Static Analysis of Accessed Regions in Recursive Data Structures

Stephen Chong and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{schong,rugina}@cs.cornell.edu

Abstract. This paper presents an inter-procedural heap analysis that computes information about how programs access regions within recursive data structures, such as sublists within lists or subtrees within trees. The analysis is able to accurately compute heap access information for statements and procedures in recursive programs with destructive updates. We formulate our algorithm as a dataflow analysis which computes shape information and heap access information at each program point. We use an abstraction of the heap based on shape graphs, whose nodes represent heap regions and whose edges encode the reachability between these regions. Our analysis is able to summarize the effects of procedures: it computes the heap regions being accessed by each procedure in terms of the heap abstraction at the entry of the procedure. We use node labels to express the regions at program points inside procedures in terms of the regions at procedure entry points. The inter-procedural analysis uses a fixpoint algorithm to compute the heap regions accessed by the whole execution of each procedure, including all of the procedures it invokes. We demonstrate how our analysis computes precise heap region access information for a recursive quicksort program that sorts a list in-place, using destructive updates.

1 Introduction

Programs often build and manipulate dynamic structures such as trees or lists. To check or enforce the correct construction and manipulation of dynamic heap structures, the compiler must automatically discover invariants that characterize the structure (or shape) of the heap in all of the possible executions of the program; such algorithms are usually referred to as shape analysis algorithms. In the past decades, researchers have developed a number of shape analysis algorithms that identify various properties of the heap, including aliasing, sharing, cyclicity, or reachability [23]. Such properties allow the compiler to distinguish between heap structures such as trees and cyclic graphs. However, none of the existing shape analysis algorithms aim at characterizing the heap regions being accessed (i.e., read or written) by statements and procedures in the program.

There has been recent research to provide language support for dynamic allocation of heap regions [5, 6, 9], or to provide algorithms for region inference [22, 10, 16]. However, these approaches typically allocate all of the elements of a recursive data structure, such as a list or a tree, in a single heap region. They are not able to model regions *within* recursive data structures, such as sublists within lists or subtrees within trees.

This paper presents a context-sensitive inter-procedural analysis that extracts information about how the program accesses (i.e., reads or writes) heap regions within recursive data structures, for languages with destructive updates. To accurately compute how programs access the heap, our analysis computes: 1) *precise shape information*, using an abstraction which models heap regions and characterizes cyclicity and reachability properties for these regions; and 2) *region access information*, which describes which regions are being read or written by statements and procedures in the program. For each procedure, the analysis summarizes the regions being accessed by the whole execution of that procedure, including all of the procedures it invokes.

The shape analysis algorithm uses a finite abstraction of the heap based on shape graphs. The nodes in a shape graph are summary nodes, representing connected heap regions, and the edges describe the reachability between regions. To compute accurate shape information, the analysis keeps track of potential cycles in each region. The analysis also uses the materialization and summarization techniques proposed in [20]; and it identifies heap regions based on reachability from stack variables, similarly to the shape analyses presented in [23, 3]. But unlike the heap abstractions in these analyses, our abstraction exclusively uses summary nodes. This leads to a smaller shape abstraction for certain programs, such as list traversals using multiple traversing pointers.

The difficulty of characterizing accessed heap regions strongly depends on the abstraction that the analysis uses. It is straightforward to determine the accessed regions when the abstraction uses a *global* partitioning of the heap, with one summary node per allocation site. Such abstractions have been used in some shape analyses [1] and pointer analyses [24, 19]. The fact that makes it easy to characterize heap accesses in this case is that each summary node represents the same set of concrete heap locations throughout the program. However, heap abstractions based on allocation sites are not accurate in the presence of destructive updates. More precise heap abstractions use a *per program point* partitioning of the heap, according to the referencing relationships or reachability from stack pointers at each point in the program [20, 23, 3]; our algorithm uses such an abstraction. In this case, a summary node may model different sets of locations at different program points, because the referencing and reachability relationships of heap locations with resprect to stack pointers may change. This makes it difficult to summarize heap accesses at different program points; in particular, it makes it difficult to summarize the heap accesses for the whole execution of each procedure.

We solve this problem using *node labels* in the shape abstraction. The analysis assigns a fresh new label to each region at the beginning of a procedure. Then, during the analysis of the procedure, it computes a set of labels for each summary node. This set describes the origins of the node's locations with respect to the regions at the procedure entry. Using the computed labels, the analysis can summarize all of the heap accesses in each procedure in terms of the regions at the beginning of the procedure.

The analysis uses a context-sensitive interprocedural algorithm to analyze function calls. At each call site, the analysis maps the shape and access region information before the call into the analysis domain of the callee, analyzes the invoked procedure, and then unmaps the result back into the caller's domain to determine the information right after the call. The analysis handles recursive procedures using a standard iterative fixpoint algorithm.

This paper makes the following contributions:

- **Analysis Problem**. It identifies a new analysis problem, that of computing the heap regions that statements and procedures in the program may access, in the presence of destructive updates;
- **Heap Abstraction**. It presents a new heap abstraction, where summary nodes represent connected heap regions and edges model reachability between regions. The abstraction further uses node labels to express the concrete locations of summary nodes in terms of the regions at the beginning of the procedure;
- **Shape Analysis**. It presents a dataflow analysis which computes shape information using this abstraction. It gives a precise, formal definition of the algorithm;
- **Theoretical Properties**. It gives formal results showing that the analysis algorithm is sound and is guaranteed to terminate.

The remainder of the paper is organized as follows. Section 2 presents an example. Section 3 presents the shape analysis algorithm and the computation of accessed regions. Finally, Section 5 discusses related work.

2 Example

Figure 1 presents an example of a program that our analysis is designed to handle. This is a quicksort program which recursively sorts a list region (i.e. a sublist) in-place, using destructive updates. For this program we want to automatically check that the two recursive calls access disjoint sublists within the same list. We first describe the execution of this program and then we discuss the analysis information required to check this property.

2.1 Program Execution

At each invocation, the function `quicksort` sorts a sublist, consisting of all the list elements between `first` and `last`, exclusively (i.e., not including `first` and `last`). At the end of the invocation, `first->next` will point to the first element in the sorted sublist, and the `next` field of the last element in the sorted sublist will point to `last`. When the function is first invoked, `first` must be not `NULL`, different than `last`, and the element pointed to by `last` must be reachable from `first`.

The function first looks for the base cases of the computation, where the sublist has at most one element, in which case it immediately returns (lines 11 and 14). If the argument sublist has at least two elements, the function sorts the sublist according to the standard quicksort algorithm. The variable `mid` represents the pivot and is initialized (line 10) to point to the next element after `first`. To partition the sublist with respect to the pivot's value, the program traverses the list with two pointers, `prev` and `crt`, using the loop between lines 16 and 27. During the partitioning, the list region between `first` and `mid` contains the elements less than or equal to the pivot, and the list region between `mid` and `crt` contains the elements greater than the pivot. Note that the program uses destructive updates to move elements into the "less than" partition (lines 22-24).

