Region-Based Shape Analysis with Tracked Locations

Brian Hackett and Radu Rugina
Computer Science Department
Cornell University
Ithaca, NY 14853
{bwh6,rugina}@cs.cornell.edu

Abstract

This paper proposes a novel approach to shape analysis: using local reasoning about individual heap locations instead of global reasoning about entire heap abstractions. We present an inter-procedural shape analysis algorithm for languages with destructive updates and formulate it as a dataflow analysis. The key feature is a novel memory abstraction that differs from traditional abstractions in two ways. First, we build the shape abstraction and analysis on top of a pointer analysis. Second, we decompose the shape abstraction into a set of independent configurations, each of which characterizes one single heap location. Our approach: 1) leads to simpler algorithm specifications, because of local reasoning about the single location; 2) leads to efficient algorithms, because of the abstraction decomposition; and 3) makes it easier to develop context-sensitive, demand-driven, and incremental shape analyses.

We have developed simple extensions that use the analysis results to find memory errors in programs with explicit deallocation, including memory leaks and accesses through dangling pointers. We have built a prototype system that implements the ideas in this paper and is designed to analyze C programs. Our experimental results support the intuition that local reasoning leads to more scalable analyses.

1 Introduction

Dynamic data structures are fundamental to virtually all programming languages. To check or enforce the correctness of programs that manipulate such structures, the compiler must automatically extract invariants that describe their *shapes*; for instance, that heap cells are not shared, i.e., not referenced by more than one other memory location. This invariant provides critical information to check high-level properties, for instance that a program builds a tree or an acyclic list; or to check low-level safety properties, for instance that there are no accesses through dangling pointers. For imperative programs with destructive updates, the task of identifying shape invariants is difficult because destructive operations temporarily invalidate them. Examples include even simple operations, such as inserting or removing elements from a list. The challenge is to show that the invariants are restored as the operations finish.

There has been significant research in the area of shape analysis in the past decades, and numerous shape analysis algorithms have been proposed [20]. At the heart of each algorithm stands a sophisticated heap abstraction that captures enough information to show that invariants are being preserved. Examples of heap abstractions include matrices of path expressions and other reachability matrices [10, 9, 6], shape graphs [15, 11], and, more recently, three-valued logic structures [17]; all of these characterize the entire heap at once. However, shape analyses have had limited success at being practical for realistic programs. We believe that their monolithic, heavyweight abstraction is the main reason for their lack of scalability.

This paper presents an inter-procedural shape analysis algorithm based on a novel memory abstraction. The main idea of this paper is to break down the entire shape abstraction into smaller components and analyze those components separately. As shown in Figure 1, we propose a decomposition of the memory abstraction along two directions:

- *Vertical decomposition:* First, we build the fine-grained shape abstraction and analysis on top of a points-to analysis that provides a coarse-grained partition of the memory (both heap and stack) into regions, and identifies points-to relations between regions;

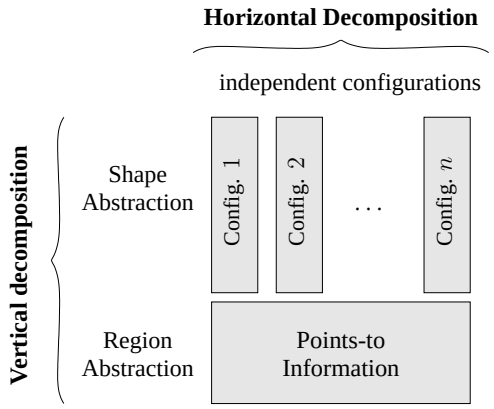

Figure 1: Decomposition of the memory abstraction

- *Horizontal decomposition:* Second, we decompose the shape abstraction itself into individual *configurations*; each configuration characterizes the state of one single heap location, called the *tracked location*. A configuration includes reference counts from each region, and indicates whether certain program expressions reference the tracked location or not. The set of all configurations provides the entire shape abstraction; however, configurations are independent and can be analyzed separately. This is the key property that enables local reasoning.

The vertical decomposition frees the shape analysis of the burden of reasoning about aliases in unrelated program structures. Further, the horizontal decomposition into independent configurations provides a range of advantages. First, it enables local reasoning about the single tracked location, as opposed to global reasoning about the entire heap. This makes the analysis simpler and more modular. Second, the finer level of abstraction granularity reduces the amount of work required by the analysis: efficient worklist algorithms can process individual configurations rather than entire abstractions. Third, it does not require keeping multiple abstractions of the entire heap at each point [16], nor any complex mechanisms to merge entire abstractions for a more compact representation [15]. The decomposition into configurations automatically yields a compact representation that is able to model many concrete heaps. Fourth, it makes it easier to formulate inter-procedural context-sensitive analyses where procedure contexts are individual configurations. Fifth, it makes it easy to build on-demand and incremental shape analysis algorithms. Minor modifications allow the algorithm to explore from only a few selected allocation sites, giving a complete shape abstraction for all cells created at those sites, and to reuse previous results when new allocation sites are being explored.

We also present simple extensions of the analysis for detecting memory errors in languages with explicit deallocation. Our algorithm can identify memory leaks and accesses to deallocated data through dangling pointers. We have built a prototype system for C programs that implements the ideas in this paper and is aimed at detecting memory errors. Our experiments show that local reasoning leads to scalable implementations; that it can correctly model shape for many core list manipulations; and that it can detect errors with low warning rates whenever it correctly models shape.

This paper makes the following contributions:

- **Memory Abstraction:** It proposes a novel memory abstraction that builds the fine-grained shape abstraction on top of a coarse-grained region abstraction; it further decomposes the shape abstraction itself into independent configurations that describe individual heap locations;
- **Analysis Algorithm and Applications:** It gives a precise specification of an inter-procedural, context-sensitive analysis algorithm for this abstraction; and shows how to use the analysis results to detect memory errors;
- **On-demand and Incremental Shape Analysis:** It shows that our approach can be applied to the demand-driven and incremental computation of shapes;
- **Theoretical Framework:** It presents the analysis algorithm in a formal setting and shows that the key parts of the algorithm are sound;

```
 1:  typedef struct list {
 2:     struct list *n;
 3:     int data;
 4:  } List;
 5:
 6:  List *splice(List *x, List *y) {
 7:    List *t = NULL;
 8:    List *h = y;
 9:    while(x != NULL) {
10:      t = x;
11:      x = t->n;
12:      t->n = y->n;
13:      y->n = t;
14:      y = y->n->n;
15:    }
16:    return h;
17:  }
```

Figure 2: Example program: splicing lists

- **Experimental Results:** It presents experimental results collected from a prototype implementation of the proposed analysis for C programs.

The rest of the paper is organized as follows. Section 2 presents an example. Section 6 discusses limitations. Next, Section 3 gives a simple source language, and Sections 4, 5, and 5.4 describe the analysis algorithm and extensions. We present experimental results in Section 8, discuss related work in Section 9, and conclude in Section 10.

2 Example

We use the example from Figure 2 to illustrate the key features of our analysis. This example is written in C and shows a procedure `splice` that takes two lists `x` and `y` as arguments, splices `x` into `y`, and returns the resulting list. The goal of shape analysis is to statically verify that, if the input lists `x` and `y` are disjoint and acyclic, then the list returned by `splice` is acyclic.

The execution of splice works as follows. First, it stores a pointer to the second list into a local variable `h`. Then, the program uses a loop to traverse the two lists with parameters `x` and `y`. At each iteration, it sets a pointer from the first list into the second, and vice-versa, using a temporary variable `t`. When the traversal reaches the end of one of the lists, it terminates and the procedure returns the list pointed to by `h`.

2.1 Memory Abstraction

Figure 3 shows two possible concrete stores that can occur during the execution of `splice`. The one on the left is a possible concrete store at the beginning of the procedure, where `x` and `y` point to acyclic lists; and the store on the right is the corresponding memory state when the loop terminates. Boxes labeled with `x`, `y`, `t`, and `h` represent the memory locations of those variables. The remaining unlabeled boxes are heap cells and represent list elements.

Figure 4 presents the memory abstraction that our analysis uses for these two concrete stores. The right part of this figure shows the region component of the abstraction, which consists of a points-to graph of regions. Each region models a set of memory locations; and different regions model disjoint sets of locations. For this program, our analysis uses a separate region to model the location of each variable[1]: X, Y, T, and H are the regions containing variables `x`, `y`, `t`, and `h`, respectively. Region L models all list elements. The points-to graph Figure 4 shows the points-to relations between abstract regions for the whole procedure, as given by a flow-insensitive pointer analysis. Hence, this graph applies to both the input and output abstractions discussed here.

[1] Although not the case in this example, it may happen that multiple variables get placed into the same region.

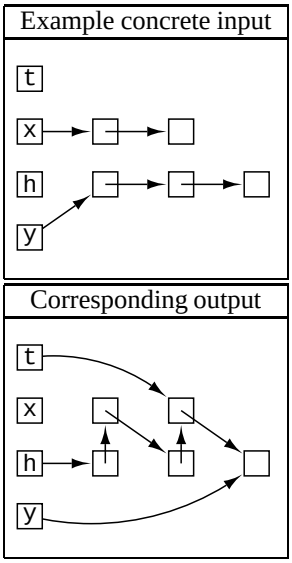

Figure 3: Example concrete memories

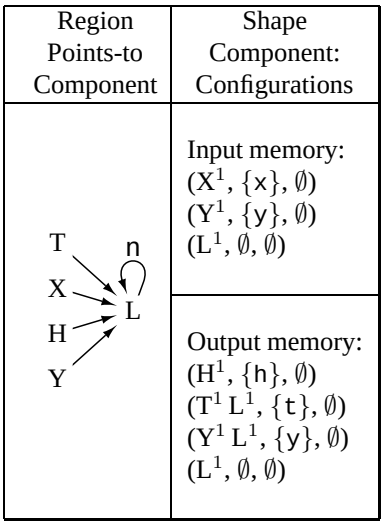

Figure 4: Memory abstraction

The right part of Figure 4 shows the shape component for each of the concrete stores. Each abstraction consists of a set of configurations: the input abstraction has 3 configurations and the output has 4. Each configuration characterizes the state of the tracked location and consists of three parts: reference counts from each region to the tracked location (shown in superscripts); program expressions that definitely reference the tracked location (*hit* expressions); and program expressions that definitely do not (*miss* expressions). For instance, configuration ($T^1\,L^1$, {t}, $\emptyset$) shows that the tracked location has one incoming reference from region T, one incoming reference from region L, and is referenced by expression `t`, but that any expression originating from L (i.e., next pointers) may either reference it or fail to reference it. Although we could have used richer sets of miss expressions, these abstractions are sufficient for our algorithm to prove the shape property.

These abstractions are complete: the set of all configurations in each abstraction provides a characterization of the entire heap. Indeed, if the tracked location is any of the five heap cells, there is a configuration that characterizes it. But although their sum collectively describes the entire heap, configurations are independent: the state described by any particular configuration is not related to the other configurations; it characterizes one heap location and has no knowledge about the state of the rest of the heap (beyond what is given by the points-to graph). This is the key property that enables local reasoning.

2.2 Analysis of `splice`

Figure 5 shows the analysis result that our algorithm computes at each point in the program. This shape abstraction builds on the region points-to abstraction from the previous section. Boxes in the figure represent individual configurations; each row represents the entire heap abstraction at a program point; and edges correlate the state of the tracked location before and after each statement. Therefore, each path shows how the state of the tracked location changes during program execution. For clarity, we omit wrap-around edges that connect configurations from the end to the beginning of the loop. Also, we omit individual variables from hit and miss sets, and show just the field accesses expressions. We use the abbreviations: tn $\equiv$ `t->n` and yn $\equiv$ `y->n`, and indicate miss expressions using overlines. Our algorithm efficiently computes this result using a worklist algorithm that processes individual configurations (i.e., individual nodes), rather full heap abstractions (i.e., entire rows).

The top row shows three configurations, Y^1, L^1, and X^1, that describe the memory for any input where `x` and `y` point to acyclic lists. The bottom row consists of configurations H^1 and L^1, and shows that at the end of the procedure the returned value `h` points to an acyclic list. Hence, the analysis successfully verifies the desired shape property.

We discuss the analysis of several configurations to illustrate how the analysis performs local reasoning. Consider

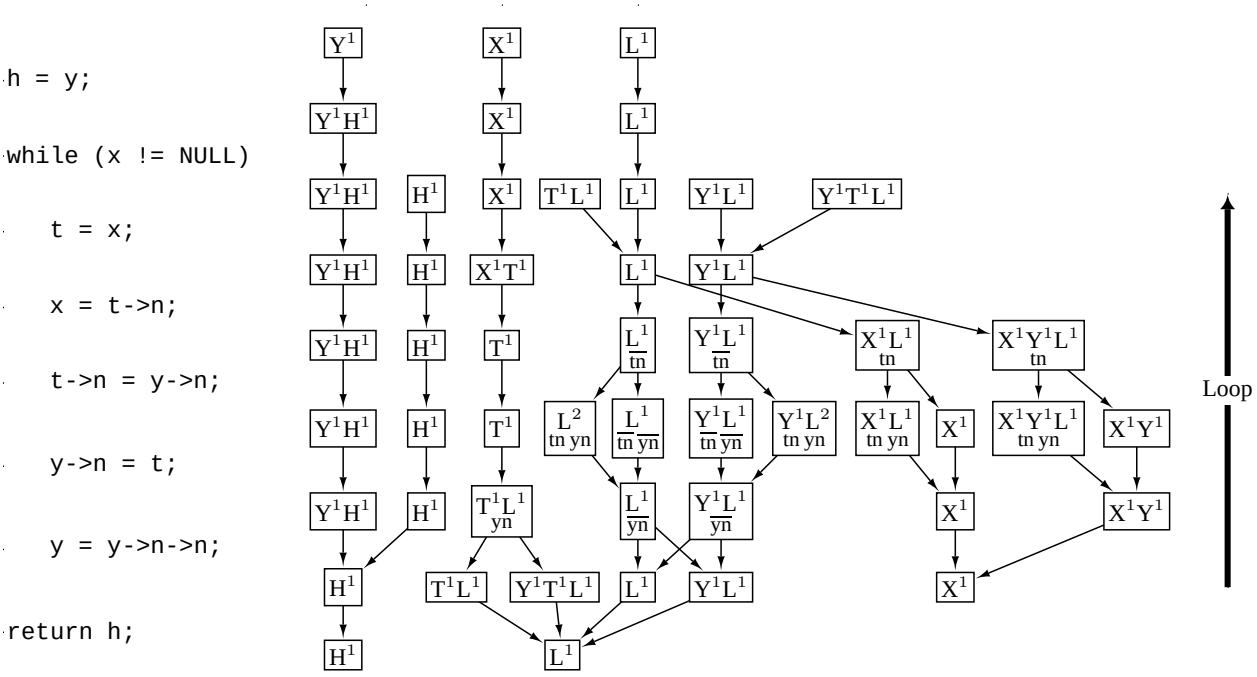

Figure 5: Shape analysis results for `splice`. Boxes represent configurations and edges show how the state of the tracked location changes during the execution. Only field expressions in the hit and miss sets are shown. We use the abbreviations: tn $\equiv$ `t->n` and yn $\equiv$ `y->n`, and we indicate miss expressions using overlines.

the configuration Y^1H^1 before the statement at line 11, `x=t->n`. The compiler inspects this assignment and tries to determine whether or not the expressions in the left- and right-hand side reference the tracked location: if the right side references the location, the assignment may add a new reference to it; and if the left side points to the tracked location, the assignment may remove a reference to it. For `x=t->n`, the analysis determines that `x` represents a location in region X, and `t->n` is a location in region L, as indicated by the points-to graph. But the reference counts in the current configuration show that the tracked location has no references from regions X or L; hence, it concludes that neither `x`, nor `t->n` reference the tracked location and this assignment doesn't affect its reference counts.

Consider now the configuration L^1 at the same program point, before line 11. The compiler can use the same judgment as above to determine that `x` does not reference the tracked location. However, it is not able to determine whether or not `t->n` references it. At this point, the compiler *bifurcates* the current configuration into two configurations where this fact is precisely known: one where `t->n` references the location and one where it doesn't. In each case, it adds `t->n` to the corresponding hit or miss set, and analyzes the assignment. The resulting two configurations L^1 and X^1L^1 after line 11 are the successors of the analyzed configuration L^1.

Keeping track of hit and miss sets provides invaluable information to the analysis. Consider configuration L^1 before statement `t->n=y->n`. The analysis of this statement yields a configuration L^2, where the tracked location has two incoming references from L and violates the desired shape property. However, the analysis identifies that the reference `y->n` is being copied, so it adds `y->n` to the hit set of configuration L^2. At the next assignment, `y->n=t`, the analysis identifies that the same expression `y->n` is being overwritten. The presence of `y->n` in the hit set enables the analysis to accurately decrease the reference count from L back to 1.

