpha_0 = (y, e)\}\)
88. \(\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket (\text{ret}, e) \rrbracket_{G}(\sigma^G_0) \ \alpha_0 = (y, e)\} \ [\text{Def. of } G_{\text{args}} \text{ see Fig. 5}]\)
89. \(\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) \ \alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
90. \(\forall \alpha_0 \in r^1_\beta, \llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \llbracket f_{\text{e}}(\alpha_0) \rrbracket_{G}(\sigma^G_0)\) \[88, \text{Lem. C.13}(5)\]
91. \(\llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) \ \alpha_0 \neq (y, e)\} \ [\text{Def. of } f]\) \[\alpha = \alpha_0, \sigma^G_0, \text{Lem. C.6}(1), \text{Lem. C.12}\]
92. \(\forall \beta_0 \in r^1_\beta, \llbracket f(\beta_0) \rrbracket_{G}(\sigma^G_0) = \llbracket f_{\text{e}}(\beta_0) \rrbracket_{G}(\sigma^G_0)\) \[89, \text{Lem. C.9}(6)\]
93. \(\llbracket f(\beta) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\beta_0) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
94. \(\forall \alpha_0 \in r^1_\beta, \forall \beta_0 \in r^1_\beta, [\alpha_0, \sigma^G_0, \delta^\alpha_0] \subseteq [\beta_0, \sigma^G_0, \delta^\alpha_0].\) \[\text{Admissibility of } \sigma^G_0\]
95. \(\sigma^G_L \propto_{J_f} \sigma^G_0\) \[95 \text{ – 96}\]
96. \(\forall \alpha_0 \in r^1_\beta, \forall \beta_0 \in r^1_\beta, [\llbracket f_{\text{e}}(\alpha_0, \delta^\alpha_0) \rrbracket_{G}(\sigma^G_0) = \llbracket f_{\text{e}}(\beta_0, \delta^\alpha_0) \rrbracket_{G}(\sigma^G_0)\]
97. \(\forall \alpha_0 \in r^1_\beta, \forall \beta_0 \in r^1_\beta, [\llbracket f_{\text{e}}(\alpha_0, \delta^\alpha_0) \rrbracket_{G}(\sigma^G_0) = \llbracket f_{\text{e}}(\beta_0, \delta^\alpha_0) \rrbracket_{G}(\sigma^G_0)\]
98. \(\forall \alpha_0 \in r^1_\beta, \forall \beta_0 \in r^1_\beta, \llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha_0) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
99. \(\llbracket f(\beta_0) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\beta_0) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
100. \(\forall \alpha_0 \in r^1_\beta, \forall \alpha_0 \in r^1_\beta, \llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
101. \(\forall \alpha_0 \in r^1_\beta, \forall \alpha_0 \in r^1_\beta, \llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
102. \(\forall \alpha_0 \in r^1_\beta, \forall \alpha_0 \in r^1_\beta, \llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)
103. \(\forall \alpha_0 \in r^1_\beta, \forall \alpha_0 \in r^1_\beta, \llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) = \{\llbracket f(\alpha) \rrbracket_{G}(\sigma^G_0) \ \delta^\alpha_0 \neq (y, e)\} \ [\text{Def. of } f_{\text{e}}]\)

48
Case 1b:

104. $[\alpha = \text{null}]_L(\sigma^*_L)$ Assumption
105. $[\beta = \text{null}]_L(\sigma^*_L)$ Assumption
106. $[f(\alpha) = \text{null}]_G(\sigma^*_G)$ 104, $\sigma^*_L \propto_f \sigma^*_G$, Case 2
107. $[f(\beta) = \text{null}]_G(\sigma^*_G)$ 105, $\sigma^*_L \propto_f \sigma^*_G$, Case 2
108. $[f(\alpha) = f(\beta)]_G(\sigma^*_G)$ 106, 107

Case 2a:

109. $[(y, \delta_y) = \text{null}]_L(\sigma^*_L)$ By assumption
110. $[(\text{ret}, \delta_y) = \text{null}]_L(\sigma^*_L)$ Lem. C.9(5a)
111. $\sigma^*_L \propto_f \sigma^*_G$, the induction assumption
112. $[f(\text{ret}, \delta_y) = \text{null}]_G(\sigma^*_G)$ 110, 111
113. $[(y, \delta_y) = \text{null}]_G(\sigma^*_G)$ 112, Def. of $f_e$
114. $[(y, \delta_y) = \text{null}]_G(\sigma^*_G)$ $h^e = h^e, \rho^e(y) = \rho^e(\text{ret})$

Case 2(b1):

115. $\forall \gamma' \leq \gamma, [\gamma']_{A^e} \in (A^e \setminus O^{\text{passed}}) \cup \{\emptyset\}$ Assumption
116. $A^e = (A^e \setminus O^{\text{passed}}) \cup \text{map}(\text{sub}(\text{bind}_{\text{ret}})) A^e$ Def. of $\mathcal{LSL}$ (see Fig. 11).
117. $\forall \gamma' \leq \gamma, [\gamma']_{\sigma^*_G} \not\in O^{\text{passed}}$ 116, Lem. C.9(3), Lem. C.8(1)
118. $[\gamma = \text{null}]_L(\sigma^*_L)$ 115, 117, $\gamma \leq \gamma$, Lem. C.9(5b)
119. $[f(\gamma) = \text{null}]_G(\sigma^*_G)$ 118, $\sigma^*_L \propto_f \sigma^*_G$

We continue with case analysis w.r.t. the value of $[(r_\gamma, \epsilon)]_G(\sigma^*_G)$

- Assume $[(r_\gamma, \epsilon)]_G(\sigma^*_G) = \text{null}$

120. $[\gamma = \text{null}]_L(\sigma^*_L)$ 119, Lem. C.9(6), Assumption $\langle r_\gamma \neq y \rangle$
121. $[\langle r_\gamma, \epsilon \rangle = \text{null}]_L(\sigma^*_L)$ 119, Lem. C.9(5a)
122. $[f(\langle r_\gamma, \epsilon \rangle) = \text{null}]_G(\sigma^*_G)$ 122, $\sigma^*_L \propto_f \sigma^*_G$
123. $[\langle r_\gamma, \epsilon \rangle = \text{null}]_G(\sigma^*_G)$ 122, Def. of $f$
124. $[\langle r_\gamma, \epsilon \rangle = \text{null}]_G(\sigma^*_G)$ Def. of $\mathcal{GSB}$
125. $\langle r_\gamma, \epsilon \rangle \leq \gamma$, Lem. C.6(2a)

- Assume $[(r_\gamma, \epsilon)]_G(\sigma^*_G) \neq \text{null}$

126. $\exists s \in \text{FieldId}, \exists \gamma', s \leq \gamma, \langle f(\gamma') \neq \text{null} \rangle_\mathcal{G}(\sigma^*_L)$

127. $\gamma' \wedge \forall \gamma'' \leq \gamma, [f(\gamma'') = \text{null}]_G(\sigma^*_G)$

128. Let $f$ and $\gamma'$ be the unique values satisfying 127

129. $[f(\gamma')]_G(\sigma^*_G) \not\in L^{\text{reach}}$ 117, 128, Lem. C.13(4)
130. $[f(\gamma')]_G(\sigma^*_G) \in L \setminus L^{\text{reach}}$ 129, $[f(\gamma') \neq \text{null}]_G(\sigma^*_L)$
131. $[f(\gamma')]_G(\sigma^*_G) = [f(\gamma')]_G(\sigma^*_G)$ 130, Lem. C.5, Lem. C.12
132. $[f(\gamma')]_G(\sigma^*_G) = \text{null}$ 127 – 128, 131, Lem. C.12
133. $[f(\gamma')]_G(\sigma^*_G) = \text{null}$ 122, Def. of equality w. $\text{null}$
134. $[f(\gamma) = \text{null}]_G(\sigma^*_G)$ 128, 133, Lem. C.6(2a), Lem. C.12

49
Case 2(b2):

135. \([\gamma]_{\sigma_L} = \emptyset\) 

\[\text{Lem. C.7(1)}\]
\[([\gamma] = \text{null}_L(\sigma^L_\gamma))\]

