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. 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 smaller granularity of the abstraction; and 3) makes it easier to develop context-sensitive, demand-driven, and incremental shape analyses.

We also show that the analysis can be used to enable the static detection of memory errors in programs with explicit deallocation. We have built a prototype tool that detects memory leaks and accesses through dangling pointers in C programs. The experiments indicate that the analysis is sufficiently precise to detect errors with low false positive rates; and is sufficiently lightweight to scale to larger programs. For a set of three popular C programs, the tool has analyzed about 70K lines of code in less than 2 minutes and has produced 97 warnings, 38 of which were actual errors.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.4 [**Programming Languages**]: Compilers—*Memory Management*

General Terms

Algorithms, Languages, Verification

Keywords

Shape analysis, static error detection, memory leaks, memory management

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

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 [34]. 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 [18, 17, 11], shape graphs [29, 21], and, more recently, three-valued logic structures [31]; all of these characterize the entire heap at once. Although shape analyses have been successful at verifying complex heap manipulations, they have had limited success at being practical for larger 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;
- *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.

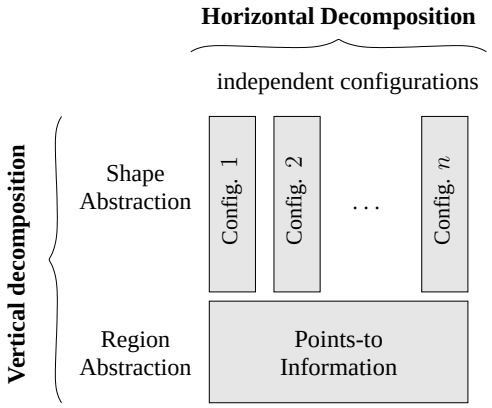


Figure 1: Decomposition of the memory abstraction

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 [30], nor any complex mechanisms to merge entire abstractions for a more compact representation [29]. 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 extensions of the analysis that enable the static detection of 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 standard list manipulations; and that it can enable the detection of memory errors with low false positive rates.

This paper makes the following contributions:

- **Memory Abstraction:** It proposes a novel memory abstraction that builds the precise shape abstraction on top of a region points-to abstraction; it further decomposes the shape abstraction 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;
- **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 3 introduces a simple language, and Sections 4 and 5 describe the algorithm in the context of this simple language. Next, Section 6 presents extensions to the algorithm. Section 7 discusses limitations. 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 argument lists `x` and `y`, splices `x` into `y`, and returns the resulting list. The code assumes that input list `x` is not longer than `y`. 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 `z`. 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 list `x`, it terminates and the procedure returns the list pointed to by `z`.

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`, `z`, and `t` 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 left 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, Z, and T are the regions containing variables `x`, `y`, `z`, and `t`, 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.

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 abstraction has 4. Each configuration characterizes the state of the tracked location and consist of: a) reference counts from each region to the tracked location, shown in superscripts; b) program expressions that definitely reference the tracked location (*hit* expressions); and c) 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,

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

```
 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 *z = 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 z;
17:  }
```

Figure 2: Example program: splicing lists

one incoming reference from region L, and is referenced by `t`, but 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 readability, 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*≡ `t->n` and *yn*≡ `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 Z^1 and L^1, and shows that at the end of the procedure the returned value `z` 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 the configuration Y^1Z^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.

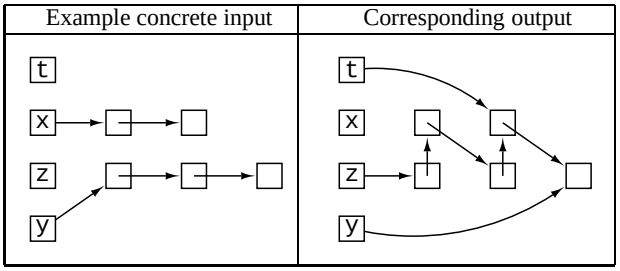


Figure 3: Example concrete memories

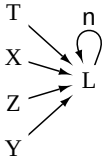
Region Points-to Component	Shape Component: Configurations for input memory	Shape Component: Configurations for output memory
	$(X^1, \{x\}, \emptyset)$ $(Y^1, \{y\}, \emptyset)$ $(L^1, \emptyset, \emptyset)$	$(Z^1, \{z\}, \emptyset)$ $(T^1\,L^1, \{t\}, \emptyset)$ $(Y^1\,L^1, \{y\}, \emptyset)$ $(L^1, \emptyset, \emptyset)$

Figure 4: Memory abstraction

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

Analyzing the behavior of `splice` for cyclic or shared input lists provides key insights about why the local reasoning about the single location works. The left part of Figure 6 shows an input store where the list pointed to by `y` contains a cycle; the right part shows the resulting memory after running `splice` with this input.