2.3 Cyclic and Shared Inputs

The analysis can describe the behavior of `splice` even for inputs with cyclic or shared lists, and can do so efficiently. Figure 6 shows a concrete input store where the list pointed to by `y` contains a cycle; the right part shows the resulting memory after executing `splice`. A closer look at the heap structure reveals that the cycle is caused by the presence

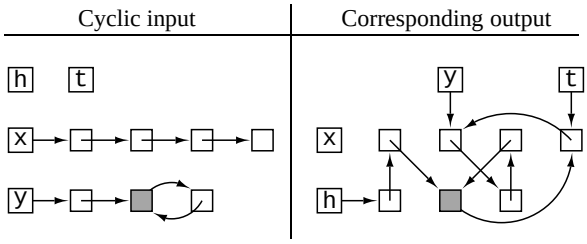

Figure 6: Example cyclic memory

of one shared cell, with two incoming references – the shaded cell. To build the abstraction for this input, we can use the previous abstraction, and augment it with one additional configuration L^2 that describes the cell in question. The analysis of L^2 will yield a set of output configurations at the end, including several configurations with a reference count of 2 from L. Therefore, the analysis identifies that the shared cell in the input may remain shared after the procedure, but that all of the non-shared cells in the input will remain non-shared in the output.

The analysis can compute this fact efficiently, reusing the results from the acyclic case. This is possible because of local reasoning: the analysis of each configuration in Figure 5 reasons only about the tracked location, and makes no assumption about the presence or absence of cycles in the rest of the structure. Hence, those results directly apply to cyclic inputs – they characterize the "acyclic portion" of the cyclic input. This is also the case for inputs with shared sublists, or inputs where one list is a sublist of the other.

This discussion brings us to the inter-procedural analysis, the main obstacle in building scalable analyses. The example shows that our abstraction allows us to efficiently build an inter-procedural context-sensitive analysis, where the analysis of each calling context can reuse results from other contexts. In our example, the analysis of `splice` for a context with cyclic lists can reuse the result from an acyclic context, and do little additional work. And if the cyclic context is being analyzed first, the result for the acyclic one is already available. This is possible because we can break down the entire heap context into finer-grain contexts that are just individual configurations.

3 Source Language and Concrete Stores

To formalize the description of the algorithm, we use the simple language shown in Figure 7. This is a typeless C-like imperative language with procedures, dynamic allocation, and explicit deallocation. A program maps procedures to their formal parameters and their bodies. The only possible values are pointers to memory locations and null pointers. Dynamic allocations create structures that contain one memory location for each field. There is a distinguished first field f_1 in each structure; dynamic allocations return a pointer to the first field in the newly allocated structure. The language supports pointers to variables and pointers into the middle of structures. An expression $e.f$ requires e to be an l-value representing the first field of a structure; then $e.f$ is the f field of that structure. A dereference expression $*e$ always represents the memory location pointed to by e (not the whole structure when e points to a structure). We represent C expressions `e->f` as `(*e).f`. Deallocation statements free all of the locations in a structure and yield dangling pointers.

We formally model concrete stores that can arise during program execution using a set of memory locations L that contains locations for all variables, as well as all of the heap locations created during the execution of the program. A concrete store is a triple $\sigma = (\sigma_v, \sigma_l, \sigma_f)$, consisting of:

$$\begin{array}{lrcl} \text{variable map:} & \sigma_v & : & V \to L \\ \text{location map:} & \sigma_l & : & L \rightharpoonup (L + \{\mathit{null}\}) \\ \text{field map:} & \sigma_f & : & (L \times F) \rightharpoonup L \end{array}$$

The map σ_v assigns a location to each variable. The partial map σ_l models points-to relations between locations. We require that $\mathit{range}(\sigma_v) \subseteq \mathit{dom}(\sigma_l)$. The set $L_\sigma = \mathit{dom}(\sigma_l) - \mathit{range}(\sigma_v)$ represents the currently allocated heap locations. Variables $x \in V$ such that $\sigma_l(\sigma_v(x)) \notin \mathit{dom}(\sigma_l)$ hold dangling pointers, and so do locations $l \in L_\sigma$

$$
\begin{array}{lll}
\text{programs:} & \mathit{prog} \in \mathit{Prog} \;=\; P \to (V^n \times S) \\
\text{procedures:} & p \in P \\
\text{statements:} & s \in S, \quad s ::= \; e_0 \leftarrow e_1 \\
 & \qquad\qquad\qquad \mid \; e \leftarrow \textbf{malloc} \;\mid\; \textbf{free}(e) \\
 & \qquad\qquad\qquad \mid \; \textbf{call}\; p(e_1, .., e_n) \;\mid\; s_0 \,;\, s_1 \\
 & \qquad\qquad\qquad \mid \; \textbf{if}\; (e)\; s_0\; \textbf{else}\; s_1 \;\mid\; \textbf{while}\; (e)\; s \\
\text{expressions:} & e \in E, \quad e ::= \; \textbf{null} \;\mid\; x \;\mid\; \&e \;\mid\; *\, e \;\mid\; e.f \\
\text{variables:} & x \in V \\
\text{fields:} & f \in F \;=\; \{f_1, \ldots, f_m\}
\end{array}
$$

Figure 7: Source language

with $\sigma_l(l) \notin \mathit{dom}(\sigma_l)$. Finally, the partial map σ_f captures the organization of fields in allocated structures. For a location $l \in L$ that represents the first field in a heap structure, $\sigma_l(l, f)$ represents the f field of that structure. We require that all locations in the domain and range of σ_f be allocated. We use the notations: $\mathit{dom}_v(\sigma) = \mathit{dom}(\sigma_v)$, $\mathit{dom}_l(\sigma) = \mathit{dom}(\sigma_l)$, and $\mathit{dom}_f(\sigma) = \mathit{dom}(\sigma_f)$. In this paper, we refer to the l-value of an expression e as the *location of* e; and to the r-value e as the *value of* e.

4 Region Analysis

The goal of region analysis is to provide a partitioning of the memory into disjoint regions and to identify points-to relations between regions. We use an existing unification-based flow-insensitive, but context-sensitive pointer analysis algorithm [14]. This algorithm computes a points-to graph for each procedure; nodes in these graphs represent memory regions and edges model points-to relations between regions. Most important for this paper is to characterize the computed points-to result, thus describing the interface between region and shape analysis. We formalize the analysis result first and then briefly describe the algorithm. Given a program, a region abstraction consists of the following:

- for each procedure p, a set of regions R^p that models the locations that p may access.
- for each procedure f, a region abstract store: $\rho^p = (\rho^p_v, \rho^p_r, \rho^p_f)$, where $\rho^p_v : V \to R^p$ maps variables to their regions; $\rho^p_r : R^p \to (R^p + r_{null})$ maps regions to target regions, or r_{null} for regions that contain only null pointers; and $\rho^p_f : (R^p \times F) \to R^p$ maps pairs of base regions and fields to field regions;
- for each call site cs, with caller p and callee q, a one-to-one mapping $\mu_{cs} : R^q \to R^p$ that maps all (parameter) regions in q to actual regions in p.

A region abstraction is sound if regions describe disjoint sets of memory locations, and points-to relations in the abstraction accurately describe points-to relations in the concrete heap. We give a formal definition of soundness in Section 7. Key to the algorithm is that regions are disjoint, so the subsequent shape analysis can safely conclude that an update in one region will not change the values of locations in other regions.

We briefly sketch the algorithm that computes the region points-to abstraction. First, the algorithm performs an intra-procedural, unification-based analysis to build a points-to graph for each function, as proposed by Steensgaard [18]. Then, it performs an inter-procedural analysis and propagates the aliasing information between different functions at each call site, using a two-phase algorithm: a first phase processes functions in the the call tree in a bottom-up fashion, propagating the aliasing information from callees to callers; after that, a top-down phase propagates the information from callers to callees.

5 Shape Analysis

This section presents the shape analysis algorithm in detail. We first present the shape abstraction and then give the intra- and inter-procedural algorithms.

5.1 Shape Abstraction

The shape abstraction uses configurations to model the referencing relations between memory locations at a finer level than the region abstraction. Each configuration abstracts the state of one individual memory location, the tracked location. The full heap abstraction consists of a finite set of configurations, such that each concrete memory location can be modeled by one configuration in the set.

A configuration counts references to the tracked location from each region; furthermore, it keeps track of expressions that definitely reference the tracked location (the hit expressions), as well as expressions that definitely don't reference the location (the miss expressions). Formally, we use an index domain I for counting references, and a secondary domain H for hit and miss expressions. A configuration is then a pair of an index and a secondary value. If R is the set of regions in the currently analyzed function and E_p is the finite set of program expressions, then the domains are:

$$\begin{array}{lrcl} \text{Index values:} & i \in I & = & R \to \{0, \ldots, k, \infty\} \\ \text{Secondary values:} & h \in H & = & \mathcal{P}(E_p) \times \mathcal{P}(E_p) \\ \text{Configurations:} & c \in C & = & I \times H \end{array}$$

Each index value i gives the reference counts for each region. We bound the reference counts to a fixed value k, to ensure that the abstraction is finite. For each region $r \in \mathit{dom}(i)$, the number $i(r)$ is the number of references to the tracked location from region r: if $i(r) \in 0..k$, then the reference count is exactly $i(r)$; otherwise, if $i(r) = \infty$, the reference count is $k+1$ or greater. In practice, we found a low value $k = 2$ to be precise enough for all of the programs that we experimented with. We emphasize that k is the maximum reference count from each region; however, there can be many more references to the tracked object, as long as they come from different regions. Finally, each secondary value $h \in H$ is a pair $h = (e^+, e^-)$, where e^+ is the hit set and e^- is the miss set.

The full shape abstraction consists of a set of configurations, with at most one configuration for each index value. In other words, the abstraction is a partial map from index values to secondary values. We represent it as a total function that maps the undefined indices to a bottom value $\perp$:

$$\text{Shape abstraction:} \quad a \in A \quad = \quad I \to (H + \perp)$$

We define a lattice domain over the abstract domain, as follows. The bottom element is $a_{\perp} = \lambda i.\perp$, meaning that no configuration is possible. The top element is $a_{\top} = \lambda i.(\emptyset, \emptyset)$, meaning that any index is feasible and, for each index, any expression can either reference or fail to reference the tracked location. Given $a_1, a_2 \in A$, their join $a_1 \sqcup a_2$ is:

$$(a_1 \sqcup a_2)(i) = \begin{cases} a_1(i) & \text{if } i \notin \mathit{dom}(a_2) \\ a_2(i) & \text{if } i \notin \mathit{dom}(a_1) \\ a_1(i) \sqcup a_2(i) & \text{if } i \in \mathit{dom}(a_1) \cap \mathit{dom}(a_2) \end{cases}$$

$$\begin{array}{ll} \text{where} & (e_1^+, e_1^-) \sqcup (e_2^+, e_2^-) = (e_1^+ \cap e_2^+, e_1^- \cap e_2^-) \\ \text{and} & \perp \sqcup (e^+, e^-) = (e^+, e^-) \sqcup \perp = (e^+, e^-) \end{array}$$

The merge operator $\sqcup$ is overloaded and applies to both A and $H + \perp$; the operator being used can be inferred from the context where it occurs. We denote by $\sqsubseteq$ the partial order that corresponds to $\sqcup$.

5.2 Intra-Procedural Analysis

We present the dataflow equations, the transfer functions for assignments, malloc, and free, and then give formal results.

For all $s \in S_{asgn},\ s_a \in S_{alloc},\ s_e \in S_{entry},\ i \in I$:

[JOIN] $\quad \mathit{Res}(\bullet s)\, i = \bigsqcup_{s' \in pred(s)} \mathit{Res}(s'\bullet)\, i$

[TRANSF] $\quad \mathit{Res}(s\bullet)\, i = \bigsqcup_{i' \in I} (\llbracket s \rrbracket(\rho, (i', \mathit{Res}(\bullet s)\, i')))\, i$

[ALLOC] $\quad \mathit{Res}(s_a\bullet)\, i_a \sqsupseteq h_a,$ where $\llbracket s_a \rrbracket^{gen}(\rho) = (i_a, h_a)$

[ENTRY] $\quad \mathit{Res}(\bullet s_e)\, i \sqsupseteq a_o\, i$

Figure 8: Intra-procedural dataflow equations.

5.2.1 Dataflow Equations

We formulate the analysis of each function in the program as a dataflow analysis which computes a shape abstraction $a \in A$ at each program point in that function. However, our algorithm differs from standard approaches in two ways. First, it uses a system of dataflow equations and a corresponding worklist algorithm that takes advantage of the fact that configurations are independent and can be analyzed separately. Second, the dataflow information is being initialized not only at the entry point in the control-flow, but also at each allocation site, as the analysis must produce a new configuration for each new location.

Let S_{asgn} be the set of assignments in the program, and $S_{alloc} \subseteq S_{asgn}$ the set of allocation assignments. For each assignment $s \in S_{asgn}$, we define two program points: $\bullet s$ is the program point before s and $s\bullet$ is the program point after s. Let $S_{entry} \subseteq S_{asgn}$ be the set of assignments that occur at the entry of the control flow (that is, the beginning of the currently analyzed function) and let *pred* and *succ* map assignment statements to their predecessor or successor assignments in the control flow (these can be easily computed from the syntactic structure of control statements).

We model the analysis of individual assignments using transfer functions that operate at the granularity of individual configurations. The transfer function $\llbracket s \rrbracket$ of a statement $s \in S_{asgn}$ takes the current region abstract store ρ and a configuration $c \in C$ before the statement to produce the set of possible configurations after the statement: $\llbracket s \rrbracket(\rho, c) \in A$. Furthermore, for each allocation $s \in S_{alloc}$, there is a new configuration $\llbracket s \rrbracket^{gen}(\rho) \in C$ being generated.

The result of the analysis is a function *Res* that maps each program point to the shape abstraction at that point. If ρ is the abstract region store for the currently analyzed function and $a_0 \in A$ is the boundary dataflow information, then the result *Res* is the least fixed point of the system of dataflow equations shown in Figure 8. Equations [JOIN], [TRANSF], and [ENTRY] are standard dataflow equations, but are being expressed such that they expose individual configurations and their dependencies. The special equation [ALLOC] indicates that the analysis creates a configuration for the new location after the statement, regardless of the abstraction before the allocation site.

This formulation allows us to build an efficient worklist algorithm for solving the dataflow equations. Instead of computing transfer functions for entire heap abstractions when the information at a program point has changed, we only need to recompute it for those indices whose secondary values have changed. Rather than being entire program statements, worklist elements are statements paired with indices. Using a worklist with this finer level of granularity serves to decrease the amount of work required to find the least fixed point.

The precise worklist algorithm is shown in Figure 9. Lines 1-14 perform the initialization: they set the value of *Res* at entry points (lines 4-7) and at allocation sites (lines 8-14), and initialize it to $a_\perp$ at all other program points (lines 2-3). The algorithm also initializes the worklist, at lines 1, 7, and 14. Then, it processes the worklist using the loop between lines 16-22. At each iteration, it removes a statement and an index from the worklist, and applies the transfer function of the statement for that particular index. Finally, the algorithm updates the information for all successors, but only for the indices whose secondary values have changed (lines 19-21). Then, it adds the corresponding pair of successor statement and index value to the worklist, at line 22.

```
WORKLIST(DataflowInfo a_0)
 1  W = ∅
 2  for each s ∈ S_asgn
 3      Res(•s) = Res(s•) = a_⊥
 4  for each s_e ∈ S_entry
 5      Res(•s_e) ⊔= a_0
 6      for each i such that Res(•s_e)i has changed
 7          W = W ∪ {(s, i)}
 8  for each s_a ∈ S_alloc
 9      let (i_a, h_a) = [[s_a]]^gen(ρ)
10      Res(s_a•)i_a ⊔= h_a
11      for each i such that Res(s_a•)i_a has changed
12          for each s ∈ succ(s_a)
13              Res(•s)i ⊔= Res(s_a•)i
14              W = W ∪ {(s, i)}
15
16  while (W is not empty)
17      remove some (s, i) from W
18      Res(s•) ⊔= [[s]](ρ, (i, Res(•s)i))
19      for each i' such that Res(s•)i' has changed
20          for each s' ∈ succ(s)
21              Res(•s')i' ⊔= Res(s•)i'
22              W = W ∪ {(s', i')}
```

Figure 9: Intra-procedural worklist algorithm.

