A Semantics for Procedure Local Heaps and its Abstractions

Noam Rinetzky[*]
Tel Aviv Univ.
Tel Aviv, Israel
maon@tau.ac.il

Jörg Bauer[†]
Univ. des Saarlandes
Saarbrücken, Germany
joba@cs.uni-sb.de

Thomas Reps[‡]
Univ. of Wisconsin
Madison, USA
reps@cs.wisc.edu

Mooly Sagiv[‡]
Tel Aviv Univ.
Tel Aviv, Israel
msagiv@tau.ac.il

Reinhard Wilhelm
Univ. des Saarlandes
Saarbrücken, Germany
wilhelm@cs.uni-sb.de

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.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—Semantics; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—assertions;invariants; Semantics of Programming Languages—Operational semantics; Program analysis

General Terms

Languages, Theory, Verification

Keywords

Abstract interpretation, shape analysis, static analysis, 3-valued logic

[*]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.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

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 [15]. 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" [8, 15], 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 [10]. 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., [5, 7, 9, 18, 20, 21]. A storeless *concrete* semantics has already done this step, which relieves the designer of an abstraction from having to do it. Thus, it is natural to base powerful pointer (shape) analysis algorithms on storeless semantics. Unfortunately, existing storeless semantics associate the entire heap with each procedure invocation and class instantiation, which makes it difficult to support procedure and data abstraction. Another problem with storeless semantics is that it is hard to relate properties of memory cells before and after a call. As a result, it is hard to scale these methods to prove properties of real-life programs. By "scaling", we mean not just cost issues but also precision. In particular, after a procedure call some information about the calling context may be lost.

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

1.2 Main Results

In this paper, we develop a method to characterize a procedure's behavior in a way that ignores parts of the heap that are not relevant to the procedure. Toward this end, the paper introduces a non-standard storeless *concrete* semantics, $\mathcal{LSL}$, for *Localized-heap Store-Less*. In this semantics, a called procedure is only passed a *part* of the heap. Based on this semantics, a new static-analysis algorithm is developed using canonical abstraction [20]. 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 [7].

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

$\mathcal{LSL}$ is based on the following ideas: Objects in the heap reachable from an actual parameter are treated differently when they separate the "local heap" that can be accessed by a procedure from the rest of the heap, which—from the viewpoint of that procedure—is non-accessible and immutable. We call these objects *cutpoints*. An object *belongs to the local-heap* when it is reachable from a procedure's actual parameters. Such an object is a *cutpoint* when it is reached via a pointer-access path that starts at a variable of a *pending* call and does not *traverse* the local-heap. When a procedure returns, the cutpoints are used to update the caller's local-heap with the effect of the call. Because our goal is to perform static analysis, $\mathcal{LSL}$ is a *storeless semantics* [10]; every dynamically allocated object o is represented by the set of *access paths* that reach o. In particular, unreachable objects are not represented. $\mathcal{LSL}$ is different from previous storeless semantics based on pointer-access paths [5,21] in the following way. It does not represent access paths that start from variables of pending calls in the "local state" of the current procedure. This means that a procedure has a local view that only includes objects that are reachable from the procedure's parameters and, in addition, any objects that it allocates.

We characterize the manner in which $\mathcal{LSL}$ simulates a standard store-based semantics and identify a class of observations for which $\mathcal{LSL}$ is equivalent to the standard store-based semantics. This allows us to prove properties ranging from the absence of runtime errors to partial and total correctness with respect to the standard store-based semantics.

The second part of the paper uses $\mathcal{LSL}$ as the starting point for static-analysis algorithms that treat the heap in a more local, more modular way than previous work. In this part of the paper, we present a new interprocedural shape-analysis algorithm for programs that manipulate dynamically allocated storage. The algorithm is based on an abstraction of $\mathcal{LSL}$. The new algorithm can 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. 18). Furthermore, the analysis is done in a way that is more likely to scale up. In particular, our analysis benefits from the fact that the heap is localized: the behavior of a procedure only depends on the contents of its local-heap. This allows analysis results to be reused for different contexts.

1.3 Outline

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

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

Figure 1: Syntax of EAlgol.

2. PRELIMINARIES

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

2.1 Syntax of EAlgol

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

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

2.2 Running Example

The **EAlgol** program shown in Fig. 2 is our running example. The program consists of a type definition for an element in a linked list (`Sll`); three list-manipulating functions: create (`crt`), destructive append (`app`), and destructive reverse (`reverse`); and a `main` function.

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

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

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

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

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

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

2.3 Global-Heap Store-Based Semantics

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

Fig. 3 defines the semantic domains. *Loc* is an unbounded set of memory locations. A *memory state* for a function p, $\sigma_G^p \in \Sigma_G^p$, keeps track of the allocated memory locations, L, an environment mapping p's local variables to values, ρ, and a mapping from fields of *allocated* locations to values, h. Due to our simplifying assumptions, a value is either a memory location or *null*.