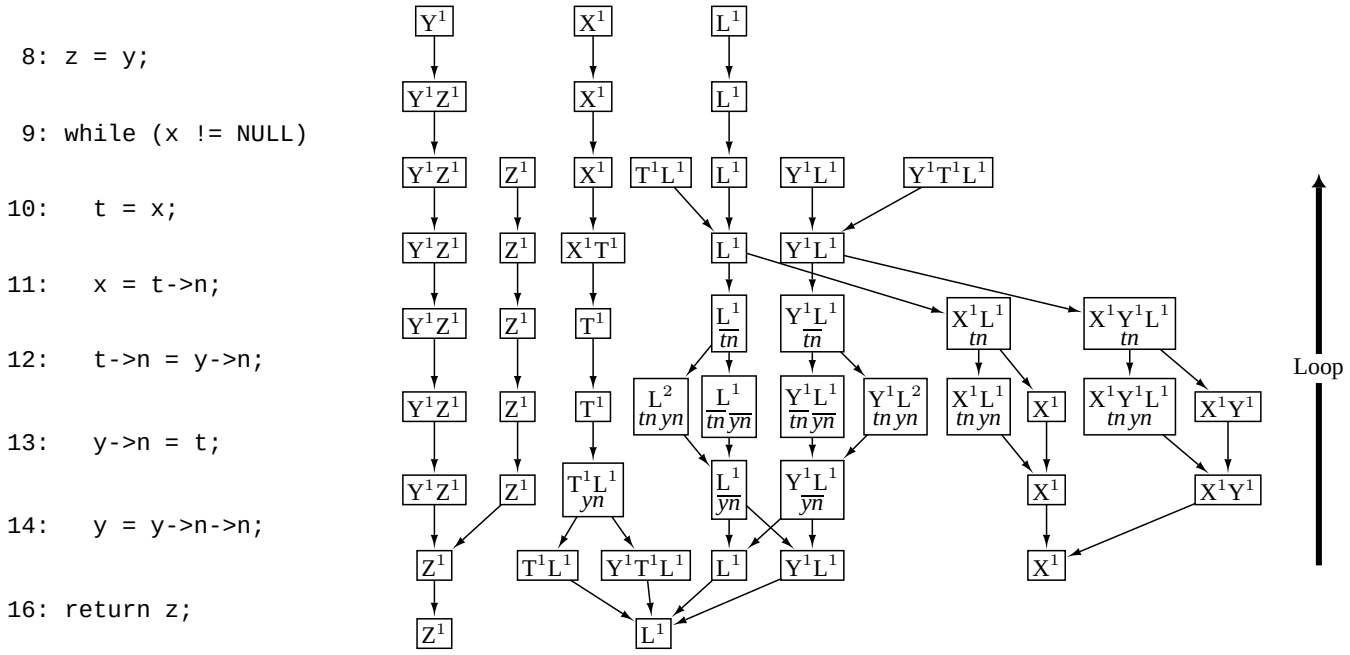


Figure 5: Shape analysis results for `splice`. Boxes represent configurations and edges show how the state of the tracked location changes during the execution. We only show field access expressions in the hit and miss sets. We use the abbreviations: *tn*≡ `t->n` and *yn*≡ `y->n`, and we indicate miss expressions using overlines. For readability, back edges from configurations at the end of the loop to the corresponding configurations at the beginning of the loop are omitted.

A closer look at the input structure reveals that the cycle is caused by the presence of one shared cell with two incoming references, the shaded cell. If we inspect the output structure we see that it is also a cyclic list. The interesting fact is not that `splice` yields a cyclic output given a cyclic input; but rather that the shaded cell that causes the input cycle is exactly the same cell that causes the cycle in the output. Therefore, reasoning about cycles requires reasoning just about this particular cell. All of the other cells in this structure behave as in the acyclic case.

To build the abstraction for the cyclic input from Figure 6, we can use the previous abstraction, and augment it with one additional configuration L^2 to describe the cell in question. The analysis of the abstraction for the cyclic input will yield a configuration graph similar to the one from Figure 5, but augmented with additional paths that originate at the L^2 configuration. These paths describe the state of the shared cell through the program. One can examine the input-output relationships in the result graph, and identify that the shared cell in the input (L^2) may remain shared, but all of the non-shared cells in the input (X^1, Y^1, and L^1) will remain non-shared in the output.

In fact, the analysis of the cyclic case can reuse all of the analysis result from the acyclic case. This is possible because the analysis of each configuration in Figure 5 reasons only about one 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. Similar situations arise for inputs that have 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 cyclic context 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.