5.2.2 Decision and Stability Functions

To simplify the formal definition of transfer functions, we introduce several evaluation functions for expressions. First, we use a *location evaluation function* $\mathcal{L}[\![e]\!]$ that evaluates an expression e to the region that holds the memory location that e represents. The function is not defined for expressions that cannot evaluate to a location ($\&e$ and **null**):

$$\begin{array}{rcl} \mathcal{L}[\![x]\!]\rho & = & \rho(x) \\ \mathcal{L}[\![*e]\!]\rho & = & \rho(\mathcal{L}[\![e]\!]\rho) \\ \mathcal{L}[\![e.f]\!]\rho & = & \rho(\mathcal{L}[\![e]\!]\rho, f) \end{array}$$

Second, we define a *decision evaluation function* $\mathcal{D}[\![e]\!]$ that takes a configuration and inspects the information in the index and in the secondary value to determine whether e references the location that c tracks: $\mathcal{D}[\![e]\!](\rho, c) \in \{+, -, ?\}$. The evaluation returns "$+$" is e references the tracked location, "$-$" if it doesn't, and "?" if there is insufficient information to make a decision:

$$\mathcal{D}[\![e]\!](\rho, (i, (e^+, e^-))) = \begin{cases} - & \text{if } e \in e^- \ \vee \ i(\mathcal{L}[\![e]\!]\rho) = 0 \\ & \quad \vee \ e = \textbf{null} \ \vee \ e = \&x \\ + & \text{if } e \in e^+ \\ ? & \text{otherwise} \end{cases}$$

(the condition $i(\mathcal{L}[\![e]\!]\rho) = 0$ in the miss case is a shorthand for "$e \neq \textbf{null} \ \wedge \ e \neq \&e' \ \wedge \ i(\mathcal{L}[\![e]\!]\rho) = 0$")

Finally, we define two *stability evaluation functions* to determine if writing into a region affects the location or the value of an expression. The location stability function $\mathcal{S}[\![e]\!]_l$ takes an abstract store and a region, and yields a boolean

that indicates whether the location of e is stable with respect to updates in that region: $\mathcal{S}[\![e]\!]_l(\rho, r) \in \{\text{true}, \text{false}\}$:

$$\begin{array}{rcl} \mathcal{S}[\![x]\!]_l(\rho, r) & = & \text{true} \\ \mathcal{S}[\![*e]\!]_l(\rho, r) & = & \mathcal{S}[\![e]\!]_l(\rho, r) \ \wedge \ \mathcal{L}[\![e]\!]\rho \neq r \\ \mathcal{S}[\![e.f]\!]_l(\rho, r) & = & \mathcal{S}[\![e]\!]_l(\rho, r) \end{array}$$

The value stability function $\mathcal{S}[\![e]\!]_v$ indicates whether the value of e is stable with respect to updates in r. Value stability implies location stability, but not vice-versa:

$$\begin{array}{rcl} \mathcal{S}[\![\textbf{null}]\!]_v(\rho, r) & = & \text{true} \\ \mathcal{S}[\![x]\!]_v(\rho, r) & = & \rho(x) \neq r \\ \mathcal{S}[\![\&e]\!]_v(\rho, r) & = & \mathcal{S}[\![e]\!]_l(\rho, r) \\ \mathcal{S}[\![*e]\!]_v(\rho, r) & = & \mathcal{S}[\![e]\!]_v(\rho, r) \ \wedge \ \mathcal{L}[\![*e]\!]\rho \neq r \\ \mathcal{S}[\![e.f]\!]_v(\rho, r) & = & \mathcal{S}[\![e]\!]_l(\rho, r) \ \wedge \ \mathcal{L}[\![e.f]\!]\rho \neq r \end{array}$$

Property 1 (Stability) *Given a sound abstract store ρ, an assignment $e_0 \leftarrow e_1$ such that $\mathcal{L}[\![e_0]\!](\rho) = r$, a concrete state σ before the assignment and a concrete state σ' after the assignment, then:*

- *for any expression e, if $\mathcal{S}[\![e]\!]_v(\rho, r)$ then e evaluates to the same value in stores σ and σ';*
- *if $\mathcal{S}[\![e_0]\!]_l(\rho, r)$ then e_0 evaluates in store σ' to the value that e_1 evaluates in store σ.*
- *if $\mathcal{S}[\![e_1]\!]_l(\rho, r)$ then e_1 evaluates to the same value in stores σ and σ';*

We illustrate the importance of stability with an example. Consider two variables: x in region r_x and y in region r_y. Assume that the tracked location is being referenced by y, that the tracked location does not have a self reference, and that the program executes the statements $x = \&x$; $*x = y$. Although we assign y to $*x$ and y is a hit expression, $*x$ will not hit the tracked location after this code fragment. The reason is that $*x$ does not represent the same memory location before and after the statement; the update has caused $*x$ to have different l-values. Our analysis captures this fact by identifying that expression $*x$ is not location-stable with respect to updates in region r_x. In general, if the left-hand side of an assignment is not location-stable, it is not safe to add it to the hit (or miss) set, even if the right-hand side hits (or misses) the tracked location.

5.2.3 Analysis of Assignments: $e_0 \leftarrow e_1$

The analysis of assignments plays the central role in the intra-procedural analysis. Given a configuration $c = (i, h)$ before the assignment, the goal is to compute all of the resulting configurations after the assignment. Given our language syntax, this form of assignment models many particular cases, such as nullifications ($x = \textbf{null}$), copy assignments ($x = y$), load assignments ($x = *y$ or $x = y.d$), store assignments ($*x = y$ or $x.d = y$), address-of assignments ($x = \&y$), as well as assignments that involve more complex expressions. Our formulation compactly expresses the analysis of all these statements in a single, unified framework.

Figure 10 shows the algorithm. The analysis must determine whether or not the expressions e_0 and e_1 reference the tracked location. For this, it invokes the decision function $\mathcal{D}[\![\cdot]\!]$ on e_0 and e_1. For each of the two expressions, if the decision function cannot precisely determine if they hit or miss the tracked location, it bifurcates the analysis in two directions: one where the expression definitely references the tracked location, and one where it definitely doesn't. The algorithm adds e_0 and e_1 to the corresponding hit or miss set and analyzes each case using the auxiliary function *assign*; it then merges the outcomes of these cases.

The function *assign* is shown Figure 11 and represents the core of the algorithm. It computes the result configurations when the referencing relations of e_0 and e_1 to the tracked location are precisely known, and are given by the boolean parameters b_0 and b_1, respectively. The algorithm works as follows. First, it evaluates the region r that holds the location being updated, using $\mathcal{L}[\![e_0]\!]$. Then, it updates the reference count from r, between lines 2-8. If e_0 references the location, but e_1 doesn't, it decrements the count; if e_1 references the location, but e_0 doesn't, it increments it; and if none or both reference it, the count remains unchanged. Special care must be taken to handle infinite reference counts; in particular, decrementing infinite counts yields two possible values, ∞ and k. The result is a set S_i that contains either one or two indices with the updated reference count(s) for r. Note that the analysis can safely preserve reference counts from all regions other than r, because regions model disjoint sets of memory locations.

$[\![e_o \leftarrow e_1]\!](\rho, (i, (e^+, e^-))) :$

case $(\mathcal{D}[\![e_0]\!](\rho, (i, (e^+, e^-))), \mathcal{D}[\![e_1]\!](\rho, (i, (e^+, e^-))))$ **of**

$(v_0 \in \{-, +\},\ v_1 \in \{-, +\}) \Rightarrow$
  assign$(e_0, e_1, \rho, i, e^+, e^-, v_0 = +, v_1 = +)$

$(?,\ v_1 \in \{+, -\}) \Rightarrow$
  assign$(e_0, e_1, \rho, i, e^+ \cup \{e_0\}, e^-, \text{true}, v_1 = +) \sqcup$
  assign$(e_0, e_1, \rho, i, e^+, e^- \cup \{e_0\}, \text{false}, v_1 = +)$

$(v_0 \in \{-, +\},\ ?) \Rightarrow$
  assign$(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^-, v_0 = +, \text{true}) \sqcup$
  assign$(e_0, e_1, \rho, i, e^+, e^- \cup \{e_1\}, v_0 = +, \text{false})$

$(?,\ ?) \Rightarrow$
  assign$(e_0, e_1, \rho, i, e^+ \cup \{e_0, e_1\}, e^-, \text{true}, \text{true}) \sqcup$
  assign$(e_0, e_1, \rho, i, e^+ \cup \{e_0\}, e^- \cup \{e_1\}, \text{true}, \text{false}) \sqcup$
  assign$(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^- \cup \{e_0\}, \text{false}, \text{true}) \sqcup$
  assign$(e_0, e_1, \rho, i, e^+, e^- \cup \{e_0, e_1\}, \text{false}, \text{false})$

Figure 10: Transfer function for assignments $[\![e_0 \leftarrow e_1]\!]$.

Next, the analysis derives new hit and miss sets, using the computation between lines 10-20. First, at lines 10 and 11, the analysis filters out expressions whose referencing relations to the tracked location no longer hold after the update. For instance, the filtered set e_n^- includes from e^- only those expressions e that meet one of the following two conditions:

- $\mathcal{S}[\![e]\!]_v(\rho, r)$: the *value* of e is stable with respect to the updated region r. In that case, e has the same value before and after the assignment, so it remains in e^-;
- $\mathcal{S}[\![e]\!]_l(\rho, r) \wedge \neg b_1$: the *location* of e is stable with respect to r and the assigned value misses the tracked location (i.e., b_1 = false). Hence, e represents the same location before and after the assignment, but its value may or may not change. If the value doesn't change, e will not reference the tracked location after the assignment because it didn't before ($e \in e^-$). If the value changes, the location gets overwritten with a value that still doesn't reference the tracked location (as indicated by b_1 = false). Hence, the analysis can conclude that in either case e will not reference the tracked location and can safely keep e in e_n^-.

At lines 13 and 14, the analysis tries to add the left-hand side expression e_0 to the hit or miss set, to capture its new value. For instance, if b_1 = true, the assigned value hits the tracked location, and the program writes that value into the location of e_0. If e_0 is location-stable, then e_0 will evaluate to the value of e_1 before the statement, and will therefore reference the tracked location after the assignment. Hence, the analysis can safely add it to e_n^+. The analysis uses a similar reasoning when b_1 = false.

At lines 16-18, the analysis derives new expressions in e_n^+ and e_n^- by substituting occurrences of $*e_1$ with $*e_0$ in expressions that do not contain address-of subexpressions. The set E_p' in this figure represents all program expression that don't contain the address-of operator. Once again, the substitutions are safe only when certain stability conditions are met, in this case that e_0 and e_1 are both location-stable.

At line 20, the analysis discards from the miss set all those expressions whose referencing relations can be inferred by the decision function using region information alone. This allows the analysis to keep smaller miss sets without losing precision. At the end, *assign* produces one configuration for each index in S_i. We use the following notation: if S is a set of indices and h a secondary value, then $\overline{(S, h)} = \lambda i \in I$. if $(i \in S)$ then h else $\perp$.

Note that bifurcation can produce up to four cases and each case may yield up to two configurations. However,

assign$(e_0, e_1, \rho, i, e^+, e^-, b_0, b_1)$:

```
1   r = L[[e_0]](ρ)
2   if (b_0 ∧ ¬b_1) then
3     if (i(r) ≤ k) then S_i = { i[r ↦ i(r) − 1] }
4                   else S_i = { i[r ↦ k], i[r ↦ ∞] }
5   else if (¬b_0 ∧ b_1) then
6     if (i(r) < k) then S_i = { i[r ↦ i(r) + 1] }
7                   else S_i = { i[r ↦ ∞] }
8   else S_i = { i }
9
10  e_n^+ = {e ∈ e^+ | S[[e]]_v(ρ, r) ∨ (S[[e]]_l(ρ, r) ∧ b_1)}
11  e_n^- = {e ∈ e^- | S[[e]]_v(ρ, r) ∨ (S[[e]]_l(ρ, r) ∧ ¬b_1)}
12
13  if (S[[e_0]]_l(ρ, r) ∧ b_1)   then e_n^+ ∪= {e_0}
14  if (S[[e_0]]_l(ρ, r) ∧ ¬b_1) then e_n^- ∪= {e_0}
15
16  if (S[[e_0]]_l(ρ, r) ∧ S[[e_1]]_l(ρ, r)) then
17      e_n^+ ∪= ( e_n^+[*e_0/*e_1] ∩ E'_p )
18      e_n^- ∪= ( e_n^-[*e_0/*e_1] ∩ E'_p )
19
20  e_n^- = {e ∈ e_n^- | ∀i' ∈ S_i . i'(L[[e]]ρ) = 0}
21
22  return (S_i, (e_n^+, e_n^-))
```

Figure 11: Helper function *assign*.

there can be at most three resulting configurations after each assignment, since we merge configurations with the same index, and the reference count from the updated region can only increase by one, decrease by one, or remain unchanged. In the example from Section 2 the analysis of each statement and configuration produces either one or two configurations.

Finally, transfer functions map configurations with a secondary value of $\perp$ to bottom abstractions $a_{\perp}$. The same is true for all of the other transfer functions in the algorithm.

5.2.4 Analysis of Malloc and Free

Figure 12 shows the analysis of dynamic allocation and deallocation statements. The transfer function of each allocation statement $[\![e \leftarrow \textbf{malloc}]\!]$ is straightforward. First, the assignment performed is equivalent to that of a nullification $[\![e \leftarrow \textbf{null}]\!]$ because the tracked location is guaranteed to be distinct from the fresh location returned by malloc (even if it happens to be allocated at the same site), and null has the same property. Second, since the contents of the fresh location are not initialized, we can add its field expressions to the miss set if e is stable.

The generating function $[\![e \leftarrow \textbf{malloc}]\!]^{gen}$ yields a configuration (i, c) for the newly created location such that the index i records a reference count of 1 from the region of e and 0 from all other regions. The secondary value h records e as a hit expression, and adds field expressions to the miss set if e is stable.

Finally, the analysis models the transfer function for deallocation statements $[\![\textbf{free}(e)]\!]$ as a sequence of assignments that nullify each field of the deallocated structure. This ensures that the analysis counts references only from valid, allocated locations.

$[\![e \leftarrow \textbf{malloc}]\!](\rho, c)$:
  $a = [\![e \leftarrow \textbf{null}]\!](\rho, c)$
  if $(\mathcal{S}[\![e]\!]_l(\rho, \mathcal{L}[\![e]\!](\rho)))$ then
    $e_n^- = \{(*e).f \mid f \in F\} \cap E_p$
    $a = \{(i, (e^+, e^- \cup e_n^-)) \mid a\ i = (e^+, e^-)\}$
  return a

$[\![e \leftarrow \textbf{malloc}]\!]^{gen}(\rho)$:
  $r = \mathcal{L}[\![e]\!](\rho)$
  $i = \lambda r'$. if $(r' = r)$ then 1 else 0
  $e^- = \{(*e).f \mid f \in F\} \cap E_p$
  if $(\mathcal{S}[\![e]\!]_l(\rho, r))$ then $h = \{\{e\},\ e^-\}$
    else $h = \{\emptyset,\ \emptyset\}$
  return (i, h)

$[\![\textbf{free}(e)]\!](\rho, c)$:
  $a = [\![t \leftarrow *e]\!](\rho, c) \quad (t\ \textit{fresh})$
  $a = \sqcup_{\{i \in I \mid ai \neq \bot\}} [\![t.f_1 \leftarrow \textbf{null}]\!](\rho, (i, ai))$
  ...
  $a = \sqcup_{\{i \in I \mid ai \neq \bot\}} [\![t.f_m \leftarrow \textbf{null}]\!](\rho, (i, ai))$
  $a = \sqcup_{\{i \in I \mid ai \neq \bot\}} [\![t \leftarrow \textbf{null}]\!](\rho, (i, ai))$
  return a

Figure 12: Analysis of **malloc** and **free**.

5.2.5 Conditional Branches

The analysis extracts useful information from tests in if and while statements because the condition expression e is guaranteed to miss the tracked location if the branch implies that e is null. To take advantage of this information, the analysis invokes the decision function $\mathcal{D}[\![e]\!](\rho, c)$. If the returned value is "?", then the analysis adds e to the miss set e^-; if the returned value is "+", then the configuration is inconsistent with the actual state of the program and the analysis reduces the secondary value to $\bot$.

The latter case occurs in the example from Section 2, where the condition `x != null` allows the analysis to filter out the configuration X^1 after the while loop.

5.3 Inter-Procedural Analysis

We formulate the inter-procedural analysis algorithm as a context-sensitive analysis that distinguishes between different calling contexts of the same procedure. Key here is that we take advantage of our fine-grained abstraction and define procedure contexts to be individual configurations, not entire heap abstractions.