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

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

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

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

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

$$\langle \mathtt{x} = \mathtt{null}, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto null], h\rangle$$
$$\langle \mathtt{x} = \mathtt{y}, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto \rho(y)], h\rangle$$
$$\langle \mathtt{x} = \mathtt{y.f}, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L, \rho[x \mapsto h(\rho(y), f)], h\rangle \quad (\mathbf{1})$$
$$\langle \mathtt{x.f} = \mathtt{null}, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto null]\rangle \quad (\mathbf{2})$$
$$\langle \mathtt{x.f} = \mathtt{y}, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L, \rho, h[(\rho(x), f) \mapsto \rho(y)]\rangle \quad (\mathbf{2})$$
$$\langle \mathtt{x} = \mathtt{alloc}\ t, \langle L, \rho, h\rangle\rangle \overset{G}{\rightsquigarrow} \langle L \cup \{l\}, \rho[x \mapsto l], h \cup I(l)\rangle \quad (\mathbf{3})$$

Figure 4: Axioms for atomic statements in the $\mathcal{GSB}$ semantics. The side-conditions are: (1) $\rho(y) \neq$ *null*, (2) $\rho(x) \neq$ *null*, and (3) $l \notin L$. I initializes all pointer fields at l to *null*.

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

where

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

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

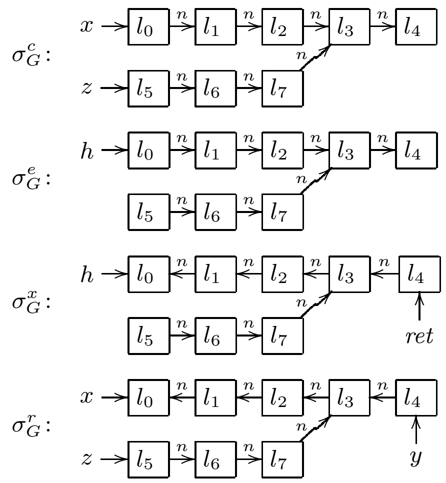
2.4 Observable Properties

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

DEFINITION 2.1 (FIELD PATHS). *A* ***field path*** $\delta \in \Delta = FieldId^*$ *is a (possibly empty) sequence of field identifiers. The empty sequence is denoted by* ϵ.

DEFINITION 2.2 (ACCESS PATH). *An* ***access path*** $\alpha = \langle x, \delta\rangle \in V_p \times \Delta$ *of a function p is a pair consisting of a local variable of p and a field path.* $AccPath_p$ *denotes the set of all access paths of function p.* $AccPath$ *denotes the union of all access paths of all functions in a program.*

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



A^c: $\{x\}$, $\{x.n\}$, $\{x.n.n\}$, $\{x.n.n.n,\ z.n.n.n\}$, $\{x.n.n.n.n,\ z.n.n.n.n\}$, $\{z\}$, $\{z.n\}$, $\{z.n.n\}$

A^e: $\{\ h\ \}, \{\ h.n\ \}, \{\ h.n.n\ \}, \left\{\begin{array}{c} h.n.n.n, \\ cpl \end{array}\right\}, \left\{\begin{array}{c} h.n.n.n.n \\ cpl.n \end{array}\right\}$

A^x: $\left\{\begin{array}{c} h,\ cpl.n.n.n \\ ret.n.n.n.n \end{array}\right\}, \left\{\begin{array}{c} cpl.n.n, \\ ret.n.n.n \end{array}\right\}, \left\{\begin{array}{c} cpl.n, \\ ret.n.n \end{array}\right\}, \left\{\begin{array}{c} cpl \\ ret.n \end{array}\right\}, \{ret\}$

A^r: $\left\{\begin{array}{c} x,\ y.n.n.n.n, \\ z.n.n.n.n.n.n \end{array}\right\}, \left\{\begin{array}{c} y.n.n.n, \\ z.n.n.n.n.n \end{array}\right\}, \left\{\begin{array}{c} y.n.n, \\ z.n.n.n.n \end{array}\right\}, \left\{\begin{array}{c} y.n, \\ z.n.n.n \end{array}\right\}, \{\,y\,\}$, $\{\,z\,\}, \{\,z.n\,\}, \{\,z.n.n\,\}$

global heap | local heap | $cpl = \{\widehat{h.n.n.n}\}$

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

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

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

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

DEFINITION 2.3 (ACCESS-PATH EQUALITY). *Access paths α and β are **equal** in a given state σ_G, denoted by $[\![\alpha = \beta]\!]_G(\sigma_G)$, if they have the same value in that state, i.e., $[\![\alpha]\!]_G(\sigma_G) = [\![\beta]\!]_G(\sigma_G)$. An access path is **equal to null**, denoted by $[\![\alpha = \mathtt{null}]\!]_G(\sigma_G)$, if $[\![\alpha]\!]_G(\sigma_G) = null$.*