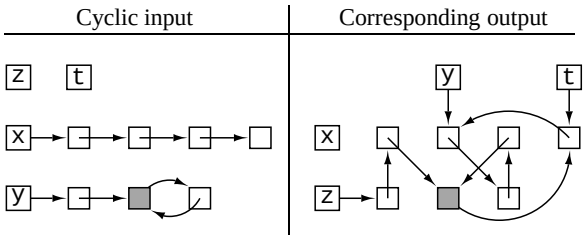


Figure 6: Example cyclic memory

3. A SIMPLE LANGUAGE

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 *prog* maps each procedure to a pair containing its formal parameters and its body. 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 $*e$ al-

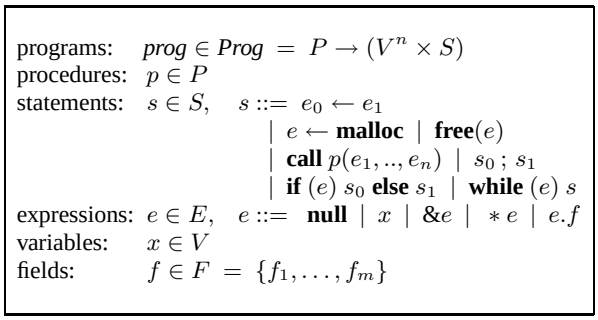


Figure 7: Source language

ways represents the memory location pointed to by e (not the whole structure when e points to a structure). C expressions of the form `e->f` are represented in our language 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 the stack locations for variables, and all of the heap locations created during program execution. Let V be the set of variables in 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 $\mathit{range}(\sigma_v)$ represents stack-allocated memory locations. The set $L_\sigma = \mathit{dom}(\sigma_l) - \mathit{range}(\sigma_v)$ represents the currently allocated heap locations. A stack or heap location l contains a dangling pointer if $\sigma_l(l) \notin \mathit{dom}(\sigma_l) \cup \{\mathit{null}\}$. 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. The Appendix describes the formal semantics of the language using a relation $\langle e, \sigma\rangle \to_l l$ that evaluates expression e to its l-value; a relation $\langle e, \sigma\rangle \to_v v$ that evaluates e to its r-value; a relation $\langle s, \sigma\rangle \to_s \sigma'$ for statements. In this paper, we refer to the l-value of an expression e as the *location of e*, and to the r-value of e as the *value of e*.

We formulate the proposed analysis algorithm in the context of this language. The overall algorithm first performs a region analysis, and then runs the shape analysis. The following two sections describe each of these analyses in turn.

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. In general, this can be achieved using any pointer analysis algorithm; our shape analysis is independent of the particular pointer analysis being used. However, we use a flow-insensitive and context-sensitive pointer analysis similar to [24] and [10]. Such algorithms seem a good match for our problem because of two reasons. First, they are efficient in practice, due to flow-insensitivity. Second, they are precise enough to distinguish between different heap structures allocated at the same site, due to context-sensitivity. Using such an algorithm, the analysis result is a points-to graph for each procedure; nodes in these graphs represent memory regions, and edges model points-to relations between regions. Most important in our system is to characterize the computed region result, thus describing the interface between region and shape analysis. 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; the partial map $\rho^p_r : R^p \rightharpoonup R^p$ models points-to relations between regions; and $\rho^p_f : (R^p \times F) \rightharpoonup R^p$ 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 5.3. 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 flow-insensitive and context-sensitive analysis that computes the region points-to abstraction. First, the algorithm performs an intra-procedural, unification-based analysis [32] to build a points-to graph for each function. Then, it performs an inter-procedural analysis and propagates the aliasing information between different functions at each call site. The algorithm uses a two-phase approach similar to [24]: a bottom-up pass through the functions in the call graph propagates the aliasing information from callees to callers; and, a top-down phase propagates the information from callers to callees.

Note that the region partitioning that other kinds of pointer analyses produce could be modeled in a similar way. For a context-insensitive pointer analysis, each call site mapping μ_{cs} is the identity function. And in the case of a flow-sensitive approach, the analysis result is not just one abstract store ρ per function, but a set of abstract stores, one for each program point.

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 role of the shape abstraction is to make the points-to information more accurate. Roughly speaking, the region abstraction provides "may" points-to relations between memory locations, and the shape abstraction augments it with "must" points-to information about each heap cell.

The shape abstraction is based on the notion of *configurations*. 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. Each configuration keeps track of: reference counts from each region to the tracked location; expressions that definitely reference the tracked location (hit expressions); and expressions that definitely don't reference the location (miss expressions).

Formally, we describe configurations using 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 $\perp$:

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

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 $H \cup \{\perp\}$; one can infer which operator is being used from its context. 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.