The result for an input configuration context is a set of corresponding output configurations at the end of the procedure. If we consider a graph model (e.g., the example graph in Figure 5) that shows how the state of the tracked location changes during execution, we can express the input-ouput relationships for procedure contexts as reachability relations: the outputs are those exit configurations that are reachable from the input configuration (the input context). We represent the graph by "labeling" configurations with the entry index that they originated from. This separates out configurations that originated from different entries, allowing the analysis to quickly determine both the output configurations for a given input, and the input configuration for a given output. For this, we extend the index domain in the analysis with the index at entry:

$$\text{Index values:} \quad (i^c, i^e) \in I_p \quad = \quad I \times I$$

For all $s_e \in S_{entry},\ s_c \in S_{call},\ i \in I_p$:

[IN] :

$$\mathit{Res}(\bullet s_e)i \ \sqsupseteq \bigsqcup_{\substack{i' \in I_p \\ \mathit{tgt}(s_c)=\mathit{fn}(s_e)}} (\llbracket s_c \rrbracket^{in}(\rho, \mu_{s_c}, (i', \mathit{Res}(\bullet s_c)i')))i$$

[OUT]

$$\mathit{Res}(s_c\bullet)i \ = \bigsqcup_{i', i'' \in I_p} (\llbracket s_c \rrbracket^{out}(\rho, \mu_{s_c}, (i', \mathit{Res}(\bullet s_c)i'), (i'', \mathit{Res}(s_x\bullet)i'')))i$$

where $s_x = \mathit{exit}(\mathit{tgt}(s_c))$

Figure 13: Additional inter-procedural dataflow equations.

The entry index i^e has no bearing on the intra-procedural transfer functions – they simply preserve this value, and operate on the current index i^c.

The analysis at procedure calls must account for the assignment of actuals to formals and for the change of analysis domain between the caller and the callee. We use two functions: an *input* function $\llbracket s \rrbracket^{in}(\rho, \mu_s, c) \in A$ which takes a caller configuration c (before the call) and produces the set of configurations at the entry point in the callee; and an *output* function $\llbracket s \rrbracket^{out}(\rho, \mu_s, c, c') \in A$ which takes an exit configuration c' at the end of the procedure, along with the caller configuration c that identifies the calling context, to produce a set of caller configurations after the call. Both functions require a the region store ρ and the region mapping μ_s at the call site in question.

Using the input and output functions, we express the inter-procedural analysis at call sites with two additional dataflow equations, shown in Figure 13. We use the following notations: $\mathit{fn}(s) \in P$ is the procedure that statement s belongs to; $\mathit{tgt}(s_c) \in P$ is the target procedure of a call statement s_c; S_{call} is the set of call sites in the program; and $\mathit{entry}(p) \in S_{entry}$ and $\mathit{exit}(p) \in S_{exit}$ are the entry and exit statements of procedure p, respectively. Equation [IN] performs the transfer from the caller to the callee; and [OUT] transfers the analysis back to the caller. The main loop of the modified worklist algorithm is shown in Figure 14; the remainder of the algorithm is unchanged. The algorithm propagates output configurations to the callers when either new exit configurations in the callee or when new call site configurations (i.e., input contexts) in the caller are found.

Figure 15 shows the input and output transfer functions. The entry transfer function accomplishes two things. First, it performs the assignments into the parameters, which may generate new references to the tracked cell. Second, it adjusts the regions of existing references according to the renaming map μ. References whose regions are not in the range of μ are not visible by the callee and are discarded by the slicing function S_{in}. The exit transfer function performs the reversed tasks. First, it accounts for context-sensitivity using the if statement at the beginning: it checks if the exit configuration in the caller originates from the input context $((i^c, i^e), h)$ before the call. If so, it restores the hit and miss expressions discarded at entry and adjusts the region counts.

5.4 Extensions

We present two extensions: incremental computation of shapes, and detection of memory errors.

5.4.1 Demand-Driven and Incremental Analyses

The goal of a demand-driven analysis is to analyze a selected set of dynamic structures in the program, created at a one or a few allocation sites. In our framework, this can be achieved with minimal effort: just apply the dataflow equation [ALLOC] from Figure 8 to the selected allocation sites. The effect is that the dataflow information will get seeded just at those sites, and the worklist algorithm will automatically propagate that information through the program. In particular, the inter-procedural analysis will propagate shape information from procedures that contain these allocations out to their callers.

Our framework also enables the incremental computation of shapes. To explore new allocation sites, we seed the dataflow information at the new sites, initialize the worklist to contain successors of those allocations, and then run the

WORKLIST(DataflowInfo a_0)

15 ...
16 **while** (W is not empty)
17 remove some (s, i) from W
18 **if** $s \in S_{call}$ **then**
19 **let** $s_e = \mathit{entry}(\mathit{tgt}(s)), s_x = \mathit{exit}(\mathit{tgt}(s))$
20 $\mathit{Res}(\bullet s_e) \sqcup= [\![s]\!]^{in}(\rho, \mu_s, \mathit{Res}(\bullet s, i))$
21 **for** each i' s.th. $\mathit{Res}(\bullet s_e)i'$ has changed
22 $W \cup= \{(s_e, i')\}$
23 **for** each i' s.th. $\mathit{Res}(s_x \bullet)i' \neq \bot$
24 $\mathit{Res}(s\bullet) \sqcup=$
25 $[\![s]\!]^{out}(\rho, \mu_s, \mathit{Res}(\bullet s, i), \mathit{Res}(\bullet s_x, i'))$
26 **else if** $s \in S_{exit}$ **then**
27 **for** s_c, i' s.th. $\mathit{tgt}(s_c) = \mathit{fn}(s), \mathit{Res}(\bullet s_c)i' \neq \bot$
28 $\mathit{Res}(s_c\bullet) \sqcup=$
29 $[\![s_c]\!]^{out}(\rho, \mu_{s_c}, \mathit{Res}(\bullet s_c, i'), \mathit{Res}(\bullet s, i))$
30 **else**
31 $\mathit{Res}(s\bullet) \sqcup= [\![s]\!](\rho, \mathit{Res}(\bullet s, i))$
32
33 **for** each s', i' s.th. $\mathit{Res}(s'\bullet)i'$ has changed
34 **for** $s'' \in \mathit{succ}(s')$
35 $\mathit{Res}(\bullet s'')i' \sqcup= \mathit{Res}(s'\bullet)i'$
36 $W \cup= \{(s'', i')\}$

Figure 14: Inter-procedural worklist algorithm

worklist algorithm. Key to the incremental computation is that our fine-grained abstraction based on configurations allows the analysis to reuse results from previously analyzed sites. This is possible both at the intra-procedural level, when the newly generated configurations match existing ones at certain program points; and at the inter-procedural level, when new calling contexts match existing contexts.

5.4.2 Memory Error Detection

We extend our analysis algorithm to enable the static detection of memory errors such as memory accessed through dangling pointers, memory leaks, or multiple frees. To detect such errors, we enrich the index with a flag indicating whether the tracked cell has been freed:

$$\text{Index values:} \quad i_f \in I_f \quad = \quad \{\text{true}, \text{false}\} \times I$$

Since this information is in the index, for any configuration we know precisely whether or not the cell has been freed. Most of the transfer functions leave this flag unchanged; and since merging configurations does not combine different index values (as before), there is no merging of flags. Only two more changes are required in the analysis. First, after the initial allocation this flag is false:

$$[\![e \leftarrow \textbf{malloc}]\!]_f^{gen}(\rho) = ((\text{false}, i_a), h_a)$$

Second, we must change the transfer function for free statements, since the allocation state of the tracked cell may change at these points. We express the modified transfer function for **free** using the original one, which preserves the

$[\![\textbf{call}\ q(e_1,..,e_n)]\!]^{in}(\rho,\mu,(i,h)) :$
$\quad a = [\![p_1 \leftarrow e_1]\!] \ldots [\![p_n \leftarrow e_n]\!](\rho,\{(i,h)\})$
$\quad \textbf{return}\ \bigsqcup_{\{i \mid a\ i \neq \bot\}} S_{in}(\rho,\mu,(i,a\ i))$

where $S_{in}(\rho,\mu,((i^c,i^e),(e^+,e^-))) :$
$\quad e_q^+ = \{e \in e^+ \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}[\![e]\!]_v(\rho,r))\}$
$\quad e_q^- = \{e \in e^- \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}[\![e]\!]_v(\rho,r))\}$
$\quad i_q^e = \lambda r \in R^q\ .\ i^c(\mu\, r)$
$\quad \textbf{return}\ ((i_q^e,i_q^e),(e^+ - e_q^+, e^- - e_q^-))$

$[\![\textbf{call}\ q(e_1,..,e_n)]\!]^{out}(\rho,\mu,((i^c,i^e),h),((i_x^c,i_x^e),h_x)) :$
$\quad \textbf{let}\ s_c = \textbf{call}\ q(e_1,..,e_n)$
$\quad \textbf{if}\ [\![s_c]\!]^{in}(\rho,\mu,((i^c,i^e),h))(i_x^e,i_x^e) = \bot\ \textbf{then}$
$\qquad \textbf{return}\ \lambda i.\bot$
$\quad \textbf{let}\ (e^+,e^-) = h$
$\quad \textbf{let}\ (e_x^+,e_x^-) = h_x$
$\quad e_q^+ = \{e \in e^+ \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}[\![e]\!]_v(\rho,r))\}$
$\quad e_q^- = \{e \in e^- \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}[\![e]\!]_v(\rho,r))\}$
$\quad i_q^c = \lambda r \in R^q\ .\ \text{if}\ (r \notin \mathit{range}(\mu))\ \text{then}\ i^c(r)$
$\qquad\qquad\qquad\qquad\qquad\qquad \text{else}\ i_x^c(\mu^{-1}(r))$
$\quad \textbf{return}\ ((i_q^c,i^e),(e_x^+ \cup e_q^+, e_x^- \cup e_q^-))$

Figure 15: Inter-procedural transfer functions.

allocation flag, and using the decision function to determine whether the freed location is the one being tracked or not:

$$\begin{array}{rcl}[\![\textbf{free}(e)]\!]_f(\rho,((f,i),h)) = & \textbf{case } (\mathcal{D}[\![e]\!](\rho,c)) \textbf{ of} \\ \text{“}-\text{”} & \Rightarrow & [\![\textbf{free}(e)]\!]\ (\rho,((f,i),h)) \\ \text{“}+\text{”} & \Rightarrow & [\![\textbf{free}(e)]\!]\ (\rho,((\text{true},i),h)) \\ \text{“}?\text{”} & \Rightarrow & [\![\textbf{free}(e)]\!]\ (\rho,((f,i),h)) \sqcup \\ & & [\![\textbf{free}(e)]\!]\ (\rho,((\text{true},i),h))\end{array}$$

With these addition, the analysis can proceed to detect errors. To identify memory leaks, it checks if there are no incoming references to a cell, but the cell was never freed. While a configuration with no incoming references means there are no pointers to the cell from regions that are in scope, this does not mean that functions further up the call stack do not still have pointers to the cell. To be sure that none of these functions have pointers to the tracked cell, the cell must not have been allocated at entry to the function. We classify memory leaks as configurations $((\text{false}, \lambda r.0), h)$ that are reachable from the boundary configurations in a_0. However, leak detection suffers from the standard problem of reference counting, that it cannot detect leaked cycles.

To detect double frees, the analysis performs the following check. For any statement $s = \textbf{free}(e)$, if $((\text{true}, i), h) \in \mathit{Res}(\bullet s)$ and $\mathcal{D}[\![e]\!](\rho,(i,h)) \neq \text{“}-\text{”}$, a possible double free has occurred. And to identify accesses to deallocated memory the analysis checks the following. For any statement s, define E_s as the set of expressions that are dereferenced by s: $\{e \mid *e \text{ appears in } s\}$. If $((\text{true}, i), h) \in \mathit{Res}(\bullet s)$ and $\mathcal{D}[\![e]\!](\rho,(i,h)) \neq \text{“}-\text{”}$ for any $e \in E_s$, a possible reference to deallocated memory has occurred.

6 Current Limitations

The current algorithm presented in this paper has several limitations. These include the following:

- *Spurious configurations*. The analysis may generate spurious configurations, but still determine the correct final shape in spite of this imprecision. This is the case in our example: configurations such as $\mathrm{Y}^1\mathrm{T}^1\mathrm{L}^1$ or $\mathrm{X}^1\mathrm{Y}^1\mathrm{L}^1$ cannot happen for acyclic and disjoint input lists. Ruling such cases out would require more complex judgments (e.g., for $\mathrm{Y}^1\mathrm{T}^1\mathrm{L}^1$); or capturing more local information in the configurations (e.g., for $\mathrm{X}^1\mathrm{Y}^1\mathrm{L}^1$).

- *Complex structural invariants*. For manipulations of linked structures with complex invariants, our algorithm may fail to identify the correct shape. For example, our algorithm cannot determine that shapes are preserved for operations on doubly linked list, because our configurations cannot record the doubly linked list invariant.

- *Robustness*. The analysis may fail if the same program is written in a different manner. This happens in our example: if we replace statement `x=t->n` with `x=x->n`, the algorithm will fail to verify the shape property. That is because configurations do not record facts such as variable equality, in this case that `x=t` before this statement.

- *Worst-case exponential blowup*. As in the case of most existing shape abstractions, the number of possible configurations has an exponential worst-case complexity. We have seen such cases occurring in practice, but only as a result of the imprecision of our algorithm, for instance for traversals of doubly linked lists using multiple pointers.

However, these are limitations of our particular algorithm, but not of the framework. We believe that they can be handled using configurations that contain more local information rather than just hit and miss expressions. We keep the algorithm in this paper cleaner to emphasize the concept and to make it amenable to formal presentation. These issues will be the subject of future work.

7 Formal Framework

This section provides formal results for the shape analysis algorithm. To keep the presentation simpler, we restrict ourselves to the intra-procedural part of the analysis. First, we show that the worklist algorithm is guaranteed to

terminate and that it faithfully implements the dataflow equation. Second, we give a formal semantics for our language and show that the transfer functions in our algorithm are correct with respect to this underlying semantics.

7.1 Worklist Algorithm: Correctness and Termination

First, we prove that the worklist algorithm from Section 5 provides a solution for our system of dataflow equations. The theorem below expresses this result.

Theorem 1 (Worklist Correctness) *If transfer functions map each configuration with $\perp$ secondary value to $a_\perp$, then the worklist algorithm from Figure 9 yields the least fixed point of the system of dataflow equations from Figure 8.*

Proof. Equation JOIN holds at after the loop at lines 2-3; then, whenever the result changes after a program point (lines 10 and 18), the algorithm adjusts the result at the successor points to ensure that this equation holds (lines 13 and 21). Further, equation INIT holds after the initialization at lines 4-7 and ALLOC holds after the loop between lines 8-14; and both are being maintained during the main loop. Equation TRANSF also holds after the loop at lines 2-3, because of the assumption about transfer functions and bottom values. Then, the execution of the main loop maintains the invariant that TRANSF holds for all the pairs (s, i) that are *not present* in the worklist. Since the worklist is empty at the end of the algorithm, this equation will be satisfied for all statements and all index values. Finally, the computed result is a least fixed point because the algorithm maintains the invariant that, at each point during its execution, any other solution to the system is greater than the current result. □

Next, we show that the worklist algorithm terminates. For this, we show that the transfer functions that our analysis uses are monotonic. Along with the fact that our shape abstraction is finite, this indicates that the algorithm always terminates. The proof uses several lemmas that describe the monotonicity of the decision function, and of the assignments.

Lemma 1 (Decision Function Monotonicity) *Let ρ be a sound region abstraction, and two configurations $(i, (e_0^+, e_0^-))$, $(i, (e_1^+, e_1^-))$ computed by the analysis such that $(e_0^+, e_0^-) \sqsubseteq (e_1^+, e_1^-)$. Define an ordering over $\{+, -, ?\}$ such that $- \sqsubseteq ?$ and $+ \sqsubseteq ?$. Then: $\mathcal{D}[\![e]\!](\rho, (i, (e_0^+, e_0^-))) \sqsubseteq \mathcal{D}[\![e]\!](\rho, (i, (e_1^+, e_1^-)))$, for all e.*

Proof. Let $v_0 = \mathcal{D}[\![e]\!](\rho, (i, (e_0^+, e_0^-)))$ and $v_1 = \mathcal{D}[\![e]\!](\rho, (i, (e_1^+, e_1^-)))$. By the definition of $\sqsubseteq$ on H, $e_1^+ \subseteq e_0^+$ and $e_1^- \subseteq e_0^-$. We now enumerate the possible combinations for v_0 and v_1 that would contradict this claim. This proof relies on the fact that the decision function $\mathcal{D}[\![\cdot]\!]$ is well-formed for all configurations computed during the analysis (see Lemma 14). We have the following cases:

- If $v_1 = +$ and $v_0 \in \{-, ?\}$, then $e \in e_1^+$ and $e \in e_0^+$, a contradiction.
- Suppose $v_1 = -$ and $v_0 \in \{+, ?\}$. If $i(\mathcal{L}[\![e]\!]\rho) = 0$ or $e =$ **null** or $e = \&x$, then $v_0 = -$ and a contradiction. If $e \in e_1^-$, then $e \in e_0^-$, a contradiction.