Finally, the program recursively sorts the two partitions (lines 29, 30); each of these calls sorts its partition in-place.

```
1  typedef struct cell {
2    int val;
3    struct cell *next;
4  } list;
5
6
7  void quicksort(list *first, list *last) {
8    list *mid, *crt, *prev;
9
10   mid = prev = first->next;
11   if (mid == last)  return;
12
13   crt = prev->next;
14   if (crt == last) return;
15
16   while(crt != last) {
17    if (crt->val > mid->val) {
18       /* append to "greater than" part */
19       prev = crt;
20     } else {
21       /* prepend to "less than" part */
22       prev->next = crt->next;
23       crt->next = first->next;
24       first->next = crt;
25     }
26     crt = prev->next;
27   }
28
29   quicksort(first, mid);
30   quicksort(mid, last);
31 }
```

Fig. 1. Quicksort on lists

2.2 Required Analysis Information

To determine that the recursive calls to `quicksort` access disjoint regions of the list, the analysis must extract the following key pieces of information:

1. *Regions*. The analysis must use an abstraction that distinguishes between different list regions. For instance, the analysis must distinguish between the sublist from `first` to `mid`, from `mid` to `prev`, and from `crt` to `last`. The abstraction must thus use different summary nodes for each of these regions.
2. *Cyclicity Information*. The analysis must accurately determine that the program preserves listness. That is, it must establish that the sorting function produces an acyclic list whenever it starts with an acyclic list. In particular, it must determine that the destructive updates that move elements into the “less than” partition (lines 22-24) preserve the listness property.

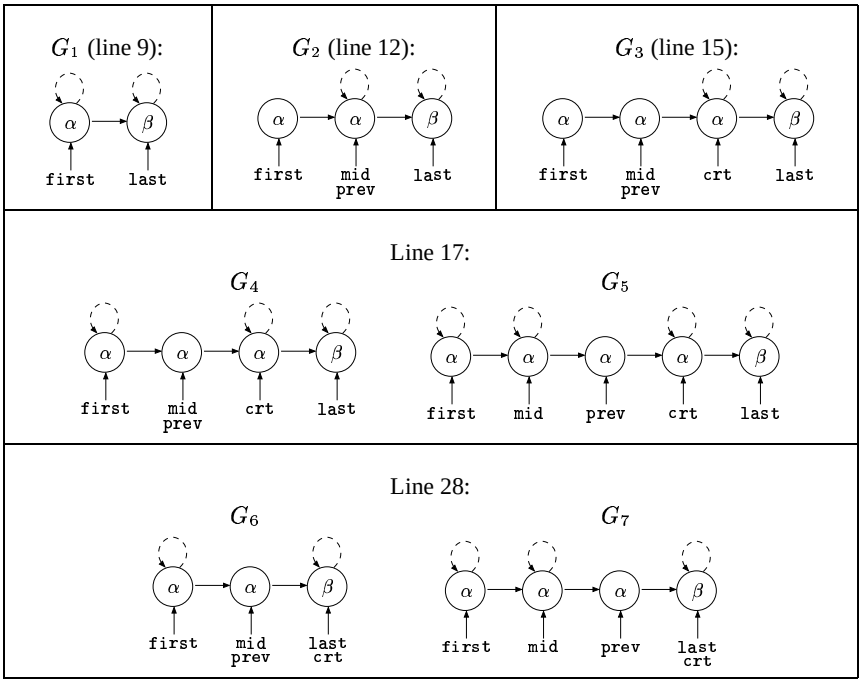

Fig. 2. Abstract shape graphs at selected program points for example

3. *Reachability Information*. The analysis must precisely determine which sublists are reachable from which other sublists.
4. *Access Information*. The analysis must determine what sublists each invocation of `quicksort` accesses. It must give a precise characterization of the accessed heap locations, with respect to the regions at the beginning of the function.

Figure 2 shows the abstraction that our analysis computes at several key points in the program. This abstraction precisely captures the above pieces of information. None of the nodes in the graphs are cyclic. The solid edges are must edges and the dashed edges are may edges.

The graph G_1 shows the abstract shape graph at the beginning of the function, which represents the portion of the heap reachable from the function arguments, `first` and `last`. The abstraction consists of two summary nodes, denoting two list regions. The summary node pointed to by `first` represents the sublist between `first`, inclusively, and `last`, exclusively. The summary node pointed to by `last` represents the tail of the list. Each of these sublists is a connected, single-entry heap region, whose entry point is directly accessible from the stack. These summary nodes represent disjoint regions within the same list structure.

The goal of the analysis is to compute the regions that `quicksort` accesses, with respect to the heap state at the beginning of the function (i.e. G_1). Therefore, the analysis

assigns different fresh new labels, α and β, to the summary nodes in G_1. These labels model the regions that these nodes represent *at the beginning of the function*. Then, at each point in the function body, the analysis expresses the concrete locations of each summary node in terms of the regions α and β.

The graph G_2 shows the abstract information at line 12 , after the assignments at line 10 and the test at line 11. The assignments traverse the `next` selector of the element pointed to by `first` and assign the resulting value to `mid` and `prev`. Since the summary node pointed to by `first` has two outgoing edges, the analysis must consider traversing each of them. In the first case, the analysis traverses the edge between the two nodes, which makes `mid`, `prev` and `last` be aliased. The analysis then eliminates this possibility because of the test condition at line 11, which requires that `last` and `prev` must be unaliased. In the second case, the analysis traverses the self edge of `first`, performing materialization and creating a new node. As part of the materialization process, the analysis assigns label α to the newly created node, because it models a subset of the region that the materialized node models. Note that during materialization, the analysis transforms the may self edge of `first` into a must edge between `first` and {`mid`,`prev`}. The algorithm similarly analyzes the statements at lines 13 and 14, to produce the graph G_3 at line 15.

Figure 2 also shows the fixed-point information that the analysis computes at the beginning of the loop, at line 17. This fixed-point solution consists of two shape graphs, G_4 and G_5, which represent the two possible heap configurations at this point: G_4 shows the case where `mid` and `prev` are aliased, that is, when the "greater than" partition is empty; and G_5 describes the case where this partition contains at least one element, and thus `mid` and `prev` point to different sublists. Similarly, the analysis computes two possible shape graphs, G_6 and G_7, right after the loop and before the first recursive call. In all of these graphs, the summary nodes pointed to by `first`, `mid`, and `prev` all have the label α. This indicates that each of these summary nodes models a subset of the locations that the region α represents in G_1.

Note that the previous shape analyses [3, 23] would produce a larger number of possible configurations for this program, because they distinguish between locations pointed to by stack pointers and locations pointed to only from the heap. The number of possible shape graphs that those analyses would produce is exponential in the number of traversing pointers with self edges: 8 graphs instead of G_4 and 16 graphs instead of G_5. Those shape graphs would also have a larger numbers of nodes, up to 9 nodes each.

2.3 Computing Heap Access Regions

The analysis can now use the computed shape and label information to determine that the two recursive calls access disjoint regions of the list. First, the analysis inspects every statement which accesses the heap using the pointers `first`, `mid`, `prev`, and `crt`. For each of these heap accesses, it examines the shape graphs at the corresponding access points and looks up the nodes where these pointers are pointing to. Since all these nodes are labeled with α, the analysis establishes that all these statements access locations from region α. The interprocedural analysis of the two recursive calls further determines that the whole execution of `quicksort`, including the recursive invocations, only accesses locations in α.

The analysis can then use the computed access region for the whole execution of `quicksort` to determine that the two recursive calls access disjoint sublists. A compiler could use this information to automatically parallelize this program and execute the recursive calls concurrently.

2.4 Detecting Potential Bugs

The analysis is able to help in the detection of potential bugs. Suppose the second recursive call (line 30) is incorrectly invoked with `quicksort(mid, last->next)`. In that case, the input shape graphs at each program point would be unchanged. However, the analysis would detect that the access region for the whole execution of `quicksort` is $\{\alpha, \beta\}$. Using this information at the recursive calls, the analysis will determine that both calls may access the tail of the list. Thus, it can no longer conclude that the two recursive calls access disjoint list regions.