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

3. CUTPOINTS AND THEIR USE

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

In $\mathcal{LSL}$, every dynamically allocated object o is represented by the set of pointer-access paths that reach o. Unlike existing storeless semantics [5], in $\mathcal{LSL}$, pending access paths are not represented as parts of the local state of the current procedure. The advantage of our approach is that when a procedure is invoked, it operates only on a part of the heap, namely, the objects that are reachable from the procedure's actual parameters. The downside of this approach is that the memory state just after the call cannot always be defined in terms of the state prior to the call. The intuitive reason for this deficiency is that the description of an object may change due to destructive updates. For example, in the running example, to determine that the pointer-access paths `y.n.n` and `z.n.n.n` are aliased after the invocation of `reverse`, we need to know that the list element pointed-to by `h.n.n.n` when the execution of `reverse` begins, is pointed-to by `ret.n` when the execution ends. To capture this kind of temporal relationship, $\mathcal{LSL}$ tracks the effect of a function on *cutpoints*. Cutpoints are the objects that separate the part of the heap that an invoked function can access from the rest of the heap (excluding the objects pointed-to by actual parameters).

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

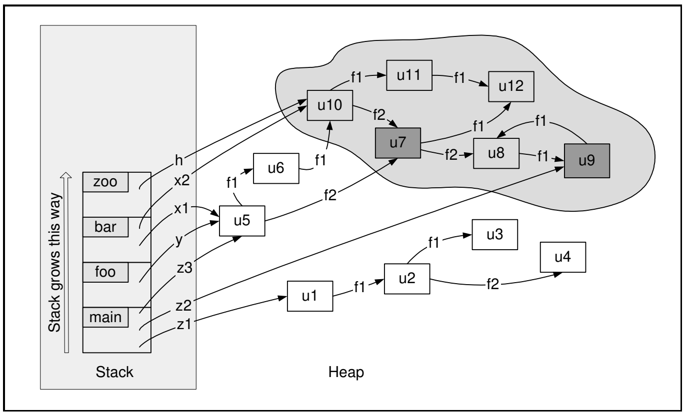


Figure 7: An illustration of the cutpoints for an invocation in a store-based small-step (stack-based) operational semantics. The figure depicts the memory state at the entry to `zoo`. The stack of activation record is depicted on the left side of the diagram. Each activation record is labeled with the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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. THE LOCALIZED-HEAP STORELESS SEMANTICS

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

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

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

r	$\in$	$Root_p = V_p \cup CPLbs_p$	
α, β	$\in$	$GAccPath_p = Root_p \times \Delta$	
o	$\in$	$Obj_L^p = 2^{GAccPath_p}$	Objects
A, A_p	$\in$	$Heap_L^p = 2^{Obj_L^p}$	Heaps
σ_L	$\in$	$\Sigma_L^p = 2^{CPLbs_p} \times Heap_L^p$	Memory state

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

4.1 Memory States

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

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

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

DEFINITION 4.1 (ADMISSIBLE MEMORY STATES). *A memory state $\langle \mathrm{CPL}_p, A_p\rangle$ for a function p at a given point in an execution is **admissible** iff (i) A generalized access path points-to (at most) one object, i.e., $\forall o, o' \in A_p$ if $o \neq o'$, then $o \cap o' = \emptyset$; (ii) A is right-regular, i.e., $\forall o_1, o_2 \in A_p$ if $\alpha, \beta \in o_1$ and $\alpha.\delta \in o_2$ then $\beta.\delta \in o_2$; (iii) A_p is prefix-closed, i.e., if $\alpha.f \in$ flat A_p, then $\alpha \in$ 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 $\langle r, \delta\rangle \in$ 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.

Because $\mathcal{LSL}$ preserves admissibility of states (see [17]), in the sequel, whenever we refer to an $\mathcal{LSL}$ state, we mean an *admissible* $\mathcal{LSL}$ state.

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

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

4.2 Inference Rules

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

4.2.1 Helper Functions

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

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

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

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

4.2.2 Atomic Statements

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

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

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

$$
\begin{array}{l}
.: \mathit{GAccPath} \times \Delta \rightarrow \mathit{GAccPath} \text{ s.t.} \\
\quad \langle r, \delta \rangle.\delta' \stackrel{\text{def}}{=} \langle r, \delta\delta' \rangle \\
.: 2^{\mathit{GAccPath}} \times \Delta \rightarrow 2^{\mathit{GAccPath}} \text{ s.t.} \\
\quad a.\delta \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a\} \\
.: 2^{\mathit{GAccPath}} \times 2^{\Delta} \rightarrow 2^{\mathit{GAccPath}} \text{ s.t.} \\
\quad a.D \stackrel{\text{def}}{=} \{\alpha.\delta \mid \alpha \in a, \delta \in D\} \\
[]: \mathit{GAccPath} \times \mathit{Heap}_L \rightarrow \mathit{Obj}_L \text{ s.t.} \\
\quad [\alpha]_A \stackrel{\text{def}}{=} \{\beta \in a \mid a \in A, \alpha \in a\} \\
\mathit{rem}: \mathit{Heap}_L \times 2^{\mathit{GAccPath}} \rightarrow \mathit{Heap}_L \text{ s.t.} \\
\quad \mathit{rem}(A, a) \stackrel{\text{def}}{=} (\mathit{map}(\lambda o.o \setminus a.\Delta)\ A) \setminus \{\emptyset\} \\
\mathit{add}: \mathit{Heap}_L \times 2^{\mathit{GAccPath}} \times \mathit{GAccPath} \rightarrow \mathit{Heap}_L \text{ s.t.} \\
\quad \mathit{add}(A, a, \alpha) \stackrel{\text{def}}{=} \mathit{map}(\lambda o.\, o \cup a.\{\delta \in \Delta \mid \alpha.\delta \in o\})\ A
\end{array}
$$