By disjunction elimination over the possible combinations contradicting the claim, we have a contradiction that verifies the claim. □

Lemma 2 (Assign Helper Function Monotonicity) *If e_0^+, e_0^-, e_1^+, and e_1^- are such that $(e_0^+, e_0^-) \sqsubseteq (e_1^+, e_1^-)$, then: assign$(e_0, e_1, \rho, i, e_0^+, e_0^-, b_0, b_1) \sqsubseteq$ assign$(e_0, e_1, \rho, i, e_1^+, e_1^-, b_0, b_1)$ for all $e_0, e_1, \rho, i, b_0, b_1$.*

Proof. The index sets S_i computed by "*assign*" are independent of the hit and miss sets, so we only need to show that at exit $(e_{n0}^+, e_{n0}^-) \sqsubseteq (e_{n1}^+, e_{n1}^-)$. We do this by executing the two calls to "*assign*" side by side and showing that the property is maintained throughout.

- Statements 10 and 11. Statement 10 is equivalent to $e_n^+ = e^+ \cap \{e' \in E_p \mid X\}$. The predicate X is independent of the sets e^+ and e^-, so the set intersected with is identical across the two executions. Since $e_1^+ \subset e_0^+$, after the assign $e_{n1}^+ \subset e_{n0}^+$. The case is the same with statement 11, and after executing both we have that $(e_{n0}^+, e_{n0}^-) \sqsubseteq (e_{n1}^+, e_{n1}^-)$.

- Statements 13 and 14. The conditions in these two statements will be the same for both executions. Thus, the sets will change to the same degree in the two executions and the ordering is maintained.
- Statement 16. The condition will be the same for both executions, so statements 17 and 18 will either be executed in both executions or in neither.
- Statements 17 and 18. The sets produced by substituting e_1 for e_0 increase in elements with the size of the set being substituted in: each element is some $e[*e_0/*e_1]$, and if this is in set a but not b, then there is some e that is in a but not b. Thus, the right side of the join has elements in proportion with the left side, and the assignments will maintain the ordering across the two executions.
- Statement 20. Since S_i is constant across the executions, this is equivalent to intersecting e^-_{n0} and e^-_{n1} with a fixed set, which will maintain the ordering between them.

Since each statement operating on the secondary values preserves the ordering between them, we have at the end that $(e^+_{n0}, e^-_{n0}) \sqsubseteq (e^+_{n1}, e^-_{n1})$. Since the S_i are constant, the results of the two executions of "*assign*" have the desired ordering. □

Lemma 3 (Assignment Monotonicity) *Let $e_0 \leftarrow e_1$ an assignment in the program and ρ a sound region abstraction. Consider two configurations (i, h) and (i, h') computed by the analysis, such that $h \sqsubseteq h'$. Then:*

$$[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) \sqsubseteq [\![e_0 \leftarrow e_1]\!](\rho, (i, h')) \qquad (I)$$

Proof. If $h = \bot$, then $[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) = a_\bot$ and the relation is trivially satisfied.

If $h = (e^+, e^-)$, then $h' \neq \bot$, so $h' = (e^+_p, e^-_p)$. Let $v_0 = \mathcal{D}[\![e_0]\!](\rho, (i, h))$, $v_1 = \mathcal{D}[\![e_1]\!](\rho, (i, h))$, $v'_0 = \mathcal{D}[\![e_0]\!](\rho, (i, h'))$, $v'_1 = \mathcal{D}[\![e_1]\!](\rho, (i, h'))$. We will show that (I) holds for each combination of values of v_0 and v_1, so by disjunction elimination (I) holds.

1. $v_0 \in \{-, +\}, v_1 \in \{-, +\}$
   Then $[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) =$
   assign$(e_0, e_1, \rho, i, e^+, e^-, v_0 = +, v_1 = +)$ (a)
   By Lemma 1, v'_0 is either either v_0 or ?, and v'_1 is either v_1 or ?. We consider each of the four resulting subcases.

   Case A: $v'_0 = v_0$, $v'_1 = v_1$:
   $[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) =$
   assign$(e_0, e_1, \rho, i, e^+_p, e^-_p, v'_0 = +, v'_1 = +)$ (b)
   By Lemma 2, $a \sqsubseteq b$ and (I) holds.

   Case B: $v'_0 = v_0$, $v'_1 =$?:
   $[\![e_0 \leftarrow e_1]\!](\rho, (i, h')) =$
   assign$(e_0, e_1, \rho, i, e^+_p \cup \{e_1\}, e^-_p, v'_0 = +, \text{true})$ $(b) \sqcup$
   assign$(e_0, e_1, \rho, i, e^+_p, e^-_p \cup \{e_1\}, v'_0 = +, \text{false})$ (c)
   If $v_1 = +$, then $e_1 \in e^+$, and by Lemma 2, $a \sqsubseteq b$ and (I) holds. If $v_1 = -$, then $e_1 \in e^-$, and by Lemma 2, $a \sqsubseteq c$ and (I) holds.

   Case C: $v'_0 =$?, $v'_1 = v_1$. Parallel to *Case B.*

   Case D: $v'_0 =$?, $v'_1 =$?.
   $[\![e_0 \leftarrow e_1]\!](\rho, (i, h')) =$
   assign$(e_0, e_1, \rho, i, e^+_p \cup \{e_0, e_1\}, e^-_p, \text{t}, \text{t})$ $(b) \sqcup$
   assign$(e_0, e_1, \rho, i, e^+_p \cup \{e_0\}, e^-_p \cup \{e_1\}, \text{t}, \text{f})$ $(c) \sqcup$

assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_1\}, e_p^- \cup \{e_0\}, \text{f}, \text{t})\ (d) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+, e_p^- \cup \{e_0, e_1\}, \text{f}, \text{f})\ (e)$

If $v_0 = +$ and $v_1 = +$, then $e_0, e_1 \in e^+$, by Lemma 2, $a \sqsubseteq b$ and (I) holds.

If $v_0 = +$ and $v_1 = -$, then $e_0 \in e^+$ and $e_1 \in e^-$, by Lemma 2, $a \sqsubseteq c$ and (I) holds.

If $v_0 = -$ and $v_1 = +$, then $e_0 \in e^-$ and $e_1 \in e^+$, by Lemma 2, $a \sqsubseteq d$ and (I) holds.

If $v_0 = -$ and $v_1 = -$, then $e_0, e_1 \in e^-$, by Lemma 2, $a \sqsubseteq e$ and (I) holds.

2. $v_0 \in \{-, +\}, v_1 = ?$

$[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) =$
assign$(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^-, v_0 = +, \text{true})\ (a) \sqcup$
assign$(e_0, e_1, \rho, i, e^+, e^- \cup \{e_1\}, v_0 = +, \text{false})\ (b)$

By Lemma 1, v_0' is either v_0 or $?$, and v_1' is $?$. We consider each of these subcases.

Case A: $v_0' = v_0$:

$[\![e_0 \leftarrow e_1]\!](\rho, (i, h')) =$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_1\}, e_p^-, v_0' = +, \text{true})\ (c) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+, e_p^- \cup \{e_1\}, v_0' = +, \text{false})\ (d)$

By Lemma 2, $a \sqsubseteq c$ and $b \sqsubseteq d$, so $a \sqcup b \sqsubseteq c \sqcup d$ and (I) holds.

Case B: $v_0' = ?$:

$[\![e_0 \leftarrow e_1]\!](\rho, (i, h')) =$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_0, e_1\}, e_p^-, \text{t}, \text{t})\ (c) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_0\}, e_p^- \cup \{e_1\}, \text{t}, \text{f})\ (d) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_1\}, e_p^- \cup \{e_0\}, \text{f}, \text{t})\ (e) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+, e_p^- \cup \{e_0, e_1\}, \text{f}, \text{f})\ (f)$

If $v_0 = +$, then $e_0 \in e^+$, by Lemma 2, $a \sqsubseteq c$ and $b \sqsubseteq d$, so $a \sqcup b \sqsubseteq c \sqcup d \sqcup ...$ and (I) holds.

If $v_0 = -$, then $e_0 \in e^-$, by Lemma 2, $a \sqsubseteq e$ and $b \sqsubseteq f$, so $a \sqcup b \sqsubseteq e \sqcup f \sqcup ...$ and (I) holds.

3. $v_0 = ?, v_1 \in \{-, +\}$

Parallel to $v_0 \in \{-, +\}, v_1 = ?$.

4. $v_0 = ?, v_1 = ?$

$[\![e_0 \leftarrow e_1]\!](\rho, (i, h)) =$
assign$(e_0, e_1, \rho, i, e^+ \cup \{e_0, e_1\}, e^-, \text{t}, \text{t})\ (a) \sqcup$
assign$(e_0, e_1, \rho, i, e^+ \cup \{e_0\}, e^- \cup \{e_1\}, \text{t}, \text{f})\ (b) \sqcup$
assign$(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^- \cup \{e_0\}, \text{f}, \text{t})\ (c) \sqcup$
assign$(e_0, e_1, \rho, i, e^+, e^- \cup \{e_0, e_1\}, \text{f}, \text{f})\ (d)$

By Lemma 1, v_0' and v_1' are both $?$. Therefore:

$[\![e_0 \leftarrow e_1]\!](\rho, (i, h')) =$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_0, e_1\}, e_p^-, \text{t}, \text{t})(e) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_0\}, e_p^- \cup \{e_1\}, \text{t}, \text{f})(f) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+ \cup \{e_1\}, e_p^- \cup \{e_0\}, \text{f}, \text{t})(g) \sqcup$
assign$(e_0, e_1, \rho, i, e_p^+, e_p^- \cup \{e_0, e_1\}, \text{f}, \text{f})(h)$

By Lemma 2, $a \sqsubseteq e$, $b \sqsubseteq f$, $c \sqsubseteq g$, and $d \sqsubseteq h$, so $a \sqcup b \sqcup c \sqcup d \sqsubseteq e \sqcup f \sqcup g \sqcup h$ and (I) holds.

□

Theorem 2 (Monotonicity) *The transfer functions $[\![s]\!]$ for assignments, malloc, and free are monotonic in the secondary value: given an abstract store ρ, an index $i \in I$, and secondary values $h, h' \in H$ such that $h \sqsubseteq h'$, we have $[\![s]\!](i,h) \sqsubseteq [\![s]\!](i,h')$.*

Proof. For assignments, the result is given by Lemma 3. For free and malloc, the result holds because assignments are monotonic. For malloc, the set of field expressions that may be added to the miss set does not depend on h, and neither does the stability condition that adds expressions to the miss set, so that does not affect monotonicity. □

Corollary 1 (Termination) *The worklist algorithm from Figure 9 is guaranteed to terminate.*

7.2 Correctness of Transfer Functions

The goal of this section is to provide a proof of correctness of the transfer functions in our algorithm with respect to the semantics of the language. First, we give an operational semantics of the language. Second we define what is means for our region abstraction and for our shape abstraction to be sound. Third, we prove that the transfer functions preserve the soundness of the shape abstraction. The proof only concerns the shape analysis algorithm; we assume that the points-to analysis algorithm yields a sound region abstraction.

7.2.1 Operational Semantics

We define a large-step operational model for the language from Figure 7. This model uses the following semantic domains for locations, values, and concrete stores:

$$\begin{array}{ll} \text{locations:} & l \in L \\ \text{values:} & v \in L + \{\mathit{null}\} \\ \text{stores:} & \sigma = (\sigma_v, \sigma_l, \sigma_f) \in (V \to L) \ \times \ (L \rightharpoonup L + \{\mathit{null}\}) \ \times \ ((L \times F) \rightharpoonup L) \end{array}$$

We define the operational semantics using the following three evaluation functions. The relation $\langle e, \sigma\rangle \to_l l$ evaluates expression e in store σ to produce the location of e (i.e., the l-value of e). Only variables, pointer dereferences, and structure fields can be evaluated for their location:

$$\frac{x \in \mathit{dom}_v(\sigma)}{\langle x, \sigma\rangle \to_l \sigma(x)} \qquad \frac{\langle e, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma) \quad \sigma(l) = l'}{\langle *e, \sigma\rangle \to_l l'} \qquad \frac{\langle e, \sigma\rangle \to_l l \quad (l,f) \in \mathit{dom}_f(\sigma)}{\langle e.f, \sigma\rangle \to_l \sigma(l,f)}$$

Next, we define relation $\langle e, \sigma\rangle \to_e v$ that evaluates e in the concrete store σ to produce the value of e (i.e., its r-value). This relation uses the l-value evaluation defined above:

$$\frac{\langle e, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma)}{\langle e, \sigma\rangle \to_e \sigma(l)} \qquad \frac{\langle e, \sigma\rangle \to_l l}{\langle \&e, \sigma\rangle \to_e l} \qquad \frac{}{\langle \mathbf{null}, \sigma\rangle \to_e \mathit{null}}$$

Finally, we and evaluation relation $\langle s, \sigma\rangle \to_s \sigma'$ for statements. The relation evaluates statement s in the concrete store σ to produce a resulting store σ' after the statement:

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma) \quad \{l_f\}_{f\in F} \ \mathit{fresh} \\ \sigma' = \sigma[l \mapsto l_{f_1}][l_f \mapsto \mathit{null}]_{f\in F}[(l_{f_1}, f) \mapsto l_f]_{f\in F}\end{array}}{\langle e \leftarrow \mathbf{malloc}, \sigma\rangle \to_s \sigma'} \qquad \frac{\begin{array}{c}\langle e, \sigma\rangle \to_e l \quad \forall f \in F : (l,f) \in \mathit{dom}_f(\sigma) \\ \sigma' = (\sigma - \{(l,f) \mapsto \_\}_{f\in F}) - \{\sigma(l,f) \mapsto \_\}_{f\in F}\end{array}}{\langle \mathbf{free}(e), \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e_0, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma) \quad \langle e_1, \sigma\rangle \to_e v}{\langle e_0 \leftarrow e_1, \sigma\rangle \to_s \sigma[l \mapsto u]} \qquad \frac{\begin{array}{c}\mathit{prog}(f) = ((x_1, .., x_n), s) \quad \forall i = 1..n : \ \langle e_i, \sigma\rangle \to_e v_i \\ \langle s, \sigma[\sigma(x_i) \mapsto v_i]_{i=1..n}\rangle \to_s \sigma'\end{array}}{\langle \mathbf{call}\ p(e_1, .., e_n), \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e,\sigma\rangle \rightarrow_e v = \mathit{null} \quad \langle s_0,\sigma\rangle \rightarrow_s \sigma'}{\langle \textbf{if } (e)\ s_0 \textbf{ else } s_1,\sigma\rangle \rightarrow_s \sigma'} \qquad \frac{\langle e,\sigma\rangle \rightarrow_e v \neq \mathit{null} \quad \langle s_1,\sigma\rangle \rightarrow_s \sigma'}{\langle \textbf{if } (e)\ s_0 \textbf{ else } s_1,\sigma\rangle \rightarrow_s \sigma'}$$

$$\frac{\langle s_0,\sigma\rangle \rightarrow_s \sigma'' \quad \langle s_1,\sigma''\rangle \rightarrow_s \sigma'}{\langle s_0\,;\ s_1,\sigma\rangle \rightarrow_s \sigma'} \qquad \frac{\langle e,\sigma\rangle \rightarrow_e v = \mathit{null}}{\langle \textbf{while } (e)\ s,\sigma\rangle \rightarrow_s \sigma} \qquad \frac{\langle e,\sigma\rangle \rightarrow_e v \neq \mathit{null} \quad \langle s,\sigma\rangle \rightarrow_s \sigma'' \quad \langle \textbf{while } (e)\ s,\sigma''\rangle \rightarrow_s \sigma'}{\langle \textbf{while } (e)\ s,\sigma\rangle \rightarrow_s \sigma'}$$

7.2.2 Definition of Abstraction Soundness

In this section, we express what it means for a region abstraction and for a shape abstraction to be sound. Because the points-to analysis that produces the region abstraction is flow-insensitive, this abstraction must describe all of the concrete stores that occur in the program. The shape analysis, on the other hand, is flow-sensitive, so the shape abstraction at each program point must characterize the concrete stores that occur just at that program point.

We first define soundness for the region abstraction. This definition is based on a notion of consistency between concrete and abstract stores, defined as follows.

Definition 1 (Region Consistency) *Given an abstract store $\rho = (\rho_v, \rho_r, \rho_f)$ and a concrete store $\sigma = (\sigma_v, \sigma_l, \sigma_f)$, we say that a region partial map $\alpha : L \rightharpoonup R$ is (σ, ρ)-consistent if:*

- *range*$(\alpha) \subseteq$ *dom*(ρ_r)*;*
- $\alpha(\sigma_v(x)) = \rho_v(x),\ \forall x \in V$*; and*
- $\rho_r(\alpha(l)) = \alpha(\sigma_l(l))$*; and*
- $\rho_f(\alpha(l), f) = \alpha(\sigma_f(l, f)),\ \forall f \in F$

for all the values where α and σ are defined.

The next definition uses region consistency to define region soundness. We denote by $|S|_{\,k}$ the cardinality of a set S if it is less or equal to k, and ∞ otherwise.