Alternatively, consider if the programmer incorrectly writes the loop test condition at line 16 as `prev` $!=$ `last`. As a result, the analysis will compute two more possible shape graphs at the beginning of loop body; in each of these, `crt` points to the summary node labelled with β. Therefore, when the analysis inspects the assignment `crt->next` $=$ `first->next`, it will determine that this statement writes a heap location represented by β. As a result, the analysis will compute an access region $\{\alpha, \beta\}$ for quicksort and will report that the two recursive calls may access the same list elements.

Although both of the above situations are bugs, in general such messages may be false positives, because the analysis is conservative. Nonetheless, when the analysis computes access region information different than a given specification, the programmer can treat it as an indication of a potential bug in the program.

3 Algorithm

This section presents the analysis algorithm in detail. Although the algorithm computes shape graphs, label information, and accessed regions at the same time, we discuss them separately for clarity in presentation. We first show the basic intra-procedural algorithm which computes shape graphs. We then present how the algorithm keeps track of labels and how it computes the accessed heap regions. Finally, we describe the inter-procedural analysis algorithm.

3.1 Shape Analysis.

We formulate our shape analysis algorithm as a dataflow analysis which computes a shape graph at each program point. This section defines the dataflow information, the meet operator, and the transfer functions for statements.

We assume that that the program is preprocessed to a form where each statement is of one of the following six forms: `x` $=$ `NULL`, `x` $=$ `y`, `x` $=$ `malloc()`, `x->n` $=$ `NULL`, `x->n` $=$ `y`, or `x` $=$ `y->n`. Without loss of generality, we assume that each variable update `x` $=$ ... is preceded by a statement `x` $=$ `NULL` and that each selector update

`x->n = y` is preceded by `x->n = NULL`. We assume that `n` is the unique selector name in the program.

Our algorithm expresses the shape information using predicates in three-valued logic [21], which consists of truth values 0 (false), 1 (true), and 1/2 (unknown). We use the standard three-valued order and operators: the conjunction $t \wedge_3 t' = min(t, t')$; the disjunction $t \vee_3 t' = max(t, t')$; the partial order $t \sqsubseteq_3 t'$ for $t = t'$ or $t' = 1/2$; and the meet $t \sqcup_3 t'$ equal to t if $t = t'$ and 1/2 otherwise. We explicitly use subscript 3 when we refer three-valued operators, and we don't use any subscripts for the standard, two-valued operators.

Concrete Heaps. Concrete heaps represent the statically unbounded set of concrete heap locations and the points-to relations between them. We formalize concrete heaps as follows. Let V_s be the set of stack variables in the program. A concrete heap is a triple (L_h, E_s, E_h) consisting of:

- A set L_h of heap locations;
- A set $E_s \in V_s \times L_h \to \{0, 1\}$ of edges from stack variables to heap locations;
- A set $E_h \in L_h \times L_h \to \{0, 1\}$ of edges from heap locations to heap locations;

For edges, a value 1 indicates the presence of the edge in the concrete heap; a value 0 indicates its absence. We then consider two predicates which characterize the paths in this graph: $P(v, l)$ indicates if there is a path in the concrete heap between the variable $v \in V_s$ and the concrete location $l \in L_h$; and $Q(v, l', l)$ indicates that there is a path in the concrete heap between variable $v \in V_s$ and location $l \in L_h$ which doesn't contain the location $l' \in L_h$.

For each set of stack variables $X \subseteq V_s$, $X \neq \emptyset$, we define the *heap root* l^c_X as the heap location pointed to exactly by the variables in X: $E_s(x, l^c_X) \Leftrightarrow x \in X$. We also define the set n^c_X of *nodes owned by* X as the concrete locations that can only be reached from l^c_X, and only on paths that don't traverse other roots:

$$n^c_X = \{\, l \mid \forall x \in X \,.\, P(x, l) \;\wedge\; \forall y \notin X \,.\, \neg Q(y, l^c_X, l) \,\}$$

Finally, for $X = \emptyset$ we define the set n^c_ϕ of all heap locations reachable from the stack, but not owned by any set X of variables. These are heap locations reachable from the stack through different roots. We emphasize that all of the sets n^c_X represent disjoint sets of concrete heap locations. Also, each of the sets n^c_X, $X \neq \emptyset$ are connected, single-entry subgraphs of the concrete heap, with entry node l^c_X; however, n^c_ϕ is not a single-entry subgraph, and is not connected either.

Dataflow Information. Let $L_a = \{\, n_X \mid X \subseteq V_s \,\}$ be a set of abstract heap nodes, where each node n_X models the concrete heap locations in n^c_X. The dataflow information of our analysis is an *abstract shape graph* $G = (N, E, C)$, where:

- $N \subseteq L_a$ is the set of nodes;
- $E : N \times N \to \{0, 1/2, 1\}$ is the set of edges with reachability information.
- $C : N \to \{0, 1/2, 1\}$, the cyclicity information on nodes.

The above predicates have the following meaning. If $E(n,m) = 1$, the edge between n and m is definitely reachable (i.e., the locations in m are definitely reachable from the locations in n); if $E(n,m) = 1/2$, the edge is possibly reachable; otherwise, it doesn't exist. The cyclic predicate $C(n)$ is 1 if there definitely is a cycle in the concrete heap structure modeled by n; it is 1/2 if there may be a cycle; and it is 0 if there definitely isn't any cycle. The predicate C refers to internal cycles consisting only of locations modeled by n. We denote by A the set of all abstract shape graphs.

We graphically represent nodes n with $C(n) \neq 0$ with double circles; definitely reachable edges with solid lines; and possibly reachable edges with dashed lines. Finally, we omit n_ϕ if it has no incoming or outgoing edges to other abstract nodes. The shape graphs from Figure 2 use this graphical representation.

The *abstraction function* δ characterizes the relation between concrete heaps and abstract shape graphs. Given a concrete heap (L_h, E_s, E_h), we define its abstraction $(N, E, C) = \delta(L_h, E_s, E_h)$ as follows:

$$N = \{\, n_X \mid n_X^c \neq \emptyset \,\} \cup \{\, n_\phi \,\}$$

$$E(n_X, n_Y) = \bigvee \{\, E_h(l_x, l_y) \mid l_x \in n_X^c, l_y \in n_Y^c \,\}$$

$$C(n_X) = \begin{cases} 1 \text{ if } \exists l_1, \ldots, l_k \in n_X^c \,.\, E_h(l_k, l_1) \wedge \bigwedge\{\, E_h(l_i, l_{i+1}) \mid 1 \leq i < k \,\} \\ 0 \text{ otherwise} \end{cases}$$

The node n_ϕ models a subgraph of the heap which is not connected and may have multiple entries. Intuitively, n_ϕ contains imprecise shape information and the analysis uses it as a fallback mechanism to conservatively handle the case of shared structures.