Figure 9: Helper functions.

$$
\begin{array}{ll}
\langle \mathtt{x} = \mathtt{null}, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, \mathit{rem}(A, \{x\}) \rangle & \\
\langle \mathtt{x} = \mathtt{y}, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, \mathit{add}(A, \{x\}, y) \rangle & \\
\langle \mathtt{x} = \mathtt{y.f}, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, \mathit{add}(A, \{x\}, y.f) \rangle & (\mathbf{1}) \\
\langle \mathtt{x.f} = \mathtt{null}, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, \mathit{rem}(A, [x]_A.f) \rangle & (\mathbf{2}) \\
\langle \mathtt{x.f} = \mathtt{y}, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, \overline{\mathit{add}(A, [x]_A.f, y)} \rangle & (\mathbf{2}) \\
\langle \mathtt{x} = \mathtt{alloc}\ t, \langle \mathit{CPL}, A \rangle \rangle \stackrel{L}{\leadsto} \langle \mathit{CPL}, A \cup \{\{x\}\} \rangle &
\end{array}
$$

Figure 10: Axioms for atomic statements in the local heap semantics. Note that the set of cutpoint-labels is not changed. The side-conditions are: $(\mathbf{1})$ $y \in \mathit{flat}\,A$ and $(\mathbf{2})$ $x \in \mathit{flat}\,A$. The side-condition $x \in \mathit{flat}\,A$ (resp. $y \in \mathit{flat}\,A$) means that `x`'s (resp. `y`) value is not *null*.

heap, the singleton set $\{x\}$, and the access path y.

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

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

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

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

4.2.3 Function Calls

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

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

The main idea behind the rule is to utilize the fact that a function cannot modify objects that are not in its local-heap (i.e., in the part of the heap that is *not* reachable from any actual parameter when the function is invoked). In particular, because $\mathcal{LSL}$ describes objects in terms of the (generalized) access paths that point-to them, these "inaccessible" objects have the same description before and after the call. Thus, only the description of the objects in the function's local-heap (i.e., in the part of the heap that the function can access) is (possibly) updated. The update is carried out using the *cutpoints of the invocation*.[4] In essence, the semantics freezes the initial descriptions of the cutpoints and arranges for them to persist throughout the execution of the called function. This sets up a relation between values on entry to values on exit. At the return, the frozen information is used to update the description of objects in the called function's local-heap via an operation that is (roughly) similar to a relational join [3]. (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 *pointed-to* by p's actual parameters (O_c^{args}). Then, the auxiliary function *RObjs* finds the part of the caller's heap (A^c) that is reachable from these objects (O_c^{passed}).

The description of the objects after the call should account for

[3]The operator $\overline{\cdot}$ is similar to the ρ_{rstc} operator in [6].

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

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

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

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

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

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

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

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

$$
\begin{array}{l}
\mathit{RObjs} \colon \mathit{Heap}_L \to (2^{\mathit{Obj}_L} \to 2^{\mathit{Obj}_L})\ s.t. \\
\quad \mathit{RObjs}(A)\ O \stackrel{\text{def}}{=} \{o \in A \mid o' \in O,\ \delta \in \Delta,\ o'.\delta \subseteq o\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Bypass} \colon 2^{\mathit{Obj}_L} \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}})\ s.t. \\
\quad \mathit{Bypass}(O)\ o \stackrel{\text{def}}{=} \{\langle r, \delta\rangle \in o \mid \forall \delta' < \delta.\ \langle r, \delta'\rangle \notin \mathit{flat}\ O\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{sub} \colon (2^{\mathit{GAccPath}} \to 2^{\mathit{GAccPath}}) \to (\mathit{Obj}_L \to 2^{\mathit{GAccPath}}) \\
s.t. \\
\quad \mathit{sub}(\mathit{bind})\ o \stackrel{\text{def}}{=} \mathit{flat} \left\{ \mathit{bind}(a).\delta \;\middle|\; \begin{array}{c} a \in \mathit{dom}(\mathit{bind}), \\ \delta \in \Delta,\ a.\delta \subseteq o \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{CPObjs}_q \colon \Sigma_L^q \to (2^{\mathit{Obj}_L^q} \times 2^{\mathit{Obj}_L^q} \to 2^{\mathit{Obj}_L^q})\ s.t. \\
\quad \mathit{CPObjs}_q(\langle \mathit{CPL}^c, A^c\rangle)\ (O_c^{args}, O_c^{passed}) \stackrel{\text{def}}{=} \\
\quad \mathit{Let} \\
\qquad \begin{array}{lcl}
O_{deep} & = & O_c^{passed} \setminus O_c^{args} \\
O_{vars} & = & \{[\langle x, \epsilon\rangle]_{A^c} \in O_{deep} \mid x \in V_q\} \\
O_{\mathit{fld}} & = & \left\{ o \in O_{deep} \;\middle|\; \begin{array}{c} \exists o' \in A^c \setminus O_c^{passed}, \\ \exists f \in \mathit{FieldId},\ o'.f \subseteq o \end{array} \right\} \\
O_{cpl} & = & \{[\langle \mathit{cpl}, \epsilon\rangle]_{A^c} \in O_{deep} \mid \mathit{cpl} \in \mathit{CPL}^c\}
\end{array} \\
\quad \mathit{in} \\
\qquad O_{vars} \cup O_{cpl} \cup O_{\mathit{fld}}
\end{array}
$$