Definition 2 (Region Abstraction Soundness) *A region abstraction, consisting of abstract stores for procedures and mappings for call sites, is sound if for each activation of each procedure p there exists $\alpha^p : L \rightharpoonup R^p$ such that:*

- *p accesses only locations in dom(α^p);*
- *for each concrete store σ that occurs during the execution of p, α is (σ, ρ)-consistent;*
- *for each call site cs in p that invokes q, if σ^p is the store before (or after) the call and σ^q is the store at the entry (or exit) of q, then $\alpha^p(l) = \mu_{cs}(\alpha^q(l)), \forall l \in L_{\sigma^q} \cap$ dom(α^q).*

We will assume that the points-to analysis is correct, i.e., that computes a sound region abstraction. In other words, region abstraction soundness can be regarded as a specification that describes the interface between pointer and shape analysis.

The definition below expresses the notion of shape abstraction soundness. It describes when a shape abstraction conservatively approximates a concrete memory.

Definition 3 (Shape Abstraction Soundness) *Consider a sound region store ρ for an activation of procedure p and let α be a sound region mapping for that activation of p. Let R be the regions of p, and σ a concrete store during the execution of p. Denote by $L_v = \{l | (l, f) \in$ dom$_f(\sigma)\} \cap$ dom(α) the set of of valid first-field locations. Then $c = (i, (e^+, e^-))$ safely approximates $l \in L_v$, written $c \approx_{\sigma,\rho} l$, if:*

- $\forall r \in R\ .\ i(r) = |\,\{l' \,|\, \alpha(l') = r \ \wedge\ \sigma(l') = l\}\,|_k$
- $\forall e \in e^+\ .\ \langle e, \sigma\rangle \rightarrow_e v\ \Rightarrow\ v = l$
- $\forall e \in e^-\ .\ \langle e, \sigma\rangle \rightarrow_e v\ \Rightarrow\ v \neq l$

A shape abstraction $a \in A$ safely approximates σ, written $a \approx_\rho \sigma$, if : $\forall l \in L_v\ .\ \exists i \in I\ .\ (i, ai) \approx_{\sigma,\rho} l$.

7.2.3 Transfer Function Soundness

To prove that the transfer functions in the shape analysis algorithm are sound, we need to prove that they maintain the sound abstractions of the concrete heaps before and after each statement. The following theorem captures this result.

Theorem 3 (Analysis Soundness) *Let s be a statement in procedure p. Consider two concrete stores σ and σ', a sound region store ρ for p, and a shape abstraction $a \in A$. If $\langle s, \sigma\rangle \rightarrow \sigma'$ and $a \approx_\rho \sigma$, then:*

- $\bigsqcup_{i \in I} [\![s]\!](\rho, (i, ai)) \approx_\rho \sigma'$, *if s is not a malloc; and*
- $([\![s]\!]^{gen}(\rho) \sqcup \bigsqcup_{i \in I} [\![s]\!](\rho, (i, ai))) \approx_\rho \sigma'$, *if s is a malloc.*

The proof of this theorem is fairly involved, so we break it down into a sequence of simpler lemmas that makes it easier to follow the general line of proof.

Lemma 4 (Region Mapping) *Let σ a concrete heap, ρ be a sound region abstraction, and α its region mapping. If $\langle e, \sigma\rangle \rightarrow_l l$ then $\alpha(l) = \mathcal{L}[\![e]\!]\rho$.*

Proof. By structural induction on e.

- Case **null** or $\&e$: cannot evaluate to a location.
- Case x. We get $r = \mathcal{L}[\![x]\!]\rho = \rho(x)$ and $\langle x, \sigma\rangle \rightarrow_l \sigma(x)$, so $l = \sigma(x)$. Then: $\alpha(l) = \alpha(\sigma(x)) = \rho(x) = r$, by (σ, ρ)-consistency of α.
- Case $*e$. We get $r = \mathcal{L}[\![*e]\!]\rho = \rho(r')$, where $r' = \mathcal{L}[\![e]\!]\rho$. Also, $\langle *e, \sigma\rangle \rightarrow_l \sigma(l')$, where $\langle e, \sigma\rangle \rightarrow_l l'$. So $l = \sigma(l')$. By I.H. for e, $\alpha(l') = r'$. Therefore: $\alpha(l) = \alpha(\sigma(l')) = \rho(\alpha(l')) = \rho(r') = r$, by (σ, ρ)-consistency of α.
- Case $e.f$. We get $r = \mathcal{L}[\![e.f]\!]\rho = \rho(r', f)$, where $r' = \mathcal{L}[\![e]\!]\rho$. Also, $\langle e.f, \sigma\rangle \rightarrow_l \sigma(l', f)$, where $\langle e, \sigma\rangle \rightarrow_l l'$. So $l = \sigma(l', f)$. By I.H. for e, $\alpha(l') = r'$. Therefore: $\alpha(l) = \alpha(\sigma(l', f)) = \rho(\alpha(l'), f) = \rho(r', f) = r$, by (σ, ρ)-consistency of α.

□

Lemma 5 (Location Stability) *Consider an assignment $s \equiv e_0 \leftarrow e_1$ and σ, σ' concrete heaps such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If e is such that $\mathcal{S}[\![e]\!]_l(\rho, r_0)$, $\langle e, \sigma\rangle \rightarrow_l l$ and $\langle e, \sigma'\rangle \rightarrow_l l'$, then $l = l'$.*

Proof. By structural induction on e. Let l_0 such that $\langle e_0, \sigma\rangle \rightarrow_l l_0$. Then $\sigma(l) = \sigma'(l), \forall l \neq l_0$. Let α be the region mapping for the sound abstraction ρ. By Lemma 4, $\alpha(l_0) = r_0$.

- Case **null** or $\&e$: cannot evaluate to a location.
- Case x. We have $l = \sigma(x)$ and $l' = \sigma'(x)$. Then $l = l'$ since σ is unchanged for variables.
- Case $*e$. We have $l = \sigma(l_1), l' = \sigma'(l'_1)$, where $\langle e, \sigma\rangle \rightarrow_l l_1$ and $\langle e, \sigma'\rangle \rightarrow_l l'_1$. Also, $\mathcal{S}[\![e]\!]_l(\rho, r_0)$ and $\mathcal{L}[\![e]\!]\rho \neq r_0$. By I.H. for e, $l_1 = l'_1$. By Lemma 4, $\alpha(l_1) = \mathcal{L}[\![e]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l_0 \neq l_1$. Thus, $\sigma(l_1) = \sigma'(l_1)$. Therefore $l = \sigma(l_1) = \sigma'(l_1) = \sigma'(l'_1) = l'$.
- Case $e.f$. We have $l = \sigma(l_1, f), l' = \sigma'(l'_1, f)$, where $\langle e, \sigma\rangle \rightarrow_l l_1$ and $\langle e, \sigma'\rangle \rightarrow_l l'_1$. Also, $\mathcal{S}[\![e]\!]_l(\rho, r_0)$. By I.H. for e, $l_1 = l'_1$. By Lemma 4, $\alpha(l_1) = \mathcal{L}[\![e]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l_0 \neq l_1$. Thus, $\sigma(l_1) = \sigma'(l_1)$. Therefore $l = \sigma(l_1, f) = \sigma'(l_1, f) = \sigma'(l'_1, f) = l'$.

□

Lemma 6 (Value Stability) *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If e is such that $\mathcal{S}[\![e]\!]_v(\rho, r_0)$, $\langle e, \sigma\rangle \rightarrow_e v$ and $\langle e, \sigma'\rangle \rightarrow_e v'$, then $v = v'$.*

Proof. By structural induction on e. Let l_0 such that $\langle e_0, \sigma\rangle \rightarrow_l l_0$. Then $\sigma(l) = \sigma'(l), \forall l \neq l_0$. Let α be the region mapping for ρ. By Lemma 4, $\alpha(l_0) = r_0$.

- Case **null**: trivial, $v = \mathit{null} = v'$.

- Case $\&e$. We get $v = l$ and $v' = l'$ where $\langle e, \sigma\rangle \rightarrow_l l$ and $\langle e, \sigma'\rangle \rightarrow_l l'$. By Lemma 5, $l = l'$.
- Case x. We have $v = \sigma(\sigma(x))$ and $v' = \sigma'(\sigma'(x))$. Also, $\mathcal{L}[\![e]\!]\rho \neq r_0$. Let $l_1 = \sigma(x)$. By Lemma 4, $\alpha(l_1) = \mathcal{L}[\![e]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l_0 \neq l_1$, so $\sigma(l_1) = \sigma'(l_1)$. Since σ is unchanged for variables, $\sigma(x) = \sigma'(x)$, we have: $v = \sigma(l_1) = \sigma'(l_1) = \sigma'(\sigma(x)) = \sigma'(\sigma'(x)) = v'$.
- Case $*e$. We have $v = \sigma(\sigma(l_1)), v' = \sigma'(\sigma'(l'_1))$, where $\langle e, \sigma\rangle \rightarrow_l l_1$ and $\langle e, \sigma'\rangle \rightarrow_l l'_1$. Also, $\mathcal{S}[\![e]\!]_v(\rho, r_0)$ and $\mathcal{L}[\![*e]\!]\rho \neq r_0$.

  We get $\langle e, \sigma\rangle \rightarrow_e \sigma(l_1)$ and $\langle e, \sigma'\rangle \rightarrow_e \sigma'(l'_1)$. By I.H., $\sigma(l_1) = \sigma'(l'_1)$.

  Also $\langle *e, \sigma\rangle \rightarrow_l \sigma(l_1)$. By Lemma 4, $\alpha(\sigma(l_1)) = \mathcal{L}[\![*e]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l_0 \neq \sigma(l_1)$. Thus, $\sigma(\sigma(l_1)) = \sigma'(\sigma(l_1))$. Therefore, $v = v'$.
- Case $e.f$. We have $v = \sigma(\sigma(l_1, f)), v' = \sigma'(\sigma'(l'_1, f))$, where $\langle e, \sigma\rangle \rightarrow_l l_1$ and $\langle e, \sigma'\rangle \rightarrow_l l'_1$. Also, $\mathcal{S}[\![e]\!]_l(\rho, r_0)$ and $\mathcal{L}[\![e.f]\!]\rho \neq r_0$.

  We get $\langle e.f, \sigma\rangle \rightarrow_l \sigma(l_1, f)$ and $\langle e.f, \sigma'\rangle \rightarrow_l \sigma'(l'_1, f)$. Since $\mathcal{S}[\![e]\!]_l(\rho, r_0)$, we have $\mathcal{S}[\![e.f]\!]_l(\rho, r_0)$. By Lemma 5, $\sigma(l_1, f) = \sigma'(l'_1, f)$.

  Again, $\langle e.f, \sigma\rangle \rightarrow_l \sigma(l_1, f)$. By Lemma 4, $\alpha(\sigma(l_1, f)) = \mathcal{L}[\![e.f]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l_0 \neq \sigma(l_1, f)$. Thus, $\sigma(\sigma(l_1, f)) = \sigma'(\sigma(l_1, f))$. Therefore, $v = v'$.

□

Lemma 7 (LHS Location Stability) *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If $\mathcal{S}[\![e_0]\!]_l(\rho, r_0)$, $\langle e_1, \sigma\rangle \rightarrow_e v_1$ and $\langle e_0, \sigma'\rangle \rightarrow_e v'_0$, then $v'_0 = v_1$.*

Proof. Analyze possible forms for e_0 (without induction). Let l_0 such that $\langle e_0, \sigma\rangle \rightarrow_l l_0$. Then $\sigma(l) = \sigma'(l), \forall l \neq l_0$. Let α be the region mapping for ρ. By Lemma 4, $\alpha(l_0) = r_0$.

- Case x. We have $\sigma' = \sigma[\sigma(x) \mapsto v_1]$. Since the value of σ remains unchanged for variables, $\sigma(x) = \sigma'(x)$. We get $v'_0 = \sigma'(\sigma'(x)) = \sigma'(\sigma(x)) = v_1$.
- Case $*e$. Then $\sigma' = \sigma[\sigma(l) \mapsto v_1]$, where $\langle e, \sigma\rangle \rightarrow_l l$. The stability condition implies $\mathcal{S}[\![e]\!]_l(\rho, r_0)$ and $\mathcal{L}[\![e]\!]\rho \neq r_0$. By Lemma 4, $\alpha(l) = \mathcal{L}[\![e]\!]\rho \neq r_0 = \alpha(l_0)$. Hence $l \neq l_0$, so $\sigma(l) = \sigma'(l)$. Since $\mathcal{S}[\![e]\!]_l(\rho, r_0)$, by Lemma 5, $\langle e, \sigma'\rangle \rightarrow_l l$. We get $v'_0 = \sigma'(\sigma'(l)) = \sigma'(\sigma(l)) = v_1$.
- Case $e.f$. Then $\sigma' = \sigma[\sigma(l, f) \mapsto v_1]$, where $\langle e, \sigma\rangle \rightarrow_l l$. The stability condition implies $\mathcal{S}[\![e]\!]_l(\rho, r_0)$. We have $\sigma(l, f) = \sigma'(l, f)$, because the field structure doesn't change. Since $\mathcal{S}[\![e]\!]_l(\rho, r_0)$, by Lemma 5, $\langle e, \sigma'\rangle \rightarrow_l l$. We get $v'_0 = \sigma'(\sigma'(l, f)) = \sigma'(\sigma(l, f)) = v_1$.

□

Lemma 8 (RHS Location Stability) *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If $\mathcal{S}[\![e_1]\!]_l(\rho, r_0)$, $\langle e_1, \sigma\rangle \rightarrow_e v_1$, and $\langle e_1, \sigma'\rangle \rightarrow_e v'_1$, then $v_1 = v'_1$.*

Proof. Consider the case where e_1 is a variable, a dereference, or a field access. Then e_1 evaluates to a location: $\langle e_1, \sigma\rangle \rightarrow_l l_1$ and $\langle e_1, \sigma'\rangle \rightarrow_l l'_1$, with $\sigma(l_1) = v_1$ and $\sigma'(l'_1) = v'_1$. By Lemma 5 (Location Stability), $l_1 = l'_1$. Let l_0 such that $\langle e_0, \sigma\rangle \rightarrow_l l_0$. We have two cases. Either $l_1 = l_0$, in which case $\sigma'(l_1) = \sigma'(l_0) = v_1$; or $l_1 \neq l_0$, in which case $\sigma'(l_1) = \sigma(l_1) = v_1$. In either case, $\sigma'(l_1) = v_1$.

If e_1 is an address-of expression, $e_1 = \&e'_1$, then $v_1 = l_1$, $v'_1 = l'_1$, and $\langle e_1, \sigma\rangle \rightarrow_l l_1$ and $\langle e_1, \sigma'\rangle \rightarrow_l l'_1$. By Lemma 5 (Location Stability), $l_1 = l'_1$. Therefore $v_1 = v'_1$. □

Corollary 2 *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If $\mathcal{S}[\![e_0]\!]_l(\rho, r_0)$, $\mathcal{S}[\![e_1]\!]_l(\rho, r_0)$, $\langle e_0, \sigma'\rangle \rightarrow_e v'_0$, and $\langle e_1, \sigma'\rangle \rightarrow_e v'_1$, then $v'_0 = v'_1$.*

Lemma 9 (Substitution) *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. If $\mathcal{S}[\![e_0]\!]_l(\rho, r_0)$, $\mathcal{S}[\![e_1]\!]_l(\rho, r_0)$, and e is an expression that does not contain address-of subexpressions, $\langle e, \sigma'\rangle \rightarrow_e v$, and $\langle e[*e_0/*e_1], \sigma'\rangle \rightarrow_e v'$, then $v = v'$.*

Proof. We prove by induction that, for any expression e that contains $*e_1$ as subexpression, but does not contain any address-of subexpressions, $\langle e, \sigma' \rangle \rightarrow_e l$ and $\langle e[*e_0/*e_1], \sigma' \rangle \rightarrow_l l_s$ imply $l = l_s$. For the base case, let $\langle *e_1, \sigma' \rangle \rightarrow_l l_1$ and $\langle (*e1)[*e0/*e1], \sigma' \rangle = \langle *e_0, \sigma' \rangle \rightarrow_l l_0$. Then $\langle e_0, \sigma' \rangle \rightarrow_e l_0$ and $\langle e_1, \sigma' \rangle \rightarrow_e l_1$. By the above corollary, $l_0 = l_1$.

For the inductive step, let $*e$ such that $e \neq e_1$, e contains e_1 but no address-of subexpression, $\langle *e, \sigma' \rangle \rightarrow_e l$ and $\langle (*e)[*e_0/*e_1], \sigma' \rangle \rightarrow_e l_s$. Then $l = \sigma'(l')$ and $l_s = \sigma'(l'_s)$, where $\langle e, \sigma' \rangle \rightarrow_e l'$ and $\langle e[*e_0/*e_1], \sigma' \rangle \rightarrow_e l'_s$. By I.H., $l' = l'_s$, hence $l = l_s$.