5.2.1 Dataflow Equations

We formulate the analysis of each function in the program as a dataflow analysis that computes a shape abstraction $a \in A$ at each program point in the function. The algorithm differs from standard approaches in two ways. First, it uses a system of dataflow equations and a corresponding worklist algorithm that operate at the granularity of individual configurations, rather than entire heap abstractions (i.e., sets of configurations). Second, the dataflow information is being initialized not only at the entry point in the control-flow, but also at each allocation site, where the analysis produces a new configuration for the newly created memory 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 beginning of the currently analyzed function (i.e., assignments reachable from the function entry point without going through other assignments), and let *pred* and *succ* map assignments to their predecessor or successor assignments in the control flow (these can be easily computed from the syntactic structure of control statements).

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

$$[\textsc{Join}] \qquad \mathit{Res}(\bullet s)\, i = \bigsqcup_{s' \in \mathit{pred}(s)} \mathit{Res}(s'\bullet)\, i$$

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

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

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

Figure 8: Intra-procedural dataflow equations.

We model the analysis of individual assignments using transfer functions that operate at the level 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. Figure 8 shows the dataflow equations that describe *Res*. In this figure, ρ is the region store for the currently analyzed function and $a_0 \in A$ is the boundary dataflow information at the function entry point. Equations [JOIN], [TRANSF], and [ENTRY] are standard dataflow equations, but are being expressed such that they expose individual configurations and their dependencies. Equation [ALLOC] indicates that the analysis always generates a configuration for the new location, regardless of the abstraction before the allocation statement.

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 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.

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 location of e. The function is not defined for expressions that do not represent l-values ($\&e$ and **null**).

```
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.

If $\rho = (\rho_v, \rho_r, \rho_f)$, then:

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

Second, we define a *decision evaluation function* $\mathcal{D}[\![e]\!]$ that takes a store ρ and a configuration c, uses this information 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 determines 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, the location of e is the same 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. It uses a similar reasoning as above to determine that e_0 is a hit (or miss) expression only if it is location-stable and the written value hits (or misses) the tracked location. 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, 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 $\perp$.

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

```
1   r = L[[e0]](ρ)
2   if (b0 ∧ ¬b1) 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 (¬b0 ∧ b1) 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) ∧ b1)}
11  e_n^- = {e ∈ e^- | S[[e]]_v(ρ, r) ∨ (S[[e]]_l(ρ, r) ∧ ¬b1)}
12
13  if (S[[e0]]_l(ρ, r) ∧ b1)   then e_n^+ ∪= {e0}
14  if (S[[e0]]_l(ρ, r) ∧ ¬b1) then e_n^- ∪= {e0}
15
16  if (S[[e0]]_l(ρ, r) ∧ S[[e1]]_l(ρ, r)) then
17      e_n^+ ∪= ( e_n^+[*e0/*e1] ∩ E_p' )
18      e_n^- ∪= ( e_n^-[*e0/*e1] ∩ 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*.

Note that bifurcation can produce up to four cases and each case may yield up to two configurations. However, 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 $[\![e \leftarrow \textbf{malloc}]\!]$ works as follows. First, the effect of the allocation 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). Second, since the contents of the fresh location are not initialized, its fields become miss expressions provided that e is location-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.

5.2.5 Conditional Branches

The analysis extracts useful information from test conditions in if and while statements. On the branch where the tested expression

$[\![e \leftarrow \textbf{malloc}]\!](\rho, c)$:
  $a = [\![e \leftarrow \textbf{null}]\!](\rho, c)$
  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)

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

Figure 12: Analysis of malloc and free.

e is null, the analysis determines that e misses the tracked location. 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 $\perp$.

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 Formal Framework

This section summarizes the formal results for the intra-procedural analysis. The proofs for all of the theorems below are available in a companion technical report [12]. The first two theorems provide correctness and termination guarantees for the worklist algorithm.

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.*

THEOREM 2 (MONOTONICITY). *The transfer functions $[\![s]\!]$ for assignments, malloc, and free are monotonic in the secondary value: if $h \sqsubseteq h'$, then $[\![s]\!](\rho, (i, h)) \sqsubseteq [\![s]\!](\rho, (i, h'))$ for all s, i, ρ.*

COROLLARY 1 (TERMINATION). *The worklist algorithm from Figure 9 is guaranteed to terminate.*

We next give soundness conditions and results. 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); $\alpha(\sigma_v(x)) = \rho_v(x)$, $\forall x \in V$; $\rho_r(\alpha(l)) = \alpha(\sigma_l(l))$; and $\rho_f(\alpha(l), f) = \alpha(\sigma_f(l, f))$ for all the locations l and fields f where α and σ are defined. The definitions below give soundness conditions for our abstractions. We denote by $|S|_k$ the cardinality of a set S if it is less or equal to k, and ∞ otherwise.