136. \(\exists \gamma' \leq \gamma, [\gamma']_{\sigma_L} \not\in (A^c \setminus O^{\text{Passed}}) \cup \{\emptyset\}\)

\[\text{Assumption}\]

137. \(\forall \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L} \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \setminus \{\emptyset\}, \forall \gamma' \leq \gamma'' \Rightarrow [\gamma'']_{\sigma_L} \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \cup \{\emptyset\}\)

\[\text{135 - 136, Lem. C.9(2)}\]

138. \(\exists \gamma', \gamma' < \gamma, [\gamma']_{\sigma_L} \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \setminus \{\emptyset\}, \forall \gamma' \leq \gamma'' \Rightarrow [\gamma'']_{\sigma_L} \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \cup \{\emptyset\}\)

\[\text{Lem. C.7(1)}\]

139. \(\exists \gamma', \gamma' \leq \gamma, \exists \gamma'' \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \wedge [\gamma' \neq \text{null}_L(\sigma^L_\gamma)] \wedge [\gamma'' \leq \gamma'' \Rightarrow [\gamma''] = \text{null}_L(\sigma^L_\gamma)]\)

\[\text{Lem. C.7(1, 3, 4)}\]

140. \(\exists \gamma', \gamma' \leq \gamma, \exists \gamma'' \in \text{map}(\text{sub}(\text{bind}_{ret})) A^c \wedge [\gamma' \neq \text{null}_L(\sigma^L_\gamma)] \wedge [\gamma'' \leq \gamma'' \Rightarrow [\gamma''] = \text{null}_L(\sigma^L_\gamma)]\)

141. \(\text{Def. of map}\)

142. \(\text{let } \gamma_0, \gamma_1, \delta^t, \sigma^t \text{ be the unique values determined by Lem. C.9(6) for } o \text{ and } \gamma'.\)

143. \(\forall \gamma_p \in r_1, [\gamma_p, \delta^t, s] = \text{null}_L(\sigma^L_p)\)

\[\text{Lem. C.9(5c)}\]

144. \(\sigma^t_L \propto_{f_L} \sigma^t_G\)

\[\text{Assumption}\]

145. \(\forall \gamma_p \in r_1, [[f_\gamma(\gamma_p, \delta^t_s)]_G(\sigma^t_L) = \text{null}\)

\[\text{143 - 144}\]

146. \(\forall \gamma_p \in r_1, [[f_\gamma(\gamma_p, \delta^t_s)]_G(\sigma^t_L) = \text{null}\)

\[\text{144, Lem. C.12}\]

147. \(\forall \gamma_p \in r_1, [[f_\gamma(\gamma_p, \delta^t_s)]_G(\sigma^t_L) = \text{null}\)

\[\text{147, Lem. C.12}\]

148. \(\forall \gamma_p \in r_1, [[f_\gamma(\gamma_p, \delta^t_s)]_G(\sigma^t_L) = \text{null}\)

\[\text{See proof for Case 1(a)4}\]

149. \(\forall \gamma_p \in r_1, [[f_\gamma(\gamma_p, \delta^t_s)]_G(\sigma^t_L) = \text{null}\)

\[\text{148 - 149}\]

Theorem C.15 (Context-aware Equivalence Preservation) Let \(p\) be a function. Let \(\sigma_L \in \Sigma^p_L\) and \(\sigma_G \in \Sigma^p_G\) 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 \(\sigma' \in \Sigma^p_L\) such that \(\langle st, \sigma_L \rangle \xrightarrow{L} \sigma'\), there exists a state \(\sigma' \in \Sigma^p_G\) such that:
   - \(\langle st, \sigma_G \rangle \xrightarrow{G} \sigma'_G\), and
   - \(\sigma'_L \propto_{f} \sigma'_G\).

2. For any state \(\sigma' \in \Sigma^p_L\) such that \(\langle st, \sigma_G \rangle \xrightarrow{G} \sigma'_G\), there exists a state \(\sigma' \in \Sigma^p_L\) such that:
   - \(\langle st, \sigma_L \rangle \xrightarrow{L} \sigma'_L\), and
   - \(\sigma'_L \propto_{f} \sigma'_G\).