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

qualified as a cutpoint if (and only if) one of the following holds:

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

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

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

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

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

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

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

In particular, the fourth element in `x`'s list is a cutpoint for the invocation of `reverse` (see Sec. 4.1) and its label is $\{\widehat{h.n.n.n}\}$. Thus, when the execution of `reverse`'s body starts, the cutpoint is represented by the following set of (generalized) access paths: $\{h.n.n.n, \{\widehat{h.n.n.n}\}\}$. When the execution of the function body ends, the cutpoint-anchored paths in the representation of every object in A^x (see Fig. 6) are replaced by access paths that start with $z.n.n.n$, the only access path that points-to the cutpoint at the call-site and *bypasses* the objects that were passed to `reverse`. For example, the cutpoint-anchored path $\{\widehat{h.n.n.n}\}.n$ in the representation of the third element in the returned list is replaced by $z.n.n.n.n$.

4.3 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. 4.2. Note that in both semantics an access path is equal to `null` when it has a prefix which is equal to `null`.

DEFINITION 4.2 (ACCESS PATH EQUALITY). *Access paths α and β are **equal** in a given state σ_L, denoted by $[\![\alpha = \beta]\!]_L(\sigma_L)$, if $\forall a \in A. \alpha \in a \iff \beta \in a$. An access path α is **equal to null** in state σ_L, denoted by $[\![\alpha = \mathtt{null}]\!]_L(\sigma_L)$, if $\alpha \notin$ flat A.*

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

DEFINITION 4.3 (OBSERVATIONAL EQUIVALENCE). *Let p be a function. The states $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ are **observationally equivalent** if for all $\alpha, \beta, \gamma \in AccPath_p$,*

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

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

4.3.1 Semantic Equivalence

The following theorem is the main theorem in the paper. It states that $\mathcal{LSL}$ is equivalent to $\mathcal{GSB}$, in the sense that both behave equivalently w.r.t. termination, and that execution of statements preserves observational equivalence. A proof of the theorem is given in [17].

THEOREM 4.4 (EQUIVALENCE). *Let p be a function. Let $\sigma_L \in \Sigma_L^p$ and $\sigma_G \in \Sigma_G^p$ be observationally equivalent states. Let st be an arbitrary statement in p. The following holds:*

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

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

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

COROLLARY 4.5. *Let P be a program, p a function , lb a program point in p. For any $\alpha, \beta \in AccPath_p$, $[\![\alpha = \beta]\!]_L$ is an invariant of P at lb iff $[\![\alpha = \beta]\!]_G$ is an invariant of P at lb.*

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

COROLLARY 4.6. *A memory leak can occur only when a variable or a field is assigned `null`. Furthermore,*

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

Both Cor. 4.5 and Cor. 4.6 are corollaries of The. 4.4. In [17] 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.

[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.

5. SHAPE ANALYSIS

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

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

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 `reverse` returns, the variables `z` and `y` point-to acyclic linked lists with a shared tail.

The algorithm is presented in terms of the 3-valued-logic framework for program analysis of [20]. Technically, 3-valued logical structures are used to represent unbounded memory states. The tracked properties are encoded as predicates.

In this paper, we focus on the abstraction of $\mathcal{LSL}$ memory states. Due to lack of space, we do not give the full details of the analyses. In particular, the abstract transfer functions are not defined. Instead, we specify the analysis using the *best abstract transformer* [4]. A detailed description of the shape-analysis algorithm is given in [19].

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

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

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

To establish the Galois connection between the set of program states (ordered by set inclusion) and *3-Struct*, it suffices to show a *representation function* that maps a program state to its "most-precise representation" in *3-Struct* (e.g., see [14]). We define the function $\beta_{shape} : \Sigma_L \to$ *3-Struct*, which maps a local-heap to its most precise representation as a 3-valued logical structure. β_{shape} is a composition of two functions: (i) *to2VLS*: $\Sigma_L \to$ *2-Struct*, which maps a local-heap σ_L to an unbounded *2-valued* logical structure S, and (ii) *canonical abstraction*: *2-Struct* $\to$ *3-Struct* which conservatively bounds S (defined as usual in [20]).