DEFINITION 1 (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 a region mapping $\alpha^p : L \rightharpoonup R^p$ such that:*

- *p accesses only locations in dom(α^p);*
- *for each concrete store σ that occurs during the execution of p, α^p is (σ, ρ^p)-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).*

DEFINITION 2 (SHAPE ABSTRACTION SOUNDNESS). *Let ρ be a sound region store for an activation of procedure p, and let α be the corresponding 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:*

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

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

The soundness theorem states that shape analysis is sound as long as the underlying region abstraction is sound.

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.*

5.4 Inter-Procedural Analysis

We formulate the inter-procedural algorithm as a context-sensitive analysis that distinguishes between different calling contexts of the same procedure. Again, we take advantage of our 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 (such as the one in Figure 5) that describes 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). However, we do not need to build the configuration graph explicitly; instead, we tag 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. Formally, we extend the index domain in the analysis with the index at entry:

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

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. For this, the analysis uses two

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

[IN] :

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

[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.

```
WORKLIST(DataflowInfo a_0)
15  ...
16  while (W is not empty)
17      remove some (s, i) from W
18      if s ∈ S_call then
19          let s_e = entry(tgt(s)), s_x = exit(tgt(s))
20          Res(•s_e) ⊔= [[s]]^in(ρ, μ_s, Res(•s, i))
21          for each i' s.th. Res(•s_e)i' has changed
22              W ∪= {(s_e, i')}
23          for each i' s.th. Res(s_x•)i' ≠ ⊥
24              Res(s•) ⊔=
25                  [[s]]^out(ρ, μ_s, Res(•s, i), Res(•s_x, i'))
26      else if s ∈ S_exit then
27          for s_c, i' s.th. tgt(s_c) = fn(s), Res(•s_c)i' ≠ ⊥
28              Res(s_c•) ⊔=
29                  [[s_c]]^out(ρ, μ_{s_c}, Res(•s_c, i'), Res(•s, i))
30      else
31          Res(s•) ⊔= [[s]](ρ, Res(•s, i))
32
33      for each s', i' s.th. Res(s'•)i' has changed
34          for s'' ∈ succ(s')
35              Res(•s'')i' ⊔= Res(s'•)i'
36              W ∪= {(s'', i')}
```

Figure 14: Inter-procedural worklist algorithm

transfer 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 the region store ρ for the current procedure and the region mapping μ_s at the call site where the information must be propagated.

We express the inter-procedural analysis using a set of dataflow equations that augments the intra-procedural equations from Figure 8 with the two additional equations from 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. Figure 14 shows the main loop of the inter-procedural worklist algorithm; the remainder of the algorithm is unchanged. The algorithm propagates output configurations to the callers when it encounters new exit configurations in the callee, or when it discovers new input contexts in the caller.

$$\begin{array}{l}
\llbracket \textbf{call } q(e_1, .., e_n) \rrbracket^{in}(\rho, \mu, (i, h)) : \\
\quad a = \llbracket p_1 \leftarrow e_1 \rrbracket \ldots \llbracket p_n \leftarrow e_n \rrbracket(\rho, \{(i, h)\}) \\
\quad \textbf{return } \bigsqcup_{\{i \mid a\ i \neq \bot\}} S_{in}(\rho, \mu, (i, a\ i)) \\
\\
\text{where } S_{in}(\rho, \mu, ((i^c, i^e), (e^+, e^-))) : \\
\quad e_q^+ = \{e \in e^+ \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}\llbracket e \rrbracket_v(\rho, r))\} \\
\quad e_q^- = \{e \in e^- \mid \forall r \in \mathit{range}(\mu)\ .\ \mathcal{S}\llbracket e \rrbracket_v(\rho, r))\} \\
\quad i_q^e = \lambda r \in R^q\ .\ i^c(\mu\, r) \\
\quad \textbf{return } ((i_q^e, i_q^e), (e^+ - e_q^+, e^- - e_q^-))
\end{array}$$

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

Figure 15: Inter-procedural transfer functions.

Figure 15 shows the input and output transfer functions. The entry transfer function accomplishes two things. First, it performs the assignments of actual to formal arguments, which may generate new references to the tracked location. Second, it adjusts the regions of existing references according to the call site 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.

6. EXTENSIONS

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

6.1 On-Demand 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: we just apply the dataflow equation [ALLOC] from Figure 8 to the selected allocation sites. The effect is that the dataflow information gets 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 analysis 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 algorithm. Key to the incremental computation is that our abstraction based on configurations allows the analysis to reuse results from previously analyzed allocation sites. This is possible both at the

intra-procedural level, when the new configurations match existing configurations at particular program points; and at the inter-procedural level, when new calling contexts match existing contexts.

6.2 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 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 additions, 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 not 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.