Proof: Let \(q\) be a function. Let \(\sigma_L = \langle CPL, A \rangle \in \Sigma^p_L\) and \(\sigma_G = \langle L, \rho, h \rangle \in \Sigma^p_G\) 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 \(\sigma_L\) and \(\sigma_G\) are also context-aware equivalent w.r.t. to \(f\). The proof is done by a case analysis.
\textbf{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.

\textbf{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
\[ \exists \sigma' \in \Sigma_0, \text{ s.t. } (x = \text{null}, \sigma') \overset{L}{\rightarrow} \sigma'_L \text{ and } \exists \sigma' \in \Sigma_0, \text{ s.t. } (x = \text{null}, \sigma) \overset{G}{\rightarrow} \sigma'_G. \]

By definition, \[ \sigma'_L = (\text{CPL, rem(A, \{x, \Delta\})) = (\text{CPL, } (\text{map}(\lambda x. \Delta \setminus \{x\}) A) \setminus \{\emptyset\}) = (\text{CPL, } \{a \setminus x, \Delta | a \in A \setminus \{\emptyset\}} \]
and \[ \sigma'_G = (L, \rho[\text{x} \mapsto \text{null}, h]). \]

Let \[ \alpha = (r_\alpha, \delta_\alpha), \beta = (r_\beta, \delta_\beta), \text{ and } \gamma = (r_\gamma, \delta_\gamma) \] be generalized access paths of function \( q \).

\[ [\alpha = \beta]_{L}^{\sigma_L} \]
\[ \forall a' \in \{a \setminus \{x\}, \Delta | a \in A \setminus \{\emptyset\}, \alpha \in a' \iff \beta \in a' \] \[ \forall a' \in \{a \setminus \{x\}, \Delta | a \in A \setminus \{\emptyset\}, \alpha \in a' \iff \beta \in a' \] \[ r_\alpha \neq x, r_\beta \neq x, \forall a \in A, \alpha \in a \iff \beta \in a \text{ or } \]
\[ r_\alpha = x, r_\beta \neq x, \forall a \in A, \beta \not\in a \text{ or } \]
\[ r_\alpha \neq x, r_\beta = x, \forall a \in A, \alpha \not\in a \text{ or } \]
\[ r_\alpha = x, r_\beta = x \]
\[ r_\alpha \neq x, r_\beta \neq x, [\alpha = \beta]_{L}^{\sigma_L} \text{ or } \]
\[ r_\alpha = x, r_\beta \neq x, [\beta = \text{null}]_{L}^{\sigma_L} \text{ or } \]
\[ r_\alpha \neq x, r_\beta = x, [\alpha = \text{null}]_{L}^{\sigma_L} \text{ or } \]
\[ r_\alpha = x, r_\beta = x \]
\[ r_\alpha \neq x, r_\beta \neq x, [\hat{f}(\alpha) = \hat{f}(\beta)]_{G}^{\sigma_G} \text{ or } \]
\[ r_\alpha = x, r_\beta \neq x, [\hat{f}(\beta) = \text{null}]_{G}^{\sigma_G} \text{ or } \]
\[ r_\alpha \neq x, r_\beta = x, [\hat{f}(\alpha) = \text{null}]_{G}^{\sigma_G} \text{ or } \]
\[ r_\alpha = x, r_\beta = x \]
\[ [\hat{f}(\alpha)]_{G}^{\sigma'_G} = [\hat{f}(\beta)]_{G}^{\sigma'_G} \]
\[ [\hat{f}(\alpha)]_{G}^{\sigma'_G} = [\hat{f}(\beta)]_{G}^{\sigma'_G} \]
\[ [\gamma = \text{null}]_{L}^{\sigma_L} \]
\[ \forall a' \in \{a \setminus \{x\}, \Delta | a \in A \setminus \{\emptyset\}, \gamma \not\in a' \] \[ \forall a' \in \{a \setminus \{x\}, \Delta | a \in A \setminus \{\emptyset\}, \gamma \not\in a' \] \[ r_\gamma \neq x, \forall a \in A, \gamma \not\in a \text{ or } \]
\[ r_\gamma = x \]
\[ r_\gamma \neq x, [\gamma = \text{null}]_{L}^{\sigma_L} \text{ or } \]
\[ r_\gamma = x \]
\[ r_\gamma \neq x, [\hat{f}(\gamma) = \text{null}]_{G}^{\sigma_G} \text{ or } \]
\[ r_\gamma = x \]
\[ [\hat{f}(\gamma)]_{G}^{\sigma'_G} = \text{null} \]
\[ [\hat{f}(\gamma)]_{G}^{\sigma'_G} = \text{null} \]