Merge Operation. The dataflow analysis merges shape graphs at each control-flow point in the program. The merge operation is non-trivial in our abstraction, because it must account for the following scenario. Consider two abstract graphs G_1 and G_2 that we want to merge. Also consider that one of these graphs contains a definitely reachable edge (n,m), but the other graphs contains only the target node m. In this case, m is not reachable from n in the second graph, so the analysis must conclude that (n,m) is possibly reachable in the merged graph. The following merge operation precisely captures this behavior. If $G_1 = (N_1, E_1, C_1)$ and $G_2 = (N_2, E_2, C_2)$, the merged graph is $G_1 \sqcup G_2 = (N, E, C)$, where:

$$N = N_1 \cup N_2$$

$$C(n) = \begin{cases} C_1(n) \sqcup_3 C_2(n) & \text{if } n \in N_1 \cap N_2 \\ C_1(n) & \text{if } n \notin N_2 \\ C_2(n) & \text{if } n \notin N_1 \end{cases}$$

$$E(n,m) = \begin{cases} E_1(n,m) \sqcup_3 E_2(n,m) & \text{if } \{n,m\} \subseteq N_1 \cap N_2 \\ 1 & \text{if } (\, E_1(n,m) = 1 \wedge m \notin N_2 \,) \vee (\, E_2(n,m) = 1 \wedge m \notin N_1 \,) \\ 0 & \text{if } (\, n \notin N_1 \vee m \notin N_1 \,) \wedge (\, n \notin N_2 \vee m \notin N_2 \,) \\ 1/2 & \text{otherwise} \end{cases}$$

In general we may have abstract shape graphs containing nodes n_X and n_Y such that X and Y each include a stack variable $\mathtt{x} \in V_s$. Clearly, in no execution of the

Case: $\texttt{x} = \texttt{malloc}()$ $N' = N \cup \{n_{\{\texttt{x}\}}\}$ $C'(n) = \begin{cases} 0 & \text{if } n = n_{\{\texttt{x}\}} \\ C(n) & \text{otherwise} \end{cases}$ $E'(n,m) = \begin{cases} 0 & \text{if } n_{\{\texttt{x}\}} \in \{n, m\} \\ E(n,m) & \text{otherwise} \end{cases}$	*Case:* $\texttt{x} = \texttt{y}$ $N' = \{ f(n) \mid n \in N \}$ $C'(n) = C(f^{-1}(n))$ $E'(n,m) = E(f^{-1}(n), f^{-1}(m))$ $f : N \to N'$ $f(n_Y) = \begin{cases} n_{Y \cup \{\texttt{x}\}} & \text{if } \texttt{y} \in Y \\ n_Y & \text{otherwise} \end{cases}$
Case: $\texttt{x->n} = \texttt{NULL}$ $N' = N$ $C'(n_X) = \begin{cases} 0 & \text{if } \texttt{x} \in X \\ C(n_X) & \text{otherwise} \end{cases}$ $E'(n_X, n_Y) = \begin{cases} 0 & \text{if } \texttt{x} \in X \\ E(n_X, n_Y) & \text{otherwise} \end{cases}$	*Case:* $\texttt{x->n} = \texttt{y}$ $N' = N$ $C'(n_Z) = \begin{cases} 1 & \text{if } \{\texttt{x}, \texttt{y}\} \subseteq Z \\ C(n_Z) & \text{otherwise} \end{cases}$ $E'(n_X, n_Y) = \begin{cases} 1 & \text{if } \texttt{x} \in X \wedge \texttt{y} \in Y \\ E(n_X, n_Y) & \text{otherwise} \end{cases}$
Case: $\texttt{x} = \texttt{null}$, if $n_{\{\texttt{x}\}} \notin N$ $N' = \{ g(n) \mid n \in N \}$ $C'(n) = C(g^{-1}(n)), \forall n \in N$ $E'(n,m) = E(g^{-1}(n), g^{-1}(m))$ $g : N \to N'$ $g(n_Z) = n_{Z - \{\texttt{x}\}}$	

Fig. 3. Transfer function $[\![s]\!]_{\text{conf}}(N, E, C) = (N', E', C')$ for simple cases

program can there coexist a concrete heap location from n^c_X with one from n^c_Y. We say that a graph G is *incompatible* if it contains two nodes n_X and n_Y such that $X \neq Y \wedge X \cap Y \neq \emptyset$. Given a shape graph G, we say that G' is a *possible configuration* if it is a maximal compatible subgraph of G. We use the term "maximal subgraph" relative to the partial order relation induced by the merge operator defined above. Finally, we denote by *Confs* (G) the set of all possible configurations of a graph G.

Transfer Functions. For each statement s in the program, we define a transfer function $[\![s]\!] : A \to A$, which describes how the statement modifies the input abstract shape graph. The analysis first splits the input graph G into all of the possible configurations in *Confs* (G). It then analyzes each configuration separately. At the end, it merges the result for each configuration. Therefore, we express the transfer function as follows:

$$[\![s]\!](G) = \bigsqcup_{G' \in \mathit{Confs}\ (G)} [\![s]\!]_{\text{conf}}(G')$$

where $[\![s]\!]_{\text{conf}}$ is a transfer function defined to operate only on compatible graphs. Figures 3, 4, and 5 present the full definition of the transfer functions $[\![s]\!]_{\text{conf}}$ that the algorithm uses to analyze each configuration.

Figure 3 shows the transfer function for statements $\texttt{x} = \texttt{malloc}()$, $\texttt{x} = \texttt{y}$, $\texttt{x->n} = \texttt{NULL}$, $\texttt{x->n} = \texttt{y}$, and $\texttt{x} = \texttt{NULL}$ when $n_{\{\texttt{x}\}} \notin N$. These equations use two helper

$$
\begin{aligned}
N' &= N - \{n_{\{\mathtt{x}\}}\} \\
C'(n) &= C(n) \ \vee_3 \ (\ S(n, n_{\{\mathtt{x}\}}) \ \wedge_3 \ C(n_{\{\mathtt{x}\}})\ ) \vee_3 \\
&\quad (\ S(n, n_{\{\mathtt{x}\}}) \ \wedge_3 \ E(n, n_{\{\mathtt{x}\}}) \ \wedge_3 \ E(n_{\{\mathtt{x}\}}, n)\ ) \\
E'(n, m) &= E(n, m) \vee_3 \\
&\quad (\ S(n, n_{\{\mathtt{x}\}}) \ \wedge_3 \ E(n_{\{\mathtt{x}\}}, m)\ ) \vee_3 \\
&\quad (\ S(n, n_{\{\mathtt{x}\}}) \ \wedge_3 \ m = n\ ) \vee_3 \\
&\quad (\ S(n_\phi, n_{\{\mathtt{x}\}}) \ \wedge_3 \ m = n_\phi \ \wedge_3 \ E(n, n_{\{\mathtt{x}\}})\ ) \vee_3 \\
&\quad (\ S(n, n_\phi) \ \wedge_3 \ E(n_\phi, m)\ )
\end{aligned}
$$

$$
S(n, m) = \begin{cases}
1 \quad \text{if } E(n,m) = 1 \ \wedge \ (\ \forall n' \notin \{n, m\} \,.\, E(n', m) = 0\ ) \\
1/2 \text{ if } E(n,m) \in \{1, 1/2\} \ \wedge \\
\qquad (\ \forall n' \notin \{n, m\} \,.\, E(n', m) \in \{0, 1/2\}\ ) \ \wedge \\
\qquad (\ E(n,m) \neq 1 \ \vee \ \exists n' \neq m \,.\, E(n', m) = 1/2\ ) \\
0 \quad \text{otherwise}
\end{cases}
$$

$$
S(n_\phi, m) = \begin{cases}
1/2 \text{ if } \exists n, n' \,.\, n \neq n' \ \wedge \ E(n,m) \neq 0 \ \wedge \ E(n', m) \neq 0\ ) \\
0 \quad \text{otherwise}
\end{cases}
$$

$$
S(n, n_\phi) = \begin{cases}
1/2 \text{ if } E(n_{\{\mathtt{x}\}}, n_\phi) \neq 0 \ \wedge \ E(n, n_\phi) \neq 0 \ \wedge \ (\ \forall n' \neq n_{\{\mathtt{x}\}} \,.\, E(n', n_{\{\mathtt{x}\}}) \neq 1\ ) \\
0 \quad \text{otherwise}
\end{cases}
$$