7. LIMITATIONS

The analysis algorithm presented in this paper has the following limitations:

- *Spurious configurations*. The analysis may generate spurious configurations, but still determine the correct final shape in spite of this imprecision. In the example from Figure 5, configurations $\mathrm{Y}^1\mathrm{T}^1\mathrm{L}^1$ and $\mathrm{X}^1\mathrm{Y}^1\mathrm{L}^1$ are spurious. Ruling such cases requires more complex decision functions $\mathcal{D}[\![\cdot]\!]$.

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

- *Robustness*. The analysis may not identify the correct shape when the same program is written in a different manner. In the example program from Section 2, if we replace statement `x=t->n` with `x=x->n`, the algorithm will not verify the shape property. This is because the analysis does not know that `x=t` at that point, and cannot add `t->n` to the hit or miss set. To address this issue, the analysis needs expression equality information.

- *Worst-case exponential blowup*. Similar to most of the existing shape abstractions, the size of our abstraction is exponential in the worst case. In practice, we have seen cases where a blowup in the number of configuration occurs, but only because of the imprecision in the analysis, e.g., for traversals of doubly-linked lists with multiple pointers.

However, we believe that these are limitations of the current algorithm, but not of the framework with local reasoning. We anticipate these issues can be addressed by extending the abstraction and the algorithm, while still reasoning about one single location. For instance, the decision function could be improved to rule out spurious configurations; an expression equality analysis could be added as an underlying analysis in the vertical decomposition to address the robustness issue; and configurations could be extended with reachability information and structural invariants about the tracked location. We chose to keep the algorithm in this paper simpler and cleaner to emphasize the concept and to make it amenable to formal presentation. These issues will be the subject of future work.

8. 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 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. 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. The experiments were conducted on a 1.2GHz Athlon machine with 256MB of memory, running Linux RedHat 9.

8.1 Core List Manipulations

We have tested the analysis on the following programs that manipulate singly-linked lists:

- `insert` : inserts an element into a list;
- `delete` : deletes an element from a list;
- `splice` : list splicing program from Section 2;
- `reverse` : iterative list reversal program from [8];
- `quicksort` : recursive quicksort program from [5];
- `insertion_sort` : iterative insertion sort.

The analysis has successfully determined that all of these programs preserve acyclic list shape. It has also determined that none of the programs leak memory or access deallocated memory. The analysis took less than 1 second for these programs altogether.

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, 38 of which were actual memory leaks (a few warnings were for double frees and accesses to deallocated memory, all of which were false). Hence, more than one warning out of three was an actual error. Given that the tool uses sound analysis techniques, we view this as a low rate of false positives.

The error reporting is based on error traces. An error trace represents an execution trace through the program that leads to an error. The tool produces these traces by following the program execution backwards through the configuration graph, from the error point to the allocation site; we find the traces to be extremely helpful in identifying whether a warning is legitimate. Furthermore, the tool clusters error traces on a per-region basis: all of traces in one cluster refer to errors about locations in the same region. The tool then reports one warning per cluster. We find this clustering technique very useful in identifying distinct bugs, since all of the errors in one cluster essentially refer to the same programming error. For our programs, the majority of clusters have a single trace; a few clusters have less than 10 traces; and one cluster has more than 100 error traces.

As mentioned above, all of the detected errors were memory leaks. Most of them were caused as functions failed to clean up local resources on abnormal return paths. For instance, some functions in SSH do not reclaim memory when a connection fails. In a few cases, however, functions failed to release resources on all return paths.

The examination of false warnings indicated that they were due to several sources of imprecision. Many of the errors were due to lack of path 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. False warnings were also due to the context-insensitive treatment of global variables in pointer analysis. For instance, one of the programs stores references to several buffers (stdin, stout, and stderr) into global variables. Since these references are being passed as the same argument to a function, pointer analysis merges them together. This imprecision in the pointer analysis then impacts the precision of shape analysis. The above-mentioned cluster with more than 100 error traces was due to this kind of imprecision.

The tool was most affected when the analysis could not accurately identify shapes, as in the case of doubly-linked lists or for heap manipulations where it lacked variable equality information, as mentioned in Section 7. This lead to a blowup in the number of feasible configurations and the analysis failed to complete in such cases. However, we are able to isolate these situations using incremental analysis: we explored one allocation site at a time and introduced a cutoff for the amount of exploration to be performed from any given allocation site (by limiting the size of the worklist). Figure 16 indicates that the analysis did not complete for 3 allocation sites in SSL and 10 sites in Binutils due to this kind of imprecision. They represent about 10% of the allocation sites in these programs.

Program	OpenSSH	OpenSSL	BinUtils
Size (LOC)	18.6 K	25.6 K	24.4 K
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: Results for Larger Programs

Overall, we consider that these results are encouraging. They suggest that the local reasoning approach to shape analysis is both sufficiently lightweight to scale to larger programs, and sufficiently precise to yield a low rate of false warnings.