Similarly, if $e.f$ is such that e contains e_1 but no address-of subexpression, $\langle e.f, \sigma' \rangle \rightarrow_e l$ and $\langle e.f[*e_0/*e_1], \sigma' \rangle \rightarrow_e l_s$, then $l = \sigma'(l', f)$ and $l_s = \sigma'(l'_s, f)$, where $\langle e, \sigma' \rangle \rightarrow_e l'$ and $\langle e[*e_0/*e_1], \sigma' \rangle \rightarrow_e l'_s$. By I.H., $l' = l'_s$, hence $l = l_s$.

Now let values v and v' such that $\langle e, \sigma' \rangle \rightarrow_e v$ and $\langle e[*e_0/*e_1], \sigma' \rangle \rightarrow_e v'$. If e is a dereference or a field access, then it can be evaluated to a location: $\langle e, \sigma' \rangle \rightarrow_e l$ and $\langle e[*e_0/*e_1], \sigma' \rangle \rightarrow_e l'$. Therefore $v = \sigma'(l)$ and $v' = \sigma'(l')$. By the above result, $l = l'$, so $v = v'$. Otherwise, e must be **null**, and in that case $v = \mathit{null} = v'$. □

Lemma 10 (Hit/Miss by Location Stability) *Consider $s \equiv e_0 \leftarrow e_1$ and σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction and $\mathcal{L}[\![e_0]\!]\rho = r_0$. Assume $\langle e_1, \sigma \rangle \rightarrow_e v_1$ and consider e is such that $\mathcal{S}[\![e]\!]_l(\rho, r_0)$, $\langle e, \sigma \rangle \rightarrow_e v$, and $\langle e, \sigma' \rangle \rightarrow_e v'$. If l is such that $v_1 \neq l$ and $v \neq l$, then $v' \neq l$. Similarly, if $v_1 = l$ and $v = l$, then $v' = l$.*

Proof. Let l, l_0 such that $\langle e, \sigma \rangle \rightarrow_l l_e$ and $\langle e_0, \sigma \rangle \rightarrow_l l_0$. If $l_e \neq l_0$, then by Lemma 5, $\langle e, \sigma' \rangle \rightarrow_l l_e$, so $v' = \sigma'(l_e)$. Because $l_e \neq l_0$, $\sigma'(l_e) = \sigma(l_e) = v$. So $v' = v$ and the result follows trivially. If $l_e = l_0$, then $v' = v_1$, and the result follows trivially. □

Lemma 11 (Assign Hit/Miss Sets) *Consider $s \equiv e_0 \leftarrow e_1$, σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$, and let ρ a sound region abstraction. Consider a tracked location l_t, two values v_0, v_1, and booleans b_0, b_1 such that $\langle e_0, \sigma \rangle \rightarrow v_0$, $\langle e_1, \sigma \rangle \rightarrow v_1$, $b_0 \Rightarrow (v_0 = l_t)$, $\neg b_0 \Rightarrow (v_0 \neq l_t)$, $b_1 \Rightarrow (v_1 = l_t)$, and $\neg b_1 \Rightarrow (v_1 \neq l_t)$. If e^+ and e^- are hit and miss sets for l_t in state σ, then the sets e_n^+ and e_n^- computed by the helper function "assign" are hit and miss sets for l_t in state σ'.*

Proof. The computation at lines 10 and 11 yields sound hit/miss sets for l in state σ' by Lemma 6 (Value Stability) and Lemma 10 (Hit/Miss by Location Stability). Next, lines 13 and 14 correctly characterize e_0 as a hit or miss expression for l in state σ' by Lemma 10 (Hit/Miss by Location Stability). Then, the substitutions at lines 16-18 safely add new hit/miss expressions by Corollary 9 (Substitution). Finally, the intersection operations at lines 17 and 18, and the filtering at line 20 do not affect soundness because they remove expressions from these sets. □

Definition 4 *If σ is a concrete store, α a region mapping, r a region, and l a location, then denote by $rc_{\alpha,\sigma}(r, l)$ the cardinality of $\{l' \mid \alpha(l') = r \wedge \sigma(l') = l\}$. The function $cut_k : \mathbb{N} \rightarrow \{0, ..., k, \infty\}$ is defined as: $cut_k(n) = n$ if $n \leq k$, and $cut_k(n) = \infty$ otherwise.*

Lemma 12 (Assign Reference Counts) *Consider $s \equiv e_0 \leftarrow e_1$, and σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, R the regions in ρ, and α its region mapping. Consider a tracked location l_t, two values v_0, v_1, and booleans b_0, b_1 such that $\langle e_0, \sigma \rangle \rightarrow v_0$, $\langle e_1, \sigma \rangle \rightarrow v_1$, $b_0 \Rightarrow (v_0 = l_t)$, $\neg b_0 \Rightarrow (v_0 \neq l_t)$, $b_1 \Rightarrow (v_1 = l_t)$, and $\neg b_1 \Rightarrow (v_1 \neq l_t)$. If i is an index that soundly characterizes the reference counts of r in state σ: $\forall r \in R \,.\, i(r) = cut_k(rc_{\alpha,\sigma}(r, l_t))$, then the set S_i of index values computed by the helper function "assign" is such that $\exists i' \in S_i$ that soundly describes the reference counts of r in state σ': $\forall r \in R \,.\, i'(r) = cut_k(rc_{\alpha,\sigma'}(r, l_t))$.*

Proof. Let $r_0 = \mathcal{L}[\![e_0]\!]\rho$ and l_0 such that $\langle e_0, \sigma \rangle \rightarrow_l l_0$. The assignment updates just the value of l_0: $\sigma'(l) = \sigma(l), \forall l \neq l_0$. Hence $i'(r) = rc_{\alpha,\sigma}(r, l_t) = rc_{\alpha,\sigma'}(r, l_t) = i(r), \forall r \neq r_0$.

If $(b_0 \wedge \neg b_1)$, then $v_0 = l_t$ and $v_1 \neq l_t$. Since $\alpha(l_0) = r_0$, $rc_{\alpha,\sigma'}(r_0, l_t) = rc_{\alpha,\sigma}(r_0, l_t) - 1$. We have two cases:

- If $i(r_0) \leq k$, then $i(r_0) = rc_{\alpha,\sigma}(r_0, l_t)$. Therefore, $rc_{\alpha,\sigma'}(r_0, l_t) = i(r_0) - 1 \leq k$. Hence $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = i(r_0) - 1$ (line 3 in "*assign*").
- Otherwise, if $i(r_0) > k$, we have two subcases. If $rc_{\alpha,\sigma}(r_0, l_t) = k + 1$, then $rc_{\alpha,\sigma'}(r_0, l_t) = k$, so $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = k$. And if $rc_{\alpha,\sigma}(r_0, l_t) \geq k + 2$, then $rc_{\alpha,\sigma'}(r_0, l_t) \geq k + 1$, so $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = \infty$. Both subcases show up in line 4 of "*assign*".

If $(\neg b_0 \wedge b_1)$, then $v_0 \neq l_t$ and $v_1 = l_t$. Therefore, $rc_{\alpha,\sigma'}(r_0, l_t) = rc_{\alpha,\sigma}(r_0, l_t) + 1$. We have two cases:

- If $i(r_0) < k$, then $i(r_0) = rc_{\alpha,\sigma}(r_0, l_t)$. Therefore, $rc_{\alpha,\sigma'}(r_0, l_t) = i(r_0) + 1 \leq k$. Hence $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = i(r_0) + 1$ (line 6).
- Otherwise, if $i(r_0) \geq k$, then $rc_{\alpha,\sigma}(r_0, l_t) \geq k$. Therefore, $rc_{\alpha,\sigma'}(r_0, l_t) \geq k+1$, so $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = \infty$ (line 7).

Finally, if $(b_0 \wedge b_1)$ or $(\neg b_0 \wedge \neg b_1)$, then $v_0 = v_1 = l_t$ or $v_0 \neq l_t \neq v_1$. I both cases, $rc_{\alpha,\sigma'}(r_0, l_t) = rc_{\alpha,\sigma}(r_0, l_t)$. Therefore, $i'(r_0) = cut_k(rc_{\alpha,\sigma'}(r_0, l_t)) = cut_k(rc_{\alpha,\sigma}(r_0, l_t)) = i(r_0)$ (line 8). □

Lemma 13 (Soundness of Helper Function Assign) *Consider $s \equiv e_0 \leftarrow e_1$, and σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, i an index, and e^+ and e^- two sets of expressions. Consider a tracked location l_t, two values v_0, v_1, and booleans b_0, b_1 such that $\langle e_0, \sigma \rangle \rightarrow v_0$, $\langle e_1, \sigma \rangle \rightarrow v_1$, $b_0 \Rightarrow (v_0 = l_t)$, $\neg b_0 \Rightarrow (v_0 \neq l_t)$, $b_1 \Rightarrow (v_1 = l_t)$, and $\neg b_1 \Rightarrow (v_1 \neq l_t)$. If $(i, (e^+, e^-)) \approx_{\sigma,\rho} l_t$ then the abstraction $a \in A$ returned by the helper function assign$(e_0, e_1, \rho, i, e^+, e^-, b_0, b_1)$ is such that $\exists i \in I \,.\, (i, ai) \approx_{\sigma',\rho} l_t$.*

Proof. Follows immediately from Lemma 11 (Assign Hit/Miss Sets) and Lemma 12 (Assign Reference Counts). □

Lemma 14 (Decision Function Well-Formed) *If ρ is a sound region abstraction, then all of the transfer functions in the algorithm maintain the invariant that, for each configuration $c = (i, (e^+, e^-))$ that the analysis computes, the decision function $\mathcal{D}[\![\cdot]\!](\rho, c)$ is well-formed.*

Proof. First, we prove that the transfer function for assignments maintain the following three invariants:

- $e^+ \cap e^- = \emptyset$.

  First, the "*assign*" helper function preserves this invariant. Filtering (lines 10 and 11), substitution (lines 16-18), and filtering again (line 20) trivially maintain the invariant. And lines 13 and 14 are mutually exclusive on b_1, so e_0 is never added to both hit and miss sets.

  Second, the transfer function for assignments preserves the invariant. That is because the helper function is always invoked with disjoint hit and miss sets, "*assign*" preserves this invariant, and so does the merge operation: if $e_1^+ \cap e_1^- = \emptyset$ and $e_2^+ \cap e_2^- = \emptyset$, then $(e_1^+, e_1^-) \sqcup (e_2^+, e_2^-) = (e_1^+ \cap e_2^+, e_1^- \cap e_2^-)$ is such that $(e_1^+ \cap e_2^+) \cap (e_1^- \cap e_2^-) = \emptyset$.

- $e \in e^+ \Rightarrow e \neq$ **null** $\wedge\, e \neq \&x$.

  In the helper function"*assign*", the filtering and substitution operations trivially maintain the invariant. This function may also add the left-hand side expression e_0 to the hit set, but this is an l-value and therefore cannot be **null** or $\&x$.

  The transfer function of assignments may add e_0 or e_1 to the hit set, but only when the decision function yields "+" or "?" for them, which shows that they are different than **null** and $\&x$. Then, "*assign*" maintains the invariant, and so does the the intersection of hit sets when merging abstractions. Therefore, the transfer function for assignments maintains the invariant.

- $e \in e^+ \wedge r = \mathcal{L}[\![e]\!]\rho \Rightarrow i(r) \geq 1$.

  Because $\mathcal{L}[\![e]\!]\rho$ is defined, e is not **null** and contains no address-of expression.

  The main part of this proof is to show that "*assign*" maintains this invariant. Note that $i'(r') = i(r'), \forall r' \neq r_0$, where $r_0 = \mathcal{L}[\![e_0]\!]\rho$, i is the input index, and $i' \in S_i$ is one of the output indices. So "*assign*" is guaranteed to maintain this invariant for all expressions in e^+ and for all $r \neq r_0$. Furthermore, the reference count for r_0 can only decrease to zero when b_1=false. We inspect the body of "*assign*" to see that the invariant is preserved..

  At line 10, the first condition doesn't hold when $r = r_0$ because $\mathcal{L}[\![e]\!]\rho = r_0 \Rightarrow \mathcal{S}[\![e]\!]_v(\rho, r_0)$. Furthermore, if b_1 = false, the second condition is also false. So if $r = r_0$ and b_1=false, the expression e is filtered out of the miss set. Therefore this filtering maintains the invariant.

At line 13, the addition of e_0 to the miss set happens only when b_1=true. Hence, the reference count decrement at lines 3-4 cannot happen. We need to examine the case at line 8, where the index remains unchanged. But since b_1 is true, this can only happen if b_0 = true. It means that the decision function for e_0 evaluates to "+" or "?", so it is guaranteed that $i(r_0) \geq 1$. Hence, adding e_0 to the hit set maintains the invariant.

At line 17, the new expressions added using substitutions maintain the invariant because, for any expression e that contains $*e_1$ and does not contain address-of subexpressions, we have $\mathcal{L}[\![e]\!]\rho = \mathcal{L}[\![e[*e_0/*e_1]]\!]\rho$. The proof is by induction.

If $e = *e_1$ then we want to show $\mathcal{L}[\![*e_1]\!]\rho = \mathcal{L}[\![*e_0]\!]\rho$. Let σ be a program state where the assignment $e_0 \leftarrow e_1$ is being executed. Then $\langle e_0, \sigma\rangle \rightarrow_l l_0$ and $\langle e_1, \sigma\rangle \rightarrow_l l_1$ (since e_1 contains no address-of subexpressions, it can evaluate to a location). If σ' is the state after the assignment, then $\sigma'(l_0) = \sigma(l_1)$. If α is the region mapping, $\alpha(\sigma'(l_0)) = \alpha(\sigma(l_1))$. By (σ, ρ)-consistency, $\rho(\alpha(l_0)) = \rho(\alpha(l_1))$. By Lemma 4, $\alpha(l_0) = \mathcal{L}[\![e_0]\!]\rho$ and $\alpha(l_1) = \mathcal{L}[\![e_1]\!]\rho$. Therefore $\mathcal{L}[\![*e_1]\!]\rho = \rho(\mathcal{L}[\![e_0]\!]) = \rho(\mathcal{L}[\![e_1]\!]\rho) = \mathcal{L}[\![*e_0]\!]\rho$.

The induction step is then straightforward. Consider $e \neq *e_1$ such that $\mathcal{L}[\![e]\!]\rho = \mathcal{L}[\![e[*e_0/*e_1]]\!]\rho$. Then $\mathcal{L}[\![*e]\!]\rho = \rho(\mathcal{L}[\![e]\!]\rho) = \rho(\mathcal{L}[\![e[*e_0/*e_1]]\!]\rho) = \mathcal{L}[\![*(e[*e_0/*e_1])]\!]\rho = \mathcal{L}[\![(*e)[*e_0/*e_1])]\!]\rho$. Similarly, $\mathcal{L}[\![e.f]\!]\rho = \mathcal{L}[\![(e.f)[*e_0/*e_1])]\!]\rho$. This completes the proof by induction.

Since $\mathcal{L}[\![e]\!]\rho = \mathcal{L}[\![e[*e_0/*e_1]]\!]\rho$ and $\mathcal{L}[\![e]\!]\rho \geq 1$ for all e in the hit set, it means that $\mathcal{L}[\![e[*e_0/*e_1]]\!]\rho \geq 1$ for all new expressions $e[*e_0/*e_1]$. This completes the proof that "*assign*" maintains this invariant.

The rest of the proof is similar to that for the previous invariant. The transfer function for assignments may add e_0 or e_1 to the hit set, but in that case the decision function does not yield "-", so their reference counts are at least 1. Next it invokes "*assign*" which maintains the invariant, and then merges abstractions, which clearly maintain the invariant.