Fig. 4. Transfer function for `x=NULL`, when $n_{\{\mathtt{x}\}} \in N$, which performs summarization. The predicate $S(n, m)$ evaluates to 1 in the case of precise summarization of m into n and 1/2 for imprecise summarization.

$$
[\![\mathtt{x} = \mathtt{y}\text{->}\mathtt{n}]\!]_{\text{conf}}(G) = \bigsqcup_{E(n_Y, p) \neq 0} G_p \quad (\text{where } \mathtt{y} \in Y)
$$

Case: $p = n_Y$ (**materialization on self edge**)

$$
\begin{aligned}
N_p &= N \cup \{n_{\{\mathtt{x}\}}\} \cup \{n_{Y \cup \{\mathtt{x}\}} \mid C(n_Y) \neq 0\} \\
C_p(n) &= \begin{cases} 0 & \text{if } n = N_Y \\ C(n_Y) & \text{if } n = n_{\{\mathtt{x}\}} \ \vee \ n = n_{Y \cup \{\mathtt{x}\}} \\ C(n) & \text{otherwise} \end{cases} \\
E_p(n, m) &= \\
&= \begin{cases}
E(n, m) & \text{if } n \notin \{n_Y, n_{\{\mathtt{x}\}}, n_{Y \cup \{\mathtt{x}\}}\} \\
1 & \text{if } n = n_Y \ \wedge \ m = n_{\{\mathtt{x}\}} \\
0 & \text{if } n = n_Y \ \wedge \ m \neq n_{\{\mathtt{x}\}} \\
E(n_Y, m) & \text{if } n = n_{\{\mathtt{x}\}} \ \wedge \ m \notin \{n_{\{\mathtt{x}\}}, n_Y\} \\
1/2 & \text{if } n = n_{\{\mathtt{x}\}} \ \wedge \ m = n_{\{\mathtt{x}\}} \\
1/2 \wedge_3 C(n_Y) & \text{if } n = n_{\{\mathtt{x}\}} \ \wedge \ m = n_Y \\
1 & \text{if } m = n = n_{Y \cup \{\mathtt{x}\}} \\
E(n_Y, m) & \text{if } m \neq n = n_{Y \cup \{\mathtt{x}\}}
\end{cases}
\end{aligned}
$$

Case: $p \neq N_Y$ *and* $p \neq n_\phi$

$$
\begin{aligned}
N_p &= \{\ h(n) \mid n \in N\ \} \\
C_p(n) &= \begin{cases} 0 & \text{if } n = N_Y \\ C(h(n)) & \text{otherwise} \end{cases} \\
E_p(n, m) &= \\
&\begin{cases}
E(h^{-1}(n), (h^{-1}(m)) & \text{if } n \neq n_Y \\
1 \quad \text{if } n = n_Y \ \wedge \ m = p \\
0 \quad \text{if } n = n_Y \ \wedge \ m \neq p
\end{cases} \\
&\text{where:} \\
h(n) &= \begin{cases} n_{Z \cup \{\mathtt{x}\}} & \text{if } p = n_Z \\ n & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5. Transfer function for `x = y->n`, which performs materialization. We obtain each graph $G_p = (N_p, E_p, C_p)$ by following one edge, from n_Y to p.

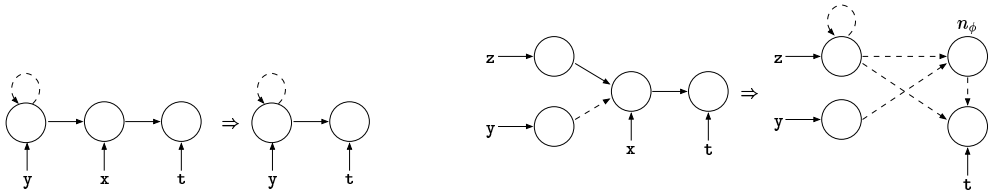

Fig. 6. Summarization for `x = NULL`: a) precise summarization, b) imprecise summarization.

functions f and g, which are invertible because the input graph G is compatible. Hence, f^{-1} and g^{-1} are well-defined.

The materialization and summarization of nodes is significantly more complex. Figure 4 presents the equations for the summarization of nodes, which takes place for a statement `x=NULL`, when $n_{\{\mathtt{x}\}} \in N$. Let n_Y be a node with an edge to $n_{\{\mathtt{x}\}}$. The analysis can precisely summarize the node $n_{\{\mathtt{x}\}}$ into n_Y only if there a single incoming edge to $n_{\{\mathtt{x}\}}$, and that edge is definitely reachable. Figure 6(a) shows this case. If there are multiple incoming edges to $n_{\{\mathtt{x}\}}$, and two or more are definitely reachable edges, then the analysis summarizes $n_{\{\mathtt{x}\}}$ into n_ϕ. Finally, if there are multiple incoming edges to $n_{\{\mathtt{x}\}}$, and at most one is a definitely reachable edges, then the analysis must summarize $n_{\{\mathtt{x}\}}$ both into n_ϕ and into the nodes that point to $n_{\{\mathtt{x}\}}$ using definitely reachable edges. Figure 6(b) shows a case of imprecise summarization.

Figure 5 presents the transfer functions for an assignment `x = y->n`. Consider that n_Y is a node such that `y` $\in Y$. If n_Y has multiple outgoing edge, the variable `x` may traverse any of them during the execution of this statement. The algorithm analyzes each of these situations separately and then merges the results together. When traversing a self edge of n_Y, the analysis performs a materialization: it "extracts" one location from n_Y and creates the abstract node $n_{\{\mathtt{x}\}}$ to model the remaining locations. If the node n_Y may have cycles, the analysis performs an imprecise materialization and adds back edges to n_Y. Otherwise, the analysis performs a precise materialization and doesn't create spurious back edges.

3.2 Formal Results

We summarizes the main theoretical results for our abstraction and and our analysis in the following theorems. We show that the meet operation is well-defined, the transfer functions are monotonic, and the analysis is sound. We use $[\![s]\!]_c$ to denote the concrete semantics of s. Because the analysis lattice has finite height, the monotonicity of the transfer functions implies that the analysis is guaranteed to terminate.

Theorem 1. *The merge operation $\sqcup$ is idempotent, commutative, and associative.*

Theorem 2. *For all statements s, the transfer function $[\![s]\!]$ is monotonic.*

Theorem 3 (Soundness). *If H_c is a concrete heap, then for all statements s we have $\delta([\![s]\!]_c(H_c)) \sqsubseteq [\![s]\!](\delta(H_c))$, where δ is the abstraction function defined in Section 3.1.*

3.3 Node Labels

This section presents the algorithm for computing accessed heap regions for statements and procedures in the program. The main difficulty when computing the accessed heap regions in our abstraction is that the same node may represent different concrete heap locations in shape graphs at different program points. This makes it difficult to summarize heap accesses at different program points. In particular, it makes it difficult to summarize the heap accesses for each procedure in the program.

Our algorithm overcomes this difficulty using *node labels* to record information about the concrete heap locations that each abstract node represents. At the beginning of each procedure, the analysis assigns fresh labels to each node in the shape graph. Each label represents the set of concrete locations that the corresponding node models in the initial shape graph. During the analysis of individual statements, the algorithm computes, for each node and at each program point, the set of labels that describe the concrete locations that the node currently models. Finally, for each statement that reads or writes a heap location, the algorithm records the labels of the node being accessed by the statement. The analysis can therefore summarize the heap locations accessed by the whole execution of each procedure as a set of labels.

For allocation statements, the analysis uses one label per allocation site to model all of the heap cells allocated at this site in the current execution of the enclosing procedure. For an allocation site s, we denote by α_s the label for this site. We denote by *Lab* the set of all labels in the analysis. This set consists of allocation site labels and fresh labels at the beginning of procedures. Within each procedure, different labels model disjoint sets of concrete heap locations.