9. RELATED WORK

There has been significant work in the past decade in the area of shape analysis [34]. Early approaches to shape analysis have proposed the use of path matrices and other matrices that capture heap reachability information [18, 17, 11]. In particular, Ghiya and Hendren [11] 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 an abstract interpretation that uses shape graphs to model heap structures [29]. They introduce materialization and summarization as key techniques for the precise computation of shapes. Role analysis [21] uses shape graph abstractions to check program specifications for heap shapes and heap effects. Experiments have not been presented for these analyses.

In subsequent work, Sagiv et. al. have proposed a parametric shape analysis framework based on 3-valued logic [31]. 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 [23] and has been successful at verifying correctness and safety properties for complex heap manipulations [22, 8, 35]. Work on inter-procedural analysis in TVLA has explored modeling the program stack using 3-valued structures [28] and expressing the function input-output relations for individual heap cells [20]. These analyses seem expensive in practice; for instance, the analysis of a recursive procedure that deletes an element from a list takes more than 200 seconds [20].

Yahav and Ramalingam [36] have 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. In contrast, our approach is homogeneous, precisely abstracts the entire heap, but models each heap location separately. Their approach enabled TVLA to analyze programs of up to 1.3 KLOC.

Reynolds, O'Hearn, and others propose Separation logic [27, 26] and BI (the logic of Bunched Implications) [19] as extensions of Hoare logic that permit reasoning about mutable linked structures. They provide features (such as the separating conjunction and implication, or the frame rule) that make it easier to express correctness proofs for pointer-based programs. Compared to static

analyses, these logics provide techniques for *verifying* program correctness; analyses, on the other hand, provide techniques for *automatically inferring* points-to properties. A significant challenge for static analysis is the automatic synthesis of loop invariants. Generally speaking, inferring points-to properties is more difficult than verifying them. Therefore, although separation logic works well for correctness proofs with local reasoning, it is not clear whether an analysis can use local reasoning alone. In our system, we push global reasoning down to the pointer analysis component; and that enables us to use local reasoning at the shape analysis level. Furthermore, we use a finite shape abstraction that localizes the reasoning to one single heap location.

Demand-driven and incremental algorithm have been proposed in the area of pointer analysis. Heintze and Tardieu [16] describe a technique to answer aliasing queries by exploring the minimal set of points-to constraints that yields the desired answer. Vivien and Rinard [33] 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 Clouseau [15], a static leak detector tool for C and C++ programs. They use a notion of pointer ownership to describe the references responsible for freeing heap cells, and formulate the analysis as an ownership constraint system. Their approach is able to detect leaks and double frees, but cannot detect accesses through dangling pointers. Intuitively, that is because their system tracks owning pointers (and doesn't have information about targets of non-owning pointers), while ours tracks memory locations and their allocation state. The experiments indicate that our shape analysis approach yields more precise results than the pointer ownership approach. In particular, Clouseau reports a large number of warnings when the program places pointers in aggregate structures, or when it manipulates multi-level pointers.

Das et. al. present ESP [6], a path-sensitive tool for verifying state machine properties. ESP uses property simulation, a technique that captures path information but avoids the exponential cost of a full path-sensitive analysis. 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 temporal properties for non-recursive structures, not shapes in recursive heap structures. DeLine and Fahndrich propose Vault [7], an extension of C where programmers can specify resource management protocols that a compiler can enforce. The system can ensure, for instance, that the program doesn't leak resources. But it cannot precisely track unbounded numbers of resources such as those that arise in recursive structures.

The SLAM Project [2, 3] uses predicate abstraction refinement along with model checking techniques to check temporal safety properties of C programs. Necula et al. propose CCured [25], a an analysis and transformation system for C programs that uses type inference to identify type-safe pointers, and instrument the remaining pointers with run-time checks. However, CCured does not address the memory deallocation problem and uses a garbage collector instead.

Other existing error-detection tools, such as Metal [9] and Prefix[4], use unsound techniques to limit the number of false positives or to avoid fixed-point computations.

Finally, dynamic memory error detection tools such as Purify [13] or SWAT [14] instrument the program to detect errors at run-time. They test only one run of the program and may miss errors that are not exposed in that run; in particular, they may miss errors in rarely executed code fragments.

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 easier 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.

11. 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] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[3] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 2002.

[4] 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.

[5] S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *Proceedings of the 10th International Static Analysis Symposium*, San Diego, CA, June 2003.

[6] 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.

[7] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, UT, June 2001.

[8] 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.

[9] 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.

[10] M. Fahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[11] 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.

[12] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. Technical Report TR2004-1968, Cornell University, October 2004.