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textit{to2VLS}\colon \Sigma_L \to \textit{2-Struct} \text{ s.t.}} \\
\multicolumn{3}{l}{\textit{to2VLS}(\langle \textit{CPL}, A\rangle) = S \text{ where } U^S = A \cup \textit{CPL} \text{ and}} \\
\quad \textit{isList}^S(v) & = & v \in A \\
\quad \textit{isLabel}^S(v) & = & v \in \textit{CPL} \\
\quad x^S(v) & = & v \in A \text{ and } x \in v \\
\quad n^S(v_1, v_2) & = & v_1 \in A, v_2 \in A \text{ and } v_1.n \subseteq v_2 \\
\quad r_x^S(v_1) & = & \exists \alpha \in v_1 \text{ s.t. } \langle x, \epsilon\rangle \leq \alpha \\
\quad \textit{ils}^S(v) & = & \exists \alpha.n \in v,\, \beta.n \in v \text{ s.t. } [\alpha]_A \neq [\beta]_A \\
\quad c^S(v) & = & \exists \alpha \in v,\, \beta \in v \text{ s.t. } \alpha < \beta \\
\quad \textit{eq}^S(v_1, v_2) & = & v_1 = v_2 \\
\quad \textit{lbl}^S(v_1, v_2) & = & v_1 \in \textit{CPL},\, v_2 \in A \text{ and } \langle v_1, \epsilon\rangle \in v_2 \\
\quad \textit{cp}^S(v) & = & \exists r \in \textit{CPL} \text{ s.t. } \langle r, \epsilon\rangle \in v \\
\quad r_{cp}^S(v) & = & \exists r \in \textit{CPL},\, \delta \in \Delta \text{ s.t. } \langle r, \delta\rangle \in v
\end{array}
$$

Figure 13: The function *to2VLS* maps states in Σ_L to 2-valued logical structures.

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

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

5.1.1 *Representing a Local-Heap by a 2-Valued Logical Structure*

The function *to2VLS*, defined in Fig. 13, maps a local heap $\sigma_L = \langle CPL, A\rangle$ to a *2-valued* logical structure S. Every object $o \in A$ and every cutpoint-label $cpl \in CPL$ 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 $\mathcal{P}$.

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

We explain the predicates' intended meanings through an example. In the example, we apply *to2VLS* to σ_L^e, the memory state at the entry point of `reverse` (shown in Fig. 6). The resulting 2-valued logical structure, denoted by S_e, is depicted in Fig. 14.

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

- The predicates $isList$ and $isLabel$ record whether a node represents a list element or a cutpoint. We draw nodes u that represent list elements, i.e., $isList^S(u) = 1$, as rectangles, e.g., nodes u_0–u_3; and we draw nodes v that represent cutpoint-labels, i.e., $isLabel^S(v) = 1$, as circles, e.g., node u_6.
- The predicates ***h***, ***n***, r_h, *ils*, *c*, and *eq* are an adaptation to local-heaps of the standard predicates used in the analysis of singly linked lists [13, 20].
  - 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.
  - 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.
  - 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.
  - 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*. Note that the predicate records only *local* sharing. In particular, $ils^{S_e}(u_2) = 0$, although in a "global-view" of the heap, the list element represented by u_2 is the `n`-successor of two list elements: one in the local heap (represented by u_{1b}) and one not in the local heap (the third element in the list pointed-to by `z`).
  - The unary predicate c holds at a node that resides on a cycle of `n`-fields.
  - The binary predicate *eq* records the equality relation. It is not drawn in the pictures.
- The predicates *lbl*, *cp*, and r_{cp} record information that is special for the abstraction of an $\mathcal{LSL}$ state.
  - The binary predicate *lbl* relates a node that represents a cutpoint-label to the node that represents the corresponding cutpoint. For example, $lbl^S(u_6, u_2) = 1$, because u_6 represents the label of the cutpoint represented by u_2.
  - The unary predicate *cp* records the property that a list element is a cutpoint, e.g., $cp^{S_e}(u_2) = 1$ because u_2 represents the (only) cutpoint in S_e; for all other nodes u, $cp^{S_e}(u) = 0$.
  - The unary predicate r_{cp} records the property that a list element is reachable by a cutpoint-anchored path. For example, $r_{cp}^{S_e}(u_2) = 1$ and $r_{cp}^{S_e}(u_3) = 1$ because (only) u_2 and u_3 represent list elements that can be reached from the cutpoint (by the cutpoint-anchored paths $\langle\{\widehat{h.n.n.n}\}, \epsilon\rangle$ and $\langle\{\widehat{h.n.n.n}\}, n\rangle$, respectively). For all other nodes u, $r_{cp}^{S_e}(u) = 0$.

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

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

Figure 16: A specification of the abstract inference rules for atomic statements.

5.1.2 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.

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

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

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

Example. The *3-valued* logical structure $S_e^\sharp$, depicted in Fig. 15 (first row, second column), (conservatively) represents the memory state σ_L^e, represented by S_e.