Our analysis algorithm computes shape graphs, node labels, and access regions simultaneously. It uses an extended heap abstraction which incorporates information about labels and accessed regions. An extended shape graph is a tuple (N, E, C, L, R, W), where N, E, and C are the same as before, $L : N \rightarrow \mathcal{P}(\mathit{Lab})$ represents the label information for nodes, and $R, W \subseteq \mathit{Lab}$ characterize the heap locations that have been read and written from the beginning of the enclosing procedure. The merge operation is the pointwise union for labels and read and write sets. That is, if $G_1 = (N_1, E_1, C_1, L_1, R_1, W_1)$ and $G_2 = (N_2, E_2, C_2, L_2, R_2, W_2)$, then the merged graph is $G_1 \sqcup G_2 = (N, E, C, L, R, W)$, where N, E, and C are computed as before, and:

$$
\begin{aligned}
R &= R_1 \cup R_2 \\
W &= W_1 \cup W_2 \\
L(n) &= \begin{cases} L1(n) \cup L_2(n) & \text{if } n \in N_1 \cap N_2 \\ L1(n) & \text{if } n \in N_1 - N_2 \\ L2(n) & \text{if } n \in N_2 - N_1 \end{cases}
\end{aligned}
$$

Figure 7 shows how the analysis computes the labels for each statement. For an allocation statement s, the analysis assigns the label α_s of that allocation site to the newly created node. During summarization the analysis adds the labels of the summarized node to the set of labels of the node which gets summarized into. During materialization, the newly created node inherits the labels from the node on which the materialization has been performed.

Statement	New Labels	Statement	New Labels
$\texttt{x} = \texttt{malloc}()$	$L'(n) = \begin{cases} L(n) & \text{if } n \in N \\ \alpha_s & \text{if } n = n_{\{\texttt{x}\}} \end{cases}$	$\texttt{x} = \texttt{y}$	$L'(n) = L(f^{-1}(n))$
$\texttt{x} = \texttt{NULL}$ $(n_{\{\texttt{x}\}} \in N)$	$L'(n) = \begin{cases} L(n) \cup L(n_{\{\texttt{x}\}}) & \text{if } S(n, n_{\{\texttt{x}\}}) \neq 0 \\ L(n) & \text{if } S(n, n_{\{\texttt{x}\}}) = 0 \end{cases}$	$\texttt{x} = \texttt{NULL}$ $(n_{\{\texttt{x}\}} \notin N)$	$L'(n) = L(g^{-1}(n))$
$\texttt{x} = \texttt{y->n}$ $(p = p_Y)$	$L'(n) = \begin{cases} L(n) & \text{if } n \in N \\ L(n_Y) & \text{if } n = n_{\{\texttt{x}\}} \\ L(n_Y) & \text{if } n = n_{Y \cup \{\texttt{x}\}} \end{cases}$	$\texttt{x->n} = \texttt{NULL}$ $\texttt{x->n} = \texttt{y}$	$L' = L$
		$\texttt{x} = \texttt{y->n}$ $(p \neq p_Y, n_\phi)$	$L'(n) = L(h^{-1}(n))$

Fig. 7. Equation for computing label sets, using the summarizarization predicate S and the functions f, g, and h defined in Figures 3, 4, and 5.

Finally, the analysis computes the locations being read and written by each statement in a straightforward manner. For each heap update $\texttt{x->n} = \texttt{NULL}$ or $\texttt{x->n} = \texttt{y}$, the analysis augments the set of written locations with the labels of all the nodes $\texttt{x}$ points to: $W' = W \cup \bigcup_{\texttt{x} \in X} L(n_X)$. Similarly, for each assignment $\texttt{x} = \texttt{y->n}$, the analysis augments the set of read locations with the labels of all the nodes $\texttt{y}$ points to: $R' = R \cup \bigcup_{\texttt{y} \in Y} L(n_Y)$. In all of the other cases, the sets of locations being read or written remain unchanged.

3.4 Interprocedural Analysis

Our algorithm performs a context-sensitive inter-procedural analysis to accurately compute shape information for procedure calls. At each call site, the algorithm maps the current analysis information into the name space of the invoked procedure, analyzes the procedure, then unmaps the results back into the name space of the caller. This general mechanism is similar to existing inter-procedural pointer analyses [4, 24, 19]. However, our mapping and unmapping processes are different than in pointer analysis because they operate on a different abstraction. In particular, our analysis maps and unmaps shape graphs, node labels, and read and write sets. Like existing context-sensitive pointer analyses, our algorithm caches the analysis results every time it analyzes a procedure. At each call site, the analysis sets up the calling context and looks this context up in the cache to determine if an analysis result is available for this context. If so, it uses the results of the previous analysis of the procedure. Otherwise, it analyzes the caller in the new context.

Mapping and Unmapping. The mapping process sets up the calling context for the invoked procedure. Consider a call statement $\texttt{f}(\texttt{a}_1, \ldots, \texttt{a}_\texttt{n})$ which invokes procedure $\texttt{f}$. Without loss of generality, we assume that each of the actual arguments $\texttt{a}_1, \ldots, \texttt{a}_\texttt{n}$ are local variables in the caller's environment. Let $\texttt{p}_1, \ldots, \texttt{p}_\texttt{n}$ be the formal parameters of the invoked procedure $\texttt{f}$. If the analysis information at the call site is $G = (N, E, C, L, R, W)$, the mapping process builds the input context G_i for the invoked procedure $\texttt{f}$ as follows:

– It first partitions the nodes N of G into two sets: N_r, representing the nodes reachable from the actual parameters $\texttt{a}_1, \ldots, \texttt{a}_\texttt{n}$ at the call site, and $N_u = N - N_r$,

representing the nodes unreachable from the actual parameters. Let G_r be the subgraph of G restricted to the nodes in N_r, and G_u be the unreachable subgraph of G, restricted to the nodes in N_u. The analysis proceeds with G_r to set up the calling context; it recovers the unreachable subgraph G_u later, during the unmapping.

- It then removes all of the local variables, except for the actual arguments, from the shape graph G_r and produces a new graph G'_r. For this, the analysis performs assignments $\mathtt{x} = \mathtt{NULL}$ for each variable $\mathtt{x}$ which is not an actual parameter. During the removal of local variables, the analysis constructs a node map $m : N'_r \rightarrow \mathcal{P}(N_r)$ that records, for each node in G'_r, which nodes of G_r it represents.
- The analysis re-labels each node in G'_r with a fresh label and produces a new graph G''_r. It records the re-labeling information using a function $l : Lab \rightarrow N'_r$ such that $l(\alpha')$ represents the node of N'_r which has been re-labeled with α'.
- Finally, the analysis replaces each actual parameter $\mathtt{a_i}$ in G''_r with the corresponding formal parameter p_i, and produces the graph $G_i = (N_i, E_i, C_i, L_i, \emptyset, \emptyset)$, which is the calling context for the invoked procedure $\mathtt{f}$.