[13] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the 1992 Winter Usenix Conference*, January 1992.

[14] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004.

[15] 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.

[16] 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.

[17] 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.

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

[19] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th Annual ACM Symposium on the Principles of Programming Languages*, London, UK, January 2001.

[20] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proceedings of the 11th International Static Analysis Symposium*, Verona, Italy, August 2004.

[21] 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.

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

[23] 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.

[24] D. Liang and M.J. Harrold. Effi cient 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.

[25] G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofi tting of legacy code. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 2002.

[26] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *Proceedings of the 31th Annual ACM Symposium on the Principles of Programming Languages*, Venice, Italy, January 2004.

[27] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 2002.

[28] N. Rinetzky and M. Sagiv. Interprocedural shape analysis for recursive programs. In *Proceedings of the 2001 International Conference on Compiler Construction*, Genova, Italy, April 2001.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

APPENDIX

A. LANGUAGE SEMANTICS

The operational semantics of the simple language from Figure 7 uses the following semantic domains:

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

The following rules describe the evaluation of expressions. Relation $\langle e, \sigma\rangle \to_l l$ evaluates e in store σ to the location l of e; and relation $\langle e, \sigma\rangle \to_v v$ evaluates e in store σ to the value v of e. Given a concrete store $\sigma = (\sigma_v, \sigma_l, \sigma_f)$, we use the notations: $\mathit{dom}_v(\sigma) = \mathit{dom}(\sigma_v)$, $\mathit{dom}_l(\sigma) = \mathit{dom}(\sigma_l)$, and $\mathit{dom}_f(\sigma) = \mathit{dom}(\sigma_f)$. The evaluation rules are as follows.

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

$$\frac{\langle e, \sigma\rangle \to_l l \quad (l, f) \in \mathit{dom}_f(\sigma)}{\langle e.f, \sigma\rangle \to_l \sigma(l, f)} \qquad \frac{\langle e, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma)}{\langle e, \sigma\rangle \to_v \sigma(l)}$$

$$\frac{\langle e, \sigma\rangle \to_l l}{\langle \&e, \sigma\rangle \to_v l} \qquad \frac{}{\langle \mathbf{null}, \sigma\rangle \to_v \mathit{null}}$$

The evaluation relation for statements $\langle s, \sigma\rangle \to_s \sigma'$ indicates that the execution of statement s in input store σ produces the store σ'. The evaluation rules are as follows:

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

$$\frac{\begin{array}{c}\langle e, \sigma\rangle \to_v l \quad \forall f \in F : (l, f) \in \mathit{dom}_f(\sigma) \\ \sigma' = (\sigma - \{(l, f) \mapsto \_\}_{f\in F}) - \{\sigma(l, f) \mapsto \_\}_{f\in F}\end{array}}{\langle \mathbf{free}(e), \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e_0, \sigma\rangle \to_l l \in \mathit{dom}_l(\sigma) \quad \langle e_1, \sigma\rangle \to_v v}{\langle e_0 \leftarrow e_1, \sigma\rangle \to_s \sigma[l \mapsto u]}$$

$$\frac{\begin{array}{c}\mathit{prog}(f) = ((x_1, .., x_n), s) \quad \forall i = 1..n : \ \langle e_i, \sigma\rangle \to_v v_i \\ \langle s, \sigma[\sigma(x_i) \mapsto v_i]_{i=1..n}\rangle \to_s \sigma'\end{array}}{\langle \mathbf{call}\ p(e_1, .., e_n), \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle s_0, \sigma\rangle \to_s \sigma'' \quad \langle s_1, \sigma''\rangle \to_s \sigma'}{\langle s_0\ ;\ s_1, \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e, \sigma\rangle \to_v v = \mathit{null} \quad \langle s_0, \sigma\rangle \to_s \sigma'}{\langle \mathbf{if}\ (e)\ s_0\ \mathbf{else}\ s_1, \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e, \sigma\rangle \to_v v \neq \mathit{null} \quad \langle s_1, \sigma\rangle \to_s \sigma'}{\langle \mathbf{if}\ (e)\ s_0\ \mathbf{else}\ s_1, \sigma\rangle \to_s \sigma'}$$

$$\frac{\langle e, \sigma\rangle \to_v v = \mathit{null}}{\langle \mathbf{while}\ (e)\ s, \sigma\rangle \to_s \sigma} \qquad \frac{\begin{array}{c}\langle e, \sigma\rangle \to_v v \neq \mathit{null} \\ \langle s, \sigma\rangle \to_s \sigma'' \\ \langle \mathbf{while}\ (e)\ s, \sigma''\rangle \to_s \sigma'\end{array}}{\langle \mathbf{while}\ (e)\ s, \sigma\rangle \to_s \sigma'}$$