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

The universe of S_e contains 6 nodes. The only nodes that have the same values for all the unary predicates are u_{1a} and u_{1b}. Thus, the universe of $S_e^\sharp$ contains five nodes. The mapping $f: U^{S_e} \to U^{S_e^\sharp}$ induced by the canonical abstraction is $f(u_0) = u_0^\sharp$, $f(u_{1a}) = f(u_{1b}) = u_1^\sharp$, $f(u_2) = u_2^\sharp$, $f(u_3) = u_3^\sharp$, and $f(u_6) = u_6^\sharp$. The only summary node is $u_1^\sharp$.

We see that any memory state represented by $S_e^\sharp$ contains one cutpoint label (the node $u_6^\sharp$ is not a summary node). The cutpoint is represented by $u_2^\sharp$. This is recorded in two ways: (i) the value of the predicate $lbl^{S_e^\sharp}(u_6^\sharp, u_2^\sharp) = 1$ and (ii) $u_2^\sharp$ represents a list element that is labeled, as indicated by the value of the unary predicate $cp^{S_e^\sharp}(u_2^\sharp) = 1$.

5.2 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. 16 and Fig. 17 shows the specification of the abstract inference rules for atomic statements and function-calls respectively. These rules are declarative in the style of the best abstract transformer [4]: every abstract inference rule emulates a corresponding concrete inference rule using represented states .

Example. Fig. 15 shows an application of the function-call inference rule from Fig. 17 to the running example. The logical structures are: $S_c^\sharp$, which arises at lb_c, the call-site to `reverse`; $S_e^\sharp$,

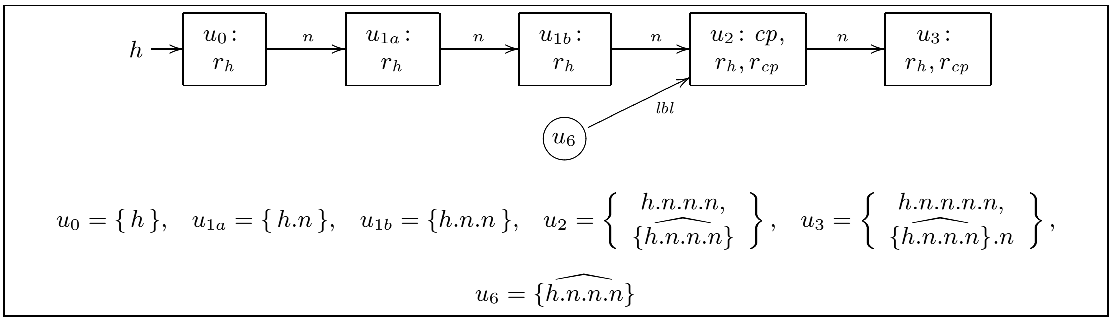


Figure 14: The 2-valued logical structure that results by applying *to2VLS* to σ_L^e, the memory state at the entry point of `reverse` (σ_L^e is shown in Fig. 6). We denote this structure by S_e.

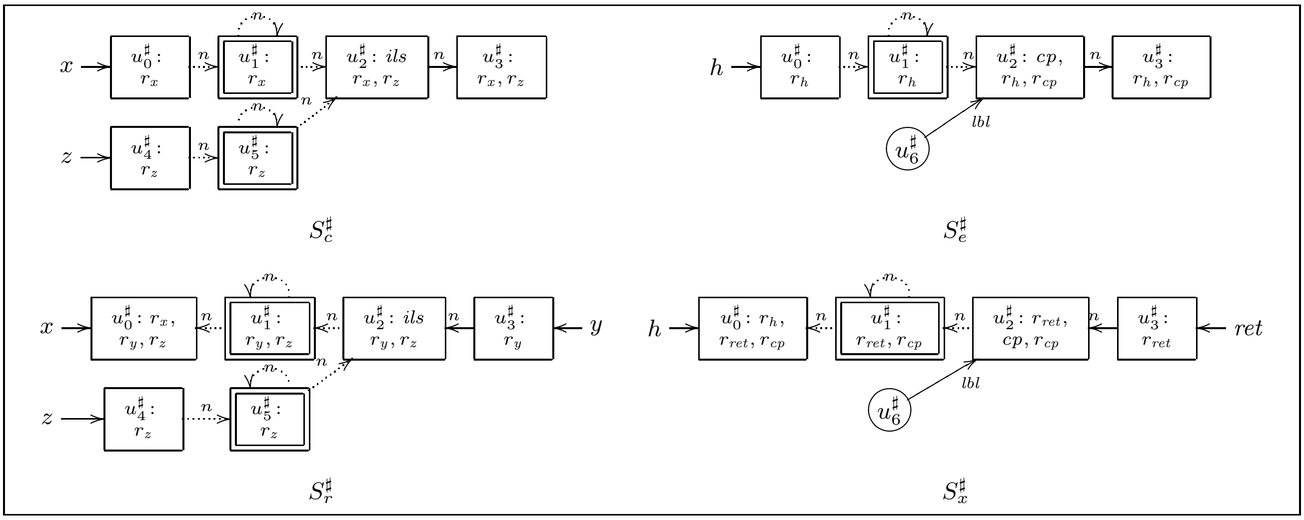


Figure 15: Representative *3-valued* logical structures that arise during the analysis of the running example at *lb*$_c$, the call-site to `reverse` (first row, first column); *lb*$_e$, the entry to `reverse` (first row, second column); *lb*$_x$, `reverse`'s exit point (second row, second column); and *lb*$_r$, the return-site from `reverse` (second row, first column).

which arises lb_e, the entry to `reverse`; $S_x^\sharp$ which arises at lb_x, the exit-point of `reverse`; and $S_r^\sharp$, the structure *computed* at the return-site.