Next, the analysis uses the constructed calling context G_i to determine the output graph $G_o = (N_o, E_o, C_o, L_o, R_o, W_o)$. The analysis computes G_o either by re-using a previous analysis for this context from the cache, or by analyzing the invoked function $\mathtt{f}$. Further, the analysis unmaps the output graph G_o and computes the graph G' in the caller's analysis domain, at the program point right after the call. The unmapping process consists of the following steps:

- The analysis replaces each formal parameter $\mathtt{p_i}$ with the corresponding actual parameter $\mathtt{a_i}$ at the call site, and produces a graph $G'_o = (N'_o, E'_o, C'_o, L'_o, R_o, W_o)$. Here we assume that the formal parameters are never modified in the procedure body. Hence, they point to the same location throughout the procedure. One can relax this condition using temporary variables and assignments which copy the initial values of the formal parameters into these temporary variables.
- Next, the analysis replace nodes in G'_o with nodes from the reachable subgraph G_r before the call; it produces a new graph $G''_o = (N''_o, E''_o, C''_o, L''_o, R''_o, W''_o)$. Intuitively, the analysis recovers the caller's local information, which has been removed during the mapping process.

  The algorithm computes the shape information N''_o, E''_o, C''_o, and L''_o as follows. For each node $n' \in N'_o$, it examines the following possibilities:
  - if $L'_o(n') = \{\alpha'\}$ and $\alpha' \notin R_o \cup W_o$: the region α' was neither read nor written by the execution of $\mathtt{f}$. It means that the internal structure of this region has not been modified by the call, and no pointers into the middle of this structure have been created. It is therefore safe to replace the node labeled with α' in G'_o with its corresponding subgraph from G_r, consisting of all the nodes in $m(l(\alpha'))$.
  - if $L'_o(n') = \{\alpha'\}$, $\alpha' \in R_o \cup W_o$, $\alpha' \notin L'_o(n''), \forall n'' \neq n', |m(l(\alpha'))\}| = 1$: the heap structure represented by the region α' may have been read or written, but it represents exactly one node in both G_r and G'_o. The analysis can safely replace the node n' with the unique node n of $m(l(\alpha'))$.
  - Otherwise, the heap structure represented by region node n' may have been modified or represents multiple nodes in G_r. The analysis conservatively re-

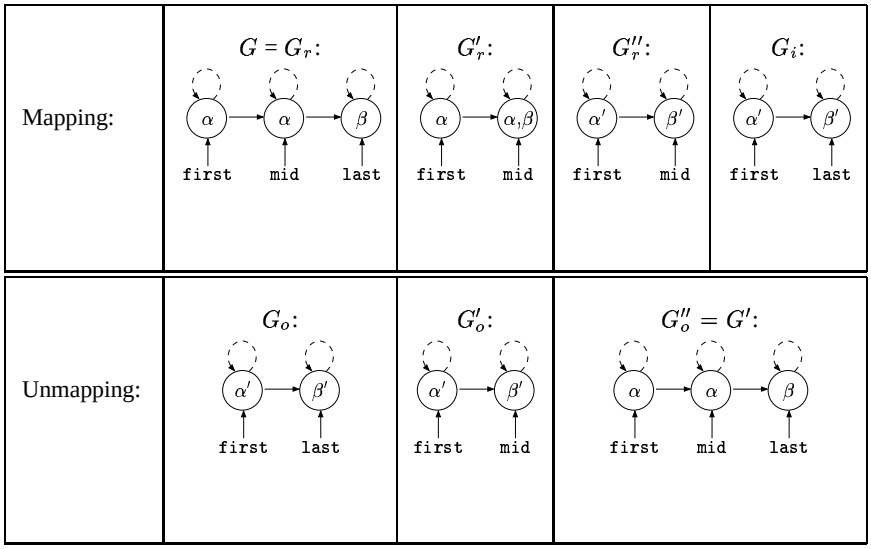

Fig. 8. Computed shape graphs during mapping and unmapping for quicksort example

places n' with all of the nodes in $\bigcup \{m(l(\alpha')) \mid \alpha' \in L'_o(n')\}$, adds edges between any two of these nodes, and makes all of the nodes cyclic.

The analysis computes the access region information R''_o and W''_o as follows:

$$R''_o = R \cup \bigcup \{L_r(n) \mid n \in m(l(\alpha')) \wedge \alpha' \in R_o\}$$
$$W''_o = W \cup \bigcup \{L_r(n) \mid n \in m(l(\alpha')) \wedge \alpha' \in W_o\}$$

where L_r is the label map of subgraph G_r.

– Finally, the analysis adds back the unreachable subgraph G_u into G''_o. The resulting graph G' represents the shape graph at the program point after the call.

Although the mapping and unmapping process is conservative, it may be imprecise for regions modified by the callee. In particular, the unmapping process imprecisely restores the local variables of the caller when these variables belong to regions modified by the invoked procedure. To reduce this kind of imprecision, our analysis performs *early nullification*: it runs a dead variable analysis, identifies the earliest points where local variables are guaranteed to be dead, and inserts nullifying statements `x = NULL` for such variables at these program points. This technique reduces the amount of local information and improves the efficiency and the precision of our algorithm.

Figure 8 shows the shape graphs that the analysis constructs during maping and unmapping for the first recursive call site in the quicksort example from Section 2. Using early nullification, the analysis inserts the statements `prev = NULL` and `crt = NULL` before the first call. These statements yield the graph G in Figure 8 at the call site. During the mapping process, the analysis derives the label map $l : \{\alpha' \mapsto n_{\{\texttt{first}\}}, \beta' \mapsto$

$n_{\{\texttt{mid}\}}\}$ and the node map $m : \{n_{\{\texttt{first}\}} \mapsto \{n_{\{\texttt{first}\}}\}, n_{\{\texttt{mid}\}} \mapsto \{n_{\{\texttt{mid}\}}, n_{\{\texttt{last}\}}\}\}$. During the unmapping process, the analysis starts with the output G_o, whose heap access information indicates that only the first node has been read and written: $R_o = \{\alpha'\}, W_o = \{\alpha'\}$. The analysis can accurately replace the node $n_{\{\texttt{mid}\}}$ of G_o with the subgraph of G_r consisting of nodes $n_{\{\texttt{mid}\}}$ and $n_{\{\texttt{last}\}}$, because the label β' in G'_o denotes a region that hasn't been read or written by the invoked procedure. The resulting graph $G''_o = G'$ represents the information after the call.

Recursive Procedures. For the analysis of recursive procedures, our algorithm uses a standard fixed-point approach [19]. For each calling context G_i, the analysis maintains a best analysis result G_o for that context. The analysis initializes the best result G_o to the bottom element in the analysis domain, which is a graph with an empty set of nodes and empty sets of read and written locations. During the analysis of the program, the algorithm creates a stack of analysis contexts that models the call stack in the execution of the program. Whenever the analysis encounters an invoked procedure already on the stack, with the same calling context, it uses its current best analysis result for that context. When the algorithm finishes the analysis of a procedure and computes a new analysis result, it merges this result with the current best analysis result for that calling context. If the current best result changes, the algorithm re-analyzes all of the dependent analyses. The process continues until it reaches a fixed point.

4 Extensions

In this section we discuss three extensions to the algorithm presented so far. These extensions improve the precision, the efficiency, and the functionality of our algorithm.

4.1 Configurations Versus Merged Graphs

During the analysis of each individual statement, our algorithm splits the current shape graph into possible configurations, analyzes each configuration separately, then merges the results together, as presented in Section 3.1. However, the merge operation may lose precision because the analysis may not be able to accurately recover the component configurations from the merged graph.

We can avoid this kind of imprecision using a more precise dataflow information consisting of sets of compatible graphs. In this case, the algorithm computes a set of shape graphs at each program point. The drawback of this approach is that the dataflow information may consists of a large number of graphs at each program point, making the analysis more expensive. With this extension, the algorithm trades efficiency for precision.

A possible trade-off is to have a single graph at join points in the control flow, but keep multiple graphs during the analysis of each basic block. In that case, the analysis splits the shape graph into configurations at the beginning at each basic block, then merges the configurations back into one single shape graph at the end of the block.