Finally, we show that **malloc** and **free** maintain all of the three invariants above. First, the transfer function for **malloc** maintains the invariants because the transfer function for $x =$ **null** maintains them and because none of the expressions $\{(*e).f\}_{f \in F}$ are hit expressions after that assignment (straightforward by inspecting the computation in "*assign*" with b_1 = false and $e_1 =$ **null**). Second, the generating function for **malloc** maintains the invariants by construction. Third, the transfer function for **malloc** maintains the invariants because the transfer function for assignments do so. □

Lemma 15 (Decision Function Soundness) *Let σ be a concrete store, ρ a sound abstract store, $l_t \in L_\sigma$ the tracked location, and c a configuration that safely approximates l_t: $c \approx_{\sigma,\rho} l_t$. If e is an expression such that $tupe, \sigma \rightarrow_e v$, then $\mathcal{D}[\![e]\!](\rho, c) =$ "$+$" implies $v = l_t$, and $\mathcal{D}[\![e]\!](\rho, c) =$ "$-$" implies $v \neq l_t$.*

Proof. Follows immediately from the definition of $\mathcal{D}[\![\cdot]\!]$, $\approx$, and from the fact that *null* and $\&x$ do not evaluate to values in L_σ and $l_t \in L_\sigma$. □

Lemma 16 (Weakening) *Let σ be a concrete store, ρ a sound abstract store, and $l_t \in L_\sigma$ the tracked location. If (i, h) is a configuration such that $(i, h) \approx_{\sigma,\rho} l_t$ and $h \sqsubseteq h'$, then $(i, h') \approx_{\sigma,\rho} l_t$. Similarly, if a is a shape abstraction such that $\exists i' \,.\, (i', ai') \approx_{\sigma,\rho} l_t$ and $a \sqsubseteq a'$, then $\exists i' \,.\, (i', a'i') \approx_{\sigma,\rho}$.*

Proof. Follows immediately from the fact that $h \sqsubseteq h'$ implies that the hit and miss sets in h' are subsets of those in h, and that ordering on shape abstractions is the pointwise ordering. □

Lemma 17 (Soundness of Assignment Transfer Function) *Consider $s \equiv e_0 \leftarrow e_1$, and σ, σ' such that $\langle \sigma, s\rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, i an index, e^+ and e^- sets of expressions, and l_t a tracked location. If $(i, (e^+, e^-)) \approx_{\sigma,\rho} l_t$ then the abstraction $a \in A$ returned by the transfer function $[\![e_0 \leftarrow e_1]\!](\rho, (i, (e^+, e^-)))$ is such that $\exists i \in I \,.\, (i, ai) \approx_{\sigma',\rho} l_t$.*

Proof. Follows from Lemma 14 (Decision Function Well-Formed), Lemma 15 (Decision Function Soundness), Lemma 13 (Assign Helper Function Soundness), Lemma 16 (Weakening), and the fact that: $(i, (e^+, e^-)) \approx_{\sigma,\rho} l_t$ and $e \notin e^+ \cup e^-$ implies that: $(i, (e^+ \cup \{e\}, e^-)) \approx_{\sigma,\rho} l_t$ or $(i, (e^+, e^- \cup \{e\})) \approx_{\sigma,\rho} l_t$. □

Lemma 18 (Soundness of Malloc Transfer Function) *Consider $s \equiv e \leftarrow malloc$, and σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, i an index, e^+ and e^- sets of expressions, and l_t a tracked location. If $(i, (e^+, e^-)) \approx_{\sigma,\rho} l_t$ then the abstraction $a \in A$ returned by the transfer function $[\![e_0 \leftarrow malloc]\!](\rho, (i, (e^+, e^-)))$ is such that $\exists i \in I \,.\, (i, ai) \approx_{\sigma',\rho} l_t$.*

Proof. Let L_f be the set of fresh locations created by malloc and let σ'' such that $\langle e \leftarrow \textbf{null}, \sigma \rangle \rightarrow_s \sigma''$. By Lemma 17 (Soundness of Assignment Transfer Function), we know that $\exists i \in I \,.\, (i, ai) \approx_{\sigma'',\rho} l_t$. We show that $(i, ai) \approx_{\sigma',\rho} l_t$.

We compare the state σ' after malloc, to the state σ'' after the nullification, to determine that they are the same in everything that the tracked location is concerned with. Let l_0 such that $\langle e, \sigma \rangle \rightarrow_l l_0$. The semantic rules show that the two states are identical except for the fresh locations and for the updated location: $\sigma'(l) = \sigma''(l), \forall l \notin L_f \cup \{l_0\}$. Then, the following facts hold:

- $\sigma'(l) = l_t \;\Leftrightarrow\; \sigma''(l) = l_t$. This is because $\sigma'(l) \neq l_t$ for $l \in L_f \cup \{l_0\}$, the newly allocated structure are null, so they do not reference the tracked location, and l_t is different than the new locations, so l_0 does not reference it; and $\sigma''(l)$ is not defined for $l \in L_f$, and $\sigma''(l_0) = \mathit{null} \neq l_t$;
- $\langle e', \sigma' \rangle \rightarrow_e l \;\Leftrightarrow\; \langle e', \sigma'' \rangle \rightarrow_e l, \forall e' \in E_p, l \notin L_f$ (straightforward proof by structural induction on e').

These two facts show that $(i, ai) \approx_{\sigma',\rho} l_t$.

Finally, we show that the field expressions are miss expressions. If $\mathcal{S}[\![e]\!]_l(\rho, \mathcal{L}[\![e]\!]\rho)$, by Lemma 5 (Location Stability) we have $\langle e, \sigma' \rangle \rightarrow_l l_0$. By the semantic rules, $\langle (*e).f, \sigma' \rangle \rightarrow_e \mathit{null} \neq l_t$. □

Lemma 19 (Soundness of Malloc generating Function) *Consider $s \equiv e \leftarrow malloc$, and σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, and l_t the first field of the newly created structure. Then $[\![\textbf{malloc}(e)]\!]^{gen}(\rho) \approx_{\sigma',\rho} l_t$.*

Proof. Let l_0 such that $\langle e, \sigma \rangle \rightarrow_l l_0$. By the semantics of malloc, $\sigma'(l_0) = l_t$. Furthermore, since l_t is fresh, $\sigma'(l) \neq l_t, \forall l \neq l_0$. Let α be the region mapping. By Lemma 4 (Region Mapping), $\alpha(l_0) = \mathcal{L}[\![e]\!]\rho = r$. Therefore, $rc_{\alpha,\sigma'}(r, l_t) = 1$ and $rc_{\alpha,\sigma'}(r', l_t) = 0, \forall r' \neq r$. Hence, the index value $i = \lambda r' \,.$ if $(r' = r)$ then 1 else 0 computed by the malloc generating function safely approximates l_t.

Furthermore, if $\mathcal{S}[\![e]\!]_l(\rho, r)$, by Lemma 5 (Location Stability) expression e evaluates to the same location l_0 after the assignment: $\langle e, \sigma' \rangle \rightarrow_l l_0$. Since $\sigma'(l_0) = l_t$, it means that $\langle e, \sigma' \rangle \rightarrow_e l_t$, hence e is a hit expression. And $\langle *e, \sigma' \rangle \rightarrow_l l_t$, so expressions $(*e).f$ represent the fields of the new structure. By the semantic rules, these field are initialized to null: $\langle (*e).f, \sigma' \rangle \rightarrow_e \mathit{null} \neq l_t$, so they are all miss expressions. Finally, for the case when e is not location-stable, $\emptyset$ trivially represents a safe hit or miss set. Hence, the secondary values computed by the generating function are sound. □

Lemma 20 (Soundness of Free Transfer Function) *Consider $s \equiv e \leftarrow malloc$, and σ, σ' such that $\langle \sigma, s \rangle \rightarrow_s \sigma'$. Let ρ be a sound region abstraction, i an index, e^+ and e^- sets of expressions, and l_t a tracked location. If $(i, (e^+, e^-)) \approx_{\sigma,\rho} l_t$ then the abstraction $a \in A$ returned by the transfer function $[\![\textbf{free}(e)]\!](\rho, (i, (e^+, e^-)))$ is such that $\exists i \in I \,.\, (i, ai) \approx_{\sigma',\rho} l_t$.*

Proof. Let L_f the set of locations representing fields in the deallocated structure. Let σ'' be the state after executing the sequence of statements $t = *e; t.f_1 \leftarrow \textbf{null}; \ldots; t.f_m \leftarrow \textbf{null}; t \leftarrow \textbf{null}$. By Lemma 17 (Soundness of Assignment Transfer Function), we know that $\exists i \in I \,.\, (i, ai) \approx_{\sigma'',\rho} l_t$. We show that $(i, ai) \approx_{\sigma',\rho} l_t$.

We compare the state σ' after free, to the state σ'' after the sequence of field nullfications, to determine that they are identical in all that matters to the tracked location. Let l_0 such that $\langle e, \sigma \rangle \rightarrow_l l_0$ (l_0 is the first field in the deallocated structure). The semantic rules show that the two states are identical except for the fields of the deallocated structure: $\sigma'(l) = \sigma''(l), \forall l \notin L_f$. The following facts hold:

- $\sigma'(l) = l_t \;\Leftrightarrow\; \sigma''(l) = l_t$. This is because $\sigma'(l)$ is not defined for the deallocated locations $l \in L_f$, as shown by the semantics of free; and $\sigma''(l) \neq l_t$ for all $l \in L_f$, because of field nullifications;
- $\langle e', \sigma' \rangle \rightarrow_e l \;\Leftrightarrow\; \langle e', \sigma'' \rangle \rightarrow_e l, \forall e' \in E_p, l \notin L_f$ (straightforward proof by structural induction on e').

These two facts show that $(i, ai) \approx_{\sigma',\rho} l_t$. □

The proof of Theorem 3 (Soundness) follows from Lemmas 17, 18, 19, and 20.

Program	OpenSSH	OpenSSL	BinUtils
Size (KLOC)	18.6	25.6	24.4
Alloc. Sites	41	31	125
Analyzed	41	28	115
Total Time (sec)	45s	22s	44s
Region Analysis	16s	13s	6s
Shape Analysis	29s	9s	38s
Reported	26	13	58
Real bugs	10	4	24

Figure 16: Experimental Results

8 Experimental Results

We have implemented all of the algorithms presented in the paper, including the points-to analysis, shape analysis, and extensions for detecting memory errors, using the SUIF Compiler infrastructure [1]. We have tested our prototype implementation on several small examples and on a few larger C programs. To experiment with the demand-driven and incremental approach, our system analyzes one allocation site at a time, reusing existing results as new sites are explored. We have extended the analysis to handle various C constructs, such as arrays, pointer arithmetic, casts, unions, and others. The points-to analysis makes the usual assumptions, e.g., that array accesses and pointer arithmetic do not violate the array or structure bounds. Our shape analysis is guaranteed to be sound as long as the underlying region abstraction computed by the points-to analysis is sound.

8.1 Core List Algorithms

We have successfully tested the analysis against a variety of singly-linked list algorithms: insert, append, delete, splice, reverse, insertion sort, and in-place quick sort. The analysis can determine that these algorithms preserve acyclic list shape. However, the analysis is sensitive to the way most of these algorithms are written: identifying the correct shape depends on certain expressions to be present in the hit or miss set. Otherwise, the analysis would require variable equality information, as described for the splice example in Section 6. We have also experimented with examples that maintain structures with more complex invariants, such as doubly-linked lists and trees with parent pointers. The analysis breaks down on such examples because its abstraction cannot capture their structural invariants.

8.2 Experience on Larger Programs

We have also used the analysis to detect memory leaks in portions of three popular C programs: OpenSSH, OpenSSL, and BinUtils. The results of these experiments are shown in Figure 16. In total, we analyzed 184 dynamic allocation sites in about 70 KLOC, taking less than two minutes. This produced 97 warnings, 42 of which were actual memory leaks (a few warnings were for double frees and accesses to deallocated memory, all of which were false). Warnings are given on a per-region basis, and consist of a collection of execution traces leading to errors on cells in that region. These traces are produced by building a graph through the configuration space (as in Figure 5) and tracing backwards from the leak point to the allocation site; we find the traces to be extremely helpful in identifying whether a warning is legitimate. Most leaks were caused as functions failed without cleaning up local resources.

The limitations discovered for the smaller examples affected our results for these programs in two main ways. First, we were unable to completely analyze several allocation sites in the code from OpenSSL and BinUtils. When linked structures that either have complex structural invariants or are manipulated in certain (fairly standard) ways, we quickly lose precision. Moreover, this leads to an exponential blowup in the number of feasible configurations, and effectively grinds the analysis to a halt. We avoid this issue using incremental analysis: we explore one allocation site at a time and introduce a cutoff for the amount of exploration to be performed from any given allocation site, by limiting the size of the worklist in the algorithm.

Second, the imprecision in the analysis impacts the rate of false positives. However, we consider that the number of warnings is low, given the program sizes and the fact that we use sound analysis techniques. Many false positives were due to a lack of extra path or structural information that could have been used to rule out the error trace. For instance, we are unable to track the correlation between the references to the tracked cell and the values of various error codes in the program.

Our results suggest that the local reasoning approach is both sufficiently precise to accurately analyze a large class of list manipulation algorithms, and sufficiently lightweight to scale to larger programs. However, they also show that our abstraction is not robust enough to handle many different implementations of particular algorithms, and that this limitation impacts our ability to analyze larger programs.

9 Related Work

There has been significant work in the past decade in the area of shape analysis [20]. Early approaches to shape analysis have proposed the use of path matrices and other matrices that capture heap reachability information [10, 9, 6]. In particular, Ghiya and Hendren [6] present an inter-procedural shape analysis that uses boolean matrices to identify trees, dags, or cyclic graphs. Their implemented system has been successful at analyzing programs of up to 3 KLOC.

Sagiv et. al. present a shape analysis based on abstract interpretation [15], using shape graphs to model heap structures. They introduce materialization and summarization as key techniques for the precise computation of shapes. Role analysis [11] uses a similar shape graph abstraction to check language specifications for shapes and heap effects. None of these analyses have been implemented.

In subsequent work, Sagiv et. al. have proposed the use of 3-valued logic for solving shape analysis problems [17]. They encode shape graphs as 3-valued structures and use a focus operation to accurately compute shapes when the analysis encounters unknown (1/2) logic values. Our bifurcation technique is similar to their focus operation, but applies to a different analysis abstraction. The 3-valued logic approach has been implemented in the TVLA system [13] and has been used to verify various safety properties of programs with linked structures [12, 21], including the absence of memory errors in linked list manipulations [4].

Yahav and Ramalingam [22] have recently proposed heterogeneous heap abstractions as a means of speeding up analyses in TVLA. Their framework constructs a heap abstraction that models different parts of the heap with different degrees of precision, and keeps precise information just for the relevant portion of the heap. However, their abstraction still describes the entire heap, not individual heap locations. In contrast, our approach is homogeneous, precisely abstracts the entire heap, but models each heap location independently of the rest of the heap. Their approach has enabled TVLA to analyze programs of up to 1.3 KLOC.

Our pointer analysis is similar to the flow-insensitive, but context-sensitive algorithm proposed by Liang and Harrold [14]. Their results indicate that this analysis scales well to large programs.

Demand-driven and incremental algorithm have been proposed in the area of pointer analysis. Heintze and Tardieu [8] describe a technique to answer aliasing queries by exploring the minimal set of points-to constraints that yields the desired answer. Vivien and Rinard [19] present an incremental points-to analysis that gradually explores more code to refine the points-to information. Our notion of demand-driven and incremental analysis is different – it refers to the number of explored allocation sites.

Heine and Lam present a leak detector tool for C and C++ [7] programs. They use a notion of ownership to describe variables responsible for freeing heap cells and formulate the analysis as an ownership constraint system. Their tool is able to detect leaks and double frees, but cannot detect accesses through dangling pointers. Das et. al. present ESP [3], a path-sensitive tool for verifying state machine properties. ESP also uses a vertical decomposition, by running a inclusion-based pointer analysis first, and building the machine state abstraction on top of it. We borrow from ESP the notion of index values in the abstraction to model critical information that the analysis must not merge at join points. However, ESP is designed to analyze properties described by state machines, not shapes in recursive structures. Other existing error-detection tools, such as Metal [5] and Prefix[2], use unsound techniques to limit the number of false positives or to avoid fixed-point computations.

10 Conclusions

We have presented a new approach to shape analysis where the compiler uses local reasoning about the state of one single heap location, as opposed to global reasoning about entire heap abstractions. We have showed that this approach makes it possible to develop efficient intra-procedural analysis algorithms, context-sensitive inter-procedural algorithms, demand-driven and incremental analyses, and can enable the detection of memory errors with low false positive rates. We believe that the proposed approach brings shape analysis a step closer to being successful for real-world programs.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[2] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, August 2000.

[3] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[4] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proceedings of the 7th International Static Analysis Symposium*, Santa Barbara, CA, July 2000.

[5] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[6] R. Ghiya and L. Hendren. Is is a tree, a DAG or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[7] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the SIGPLAN '03 Conference on Program Language Design and Implementation*, San Diego, CA, June 2003.

[8] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[9] L. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[10] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[11] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 2002.

[12] T. Lev-ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *2000 International Symposium on Software Testing and Analysis*, August 2000.

[13] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the 7th International Static Analysis Symposium*, Santa Barbara, CA, July 2000.

[14] D. Liang and M.J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the ACM SIGSOFT '99 Symposium on the Foundations of Software Engineering*, Toulouse,France, September 1999.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[16] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1999.

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

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[19] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[20] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proceedings of the 2000 International Conference on Compiler Construction*, Berlin, Germany, April 2000.

[21] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the 28th Annual ACM Symposium on the Principles of Programming Languages*, London, UK, January 2001.

[22] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the SIGPLAN '04 Conference on Program Language Design and Implementation*, Washington, DC, June 2004.