\textbf{x=y} Analogous to [x=null].

\textbf{x=y.f} Analogous to [x=null].

\textbf{x.f=null} Analogous to [x=null].

\textbf{x.f=x} Analogous to [x=null].

\textbf{x=alloc t} Analogous to [x=null].

\textbf{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.
Assume that $st = st_1; st_2$ and that $(st_1; st_2, \sigma_L) \xrightarrow{L} \sigma'_L$.

155. $(st_1; st_2, \sigma_L) \xrightarrow{L} \sigma'_L$

156. $\exists \sigma'_L, (st_1, \sigma_L) \xrightarrow{L} \sigma''_L \wedge (st_2, \sigma''_L) \xrightarrow{L} \sigma'_L$

157. $\exists \sigma''_L, \exists \sigma''_G, (st_1, \sigma_L) \xrightarrow{L} \sigma''_L \wedge (st_2, \sigma''_L) \xrightarrow{L} \sigma'_L \wedge (st_1, \sigma_G) \xrightarrow{G} \sigma''_G \wedge \sigma''_L \propto f \sigma''_G$

158. $\exists \sigma''_G, (st_1; st_2, \sigma_G) \xrightarrow{G} \sigma''_G \wedge (st_2, \sigma''_G) \xrightarrow{G} \sigma'_G \wedge \sigma''_L \propto f \sigma''_G$

159. $\exists \sigma_G, (st_1; st_2, \sigma_G) \xrightarrow{G} \sigma''_G \wedge \sigma'_L \propto f \sigma''_G$

The proof in the other direction is analogous.

if-ff Analogous to [seq].

while Analogous to [seq].

**Induction step (interprocedural):**

The transition labeling the root of the derivation tree contains a function call. Thus, the induced derivation tree is not a leaf. Without loss of generality, assume that the invocation is $y = p(x_1, \ldots, x_k)$. To simplify notations, we assume that $\sigma'_L = \sigma_L$ and that $\sigma''_G = \sigma_G$.

Assume that $(y = p(x_1, \ldots, x_k), \sigma'_L) \xrightarrow{L} \sigma''_L$.

160. $(y = p(x_1, \ldots, x_k), \sigma'_L) \xrightarrow{L} \sigma''_L$

161. $\exists \sigma''_L, \sigma''_L \in \Sigma_G^p, \sigma'_L \in \Sigma_L^p, s.t. \text{(body of } p, \sigma'_L) \xrightarrow{L} \sigma''_L \wedge \sigma''_L \text{ and } \sigma''_L \text{ as defined in Fig. 11}$

162. Let $f_e$ be that state that arise at the entry to $p$ when $p$ is invoked at $\sigma''_L$.

163. Let $f_e$ be that renaming function defined as in Lem. C.14 for $\sigma''_L$ and $\sigma''_G$

164. $\sigma''_L \propto f_e \sigma''_G$

165. $\exists \sigma''_G \in \Sigma_G^p, (\text{body of } p, \sigma''_G) \xrightarrow{G} \sigma''_G \wedge \sigma''_L \propto f_e \sigma''_G$

166. $\exists \sigma''_G \in \Sigma_G^p, (y = p(x_1, \ldots, x_k), \sigma''_G) \xrightarrow{G} \sigma''_G \wedge \sigma''_G \text{ as defined in Fig. 5 for } \sigma''_G$ and $\sigma''_G$

167. $\sigma''_L \propto f \sigma''_G$

The proof in the other direction is analogous.