4.2 Refining the n_ϕ node

In our algorithm, the n_ϕ node models all concrete heap locations that are reachable from at least two stack variables on a path that does not go through any other root locations (i.e., locations pointed to directly from stack variables). Essentially, these locations are shared, being reachable from multiple variables. Merging all of these locations into one single abstract node is imprecise — such an abstraction cannot accurately describe the shape of the heap structure represented by these locations.

We can extend our algorithm with a more precise heap abstraction which refines the n_ϕ node. More precisely, we can replace the n_ϕ node with several *shared nodes* s_X, where X is a non-empty set of stack pointers. Each shared node s_X represents all of the locations reachable exactly from the variables in X, through at least two root nodes. This approach is similar to the abstractions used in previous shape analyses [3, 23], which distinguish between nodes based on the set of variables they are reachable from.

4.3 Multiple Selectors

Our current algorithm assumes one single selector name. This restricts the data structures that this algorithm can analyze to linked lists. We briefly describe how to extend our algorithm to support multiple selector names, allowing it to handle more complex structures such as binary trees.

We first extend our abstraction with information about selector names. If *Sel* represents the set of all selector names in the program, a shape graph for multiple selectors is a tuple $G = (N, E, C, S)$, where:

- N is the set of nodes.
- $E \in N \times N \rightarrow \mathcal{P}(Sel) \times \{0, 1/2, 1\}$ is the set of edges. Each edge contains information about the set of selector names it represents.
- $C \in N \times \mathcal{P}(Sel) \rightarrow \{0, 1/2, 1\}$ describes the cyclicity information for self-edges. A self-edge on a node n is cyclic on a set of selectors $\{f_1, \ldots f_m\}$ if there is a cycle over some concrete locations modeled by n, and that cycle uses only selectors in the set $\{f_1, \ldots f_m\}$.
- $S \in N \times \mathcal{P}(Sel) \rightarrow \{0, 1/2, 1\}$ describes the shared information for self-edges. A node n is shared for a set of selectors $\{f_1, \ldots f_n\}$ if it models a concrete heap location which be pointed to by multiple heap locations, though selectors in the set $\{f_1, \ldots f_n\}$.

This extension tracks the predicates C and S for each subset of Sel separately. It allows the analysis to accurately capture the shape of structures with multiple selectors. For instance, consider a binary tree with selectors `left` and `right`, with all of the leaves connected in a (non-cyclic) linked list with selector `next`. For this structure, the shared predicate S for $\{\texttt{left}, \texttt{next}\}$ and $\{\texttt{right}, \texttt{next}\}$ is non-zero; however, the shared predicate for $\{\texttt{left}, \texttt{right}\}$ of $\{\texttt{next}\}$ *is* zero, thus allows us to deduce that the selectors `left` and `right` define a tree structure, and the selector `next` defines a list.

The materialization and summarization operations are generalizations of the corresponding operations for a single selector. The analysis peforms precise summarization

for a statement `x = y->sel` on a self edge if the node being summarized is not cyclic on any set which includes the selector name `sel`, used to traverse the structure. Similarly, the analysis performs precise materialization for a statement `x = NULL` if the node that `x` points to is precisely reachable from exactly one other node.

Tracking both the cyclic and shared predicates allows us to distinguish certain types of graphs represented by a given abstract node n, based on possible combinations of the predicates, as shown in the following table:

	n is not cyclic	n is cyclic
n is not shared	tree	large cycle
n is shared	DAG	arbitrary graph

In this table, a large cycle represents a cycle which include of all of the locations that node n models.

5 Related Work

Early approaches to shape analysis [14, 1] propose graph abstractions of the heap based on allocation sites: each summary node in the shape graph represents all of the heap locations allocated at a certain site. Because of the abstraction based on allocation sites, these algorithms are imprecise in the presence of destructive updates.

A number of approaches to shape analyses have used access paths in the form of regular expressions to describe reachability of heap locations from the stack. Larus and Hilfinger present a dataflow analysis algorithm which computes access paths [15]. Other approaches use matrices of access paths to describe the reachability information [13, 12]. They propose algorithms that use the computed access paths to determine whether structures pointed to by different stack locations always access different heap locations, and use this information to parallelize applications that manipulate recursive heap structures. Researchers have also proposed language support for heap structures: in the Abstract Description of Data Structures (ADDS) [11], programmers can specify properties such as disjointness or backward pointers, and the compiler then uses analysis techniques based on access path matrices to check these properties. Deutsch [2] proposes a shape analysis which expresses aliasing using pairs of symbolic access paths. The analysis can parameterize the computed symbolic alias pairs, and show, for instance, that a list copy program produces a list whose elements are aliased with the corresponding elements of the original list.

Similar approaches use matrices of booleans to express reachability information [7, 8]. For instance, the interference matrix indicates whether there may be heap locations reachable from different stack locations; and the direction matrix indicates if the heap location pointed to by a stack variable may be reachable from the heap location pointed to by another variable. The analysis uses the reachability information in these matrices to distinguish between trees, DAGs, and arbitrary graphs.

A more sophisticated analysis proposes a shape graph abstraction of the heap which distinguishes between heap locations depending on the stack variables that point to them [20]. This approach keeps track of the sharedness of summary nodes to identify acyclic lists or tree structures. The algorithm also introduces two key techniques

that allow the analysis to compute accurate heap information: summarization into and materialization from summary nodes. Using these techniques, the analysis is able to determine that an in-place list reversal program preserves listness. Later analyses [23, 3] extend this algorithm with reachability information and thus are able to distinguish between sub-regions of the same heap structure. However, none of these algorithms is able to summarize heap access information for the whole execution of each procedure in the program.

More recently, researchers have proposed the use of three valued logic to express heap properties [21]. They propose a general framework which allows to express the heap abstraction using three valued logic formulas and show that existing analyses are instances of this framework. Subsequent work shows how to apply this framework to check the correctness of an insertion sort algorithm [17], and how to extend this framework for interprocedural analysis [18].

6 Conclusion

We have presented an inter-procedural analysis which is able to compute information about the program accesses heap regions within recursive data structures, such as sublists within lists. The analysis is designed to handle recursive programs which destructively update heap structures. We use a shape graph abstraction whose edges record reachability information and nodes record cyclicity information, and we formulate our algorithm as a dataflow analysis which computes shape and region information at each program point. As part of the analysis algorithm, we summarize the heap regions accessed by each procedure in terms of the regions at the beginning of the procedure. For this, we use labels on the nodes of the initial shape graph, to denote the regions as of the procedure entry point. Our analysis is able to accurately analyze a recursive quicksort program which sorts a list in-place, and determine that the two recursive calls access disjoint sublists within the list.

References

1. D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
2. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.
3. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proceedings of the 8th International Static Analysis Symposium*, Santa Barbara, CA, July 2000.
4. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.
5. D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

6. D. Gay and A. Aiken. Language support for regions. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, Snowbird, UT, June 2001.
7. R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.
8. 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.
9. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
10. N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the SIGPLAN '02 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
11. L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.
12. 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 Programming Language Design and Implementation*, Orlando, FL, June 1994.
13. L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.
14. N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the 9th Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1982.
15. J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.
16. C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. In *Proceedings of the SIGPLAN '02 Workshop on Memory System Performance*, Berlin, Germany, June 2002.
17. T. Lev-ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *2000 International Symposium on Software Testing and Analysis*, August 2000.
18. 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.
19. R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.
20. 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.
21. 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.
22. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.

23. R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proceedings of the 2000 International Conference on Compiler Construction*, Berlin, Germany, April 2000.
24. R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.