In $S_x^\sharp$, the list pointed-to by `ret` is reversed. As a result, $u_0^\sharp$ 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 $S_x^\sharp$, the inference rule allows us to conclude that at $S_r^\sharp$, the return-site's logical structure, $u_0^\sharp$ becomes reachable from `z`. Similarly, $u_3^\sharp$ is no longer reachable from `z`. To conclude, definite values of many of the tracked properties of `z` can be established after the function call returns.

5.3 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 [9, 18] 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 [9, 18]. For example, we can verify that a recursive function that destructively merges two acyclic lists, returns an acyclic list.

It is straightforward to allow multiple cutpoints for functions with multiple formal arguments by discriminating cutpoints reachable from different formal parameters. This will improve the precision of handling functions that are passed multiple lists.

6. RELATED WORK

Storeless Semantics. Storeless semantics was first introduced by Jonkers [10]. The original work does not handle procedure calls. Intraprocedural storeless semantics is also used in [1] to develop a logic that allows to express regular properties of unbounded data structures.

A storeless semantics that handles function-calls is defined in [6]. The semantics is used to develop a may-alias algorithm. In contrast to $\mathcal{LSL}$, in [6] pending access paths are explicitly represented.

May-Alias Analysis. May-alias algorithms find an upper approximation for the sets of aliased access paths at every program point. Deutsch's interprocedural may-alias algorithm of [7] uses a storeless representation of the heap. The algorithm is polynomial and can handle function calls, dynamic memory allocation and destructive updates.

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

where

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

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

Figure 17: A specification of the abstract inference rules for function calls. The functions $Call_q^{y=p(x_1,\ldots,x_k)}$ and $Ret_q^{y=p(x_1,\ldots,x_k)}$ are defined in Fig. 11. Note that we apply $Ret_q^{y=p(x_1,\ldots,x_k)}$ only for *compatible* pairs of memory states. Memory states σ^c_L and σ^x_L are compatible when the sharing pattern that results from the invocation of p at σ^c_L matches the description of the context in σ^x_L, the state of p at the exit-site. Formally, $compatible(\sigma^c_L, \sigma^x_L) \iff (CPL^e = CPL^x \wedge \forall h, h' \in F_p.[\![h = h']\!]_L(\sigma^e_L) \iff [\![h = h']\!]_L(\sigma^x_L) \wedge \forall h \in F_p.[\![h = \mathtt{null}]\!]_L(\sigma^e_L) \iff [\![h = \mathtt{null}]\!]_L(\sigma^x_L))$, where $\sigma^e_L = Call_q^{y=p(x_1,ldots,x_k)}(\sigma^c_L)$.

$\mathcal{LSL}$ provides insight into Deutsch's work on static may-alias analyses based on pointer-access paths [7]—in particular, the treatment of variables of pending calls, which is one of the most complicated aspects of [7]. For instance, a surprising aspect of the method given in [7] 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. Furthermore, in [17], we show that Deutsch's algorithm can be seen as an abstraction of the $\mathcal{LSL}$ semantics by we defining a Galois connection between memory states in $\mathcal{LSL}$ with the abstract domain of [7].

Interprocedural Shape Analysis. The original motivation for our work comes from our attempt to apply interprocedural shape analysis (e.g., [20]) to heap-manipulating programs in a modular fashion. In [16, 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, [16] does not handle recursive procedures.

Interprocedural shape analysis has been studied in [9,18]. In [18], 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 [9], 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 [2]. 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.

7. CONCLUSIONS

In this paper, we develop $\mathcal{LSL}$, a storeless semantics for languages with dynamic memory allocation, destructive updating and procedure calls. Our storeless semantics is unique in that called procedures are only passed *parts* of the heap.

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 [5, 12] for pointer analysis. We believe we are the first ones to use it in semantics.

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

Using an abstraction of the non-standard concrete semantics, we present a new interprocedural shape-analysis algorithm for programs that manipulate dynamically allocated storage. Our approach is markedly different from previous works that analyze a function invocation in the calling context [9, 18]. 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. 18). 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 [7].

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 [20]) 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 [8, 15], 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.

Acknowledgments. We are grateful for the helpful comments of E. Yahav, G. Yorsh, and the anonymous referees.

8. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


```
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 18: A recursive C procedure that merges two singly linked lists using destructive updates.

(a)

```
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
```

(b)

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

APPENDIX

A. ADDITIONAL CODE

Fig. 18 shows the code for the `merge` function. Fig. 19 shows the code for the functions `crt` and `app` used in the running example.