Precise Shape Analysis using Field Sensitivity

Sandeep Dasgupta*
Intel Technology India Pvt. Ltd.
sandeep.dasgupta@intel.com

Amey Karkare
Dept of CSE, IIT Kanpur
karkare@cse.iitk.ac.in

ABSTRACT

Programs in high level languages make intensive use of heap to support dynamic data structures. Analyzing these programs requires precise reasoning about the heap structures. Shape analysis refers to the class of techniques that statically approximate the run-time structures created on the heap. In this paper, we present a novel field sensitive shape analysis technique to identify the shapes of the heap structures. The novelty of our approach lies in the way we use field information to remember the paths that result in a particular shape (Tree, DAG, Cycle). We associate the field information with a shape in two ways: (a) through boolean functions that capture the shape transition due to change in a particular field, and (b) through matrices that store the field sensitive path information among two pointer variables. This allows us to easily identify transitions from Cycle to DAG, from Cycle to Tree and from DAG to Tree, thus making the shape more precise.

Categories and Subject Descriptors

F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; D.3.4 [**Programming Languages**]: Processors —*Compilers*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; E.1 [**Data**]: Data Structures—*Graphs and networks, trees*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs* ; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Algorithms, Languages, Verification, Theory

Keywords

Shape analysis, dataflow analysis, pointer analysis, static analysis, heap analysis

*This work was done when the first author was at IIT Kanpur.

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

1. INTRODUCTION

Shape analysis is the term for the class of static analysis techniques that are used to infer useful properties about heap data and the programs manipulating the heap. The shape information of a data structure accessible from a heap directed pointer can be used for disambiguating heap accesses originating from that pointer. This is useful for variety of applications, for e.g. compile time optimizations, compile-time garbage collection, debugging, verification, instruction scheduling and parallelization.

In the last two decades, several shape analysis techniques have been proposed in literature. However, there is a trade-off between speed and precision for these techniques. Precise shape analysis algorithms [14, 16, 6, 10] are not practical as they do not scale to the size of complex heap manipulating programs. To achieve scalability, the practical shape analysis algorithms [2, 7, 13] trade precision for speed.

In this paper, we present a shape analysis technique that uses limited field sensitivity to infer the shape of the heap. The novelty of our approach lies in the way we use field information to remember the paths that result in a particular shape (Tree, DAG, Cycle). This allows us to identify transitions from a conservative shape to a more precise shape (i.e., from Cycle to DAG, from Cycle to Tree and from DAG to Tree) due to destructive updates. This in turn enables us to infer precise shape information.

The field sensitivity information is captured in two ways: (a) we use field based boolean variables to remember the direct connections between two pointer variables, and (b) we compute field sensitive matrices that store the approximate path information between two pointer variable. We generate boolean functions at each program point that use the above field sensitive information to infer the shape of the pointer variables.

We discuss some of the prior works on shape analysis in Sect. 2. A motivating example is used in Sect. 3 to explain the intuition behind our analysis. The analysis is formalized in Sect. 4 that describes the notations used and in Sect. 5 that gives the analysis rules. Some properties of our analysis are described in Sect. 6. Section 7 concludes the presentation and gives directions for future work.

2. RELATED WORK

The shape-analysis problem was initially studied in the context of functional languages. Jones and Muchnick [11] proposed one of the earliest shape analysis technique for Lisp-like languages with destructive updates of structure. They used sets of finite shape graphs at each program point to describe the heap structure. To keep the shape graphs finite, they introduced the concept of *k*-limited graphs where all nodes beyond *k* distance from root of the graph are summarized into a single node. Hence the analysis re-

sulted in conservative approximations. The analysis is not practical as it is extremely costly both in time and space. Chase et. al. [2] introduced the concept of limited reference count to classify heap objects into different shapes. They also classified the nodes in concrete and summary nodes, where summary nodes were used to guarantee termination. Using the reference count and concreteness information of the node, they were able to kill relations (*strong updates*) for assignments of the form $p \rightarrow f = q$ in some cases. However, this information is not insufficient to compute precise shape, and detects false cycle even in case of simple algorithms like destructive list reversal.

Sagiv et. al. [14, 15] proposed generic, unbiased shape analysis algorithms based on *Three-Valued* logic. They introduce the concepts of *abstraction* and *re-materialization*. Abstraction is the process of summarizing multiple nodes into one and is used to keep the information bounded. Re-materialization is the process of obtaining concrete nodes from summary node and is required to handle destructive updates. By identifying suitable predicates to track, the analysis can be made very precise. However, the technique has potentially exponential run-time in the number of predicates, and therefore not suitable for large programs. Distefano et. al. [6] presented a shape analysis technique for linear data structures (linked-list etc.), which works on symbolic execution of the whole program using separation logic. Their technique works on suitable abstract domain, and guarantees termination by converting symbolic heaps to finite canonical forms, resulting in a fixed-point. By using enhanced abstraction scheme and predicate logic, Cherini et. al. [4] extended this analysis to support nonlinear data structure (tree, graph etc.). The abstraction is complex which will make verification of serious program expensive. On the other hand we propose a simpler abstraction that can be implemented in real compilers.

Berdine et. al. [1] proposed a method for identifying composite data structures using generic higher-order inductive predicates and parameterized spatial predicates. However, using of separation logic does not perform well in inference of heap properties. Hackett and Rugina in [10] presented a new approach for shape analysis which reasons about the state of a single heap location independently. This results in precise abstractions of localized portions of heap. This local reasoning is then used to reason about global heap using context-sensitive interprocedural analysis. Cherem et. al. [3] use the local abstraction scheme of [10] to generate local invariants to accurately compute shape information for complex data structures. Jump and McKinley [12] give a technique for dynamic shape analysis that characterizes the shape of recursive data structure in terms of dynamic degree metrics which uses in-degrees and out-degrees of heap nodes to categorize them into classes. While this technique is useful for detecting certain types of errors; it fails to visualize and understand the shape of heap structure and cannot express the sharing information in general.

Our work is closest to the work proposed by Ghiya et. al. [7] and by Marron et. al. [13]. Ghiya et. al. [7] keeps interference and direction matrices between any two pointer variables pointing to heap object and infer the shape of the structure as Tree, DAG or Cycle. They have demonstrated the practical applications of their analysis [8, 9] and shown that it works well on practical programs. The main shortcoming of this approach is that it cannot handle kill information. In particular, the approach is unable to identify transitions from Cycle to DAG, from Cycle to Tree and from DAG to Tree, and hence conservatively identify the shapes. Marron et. al. [13] presents a data flow framework that uses heap graphs to model data flow values. The analysis uses technique similar to re-materialization, but unlike parametric shape analysis techniques [15], the re-materialization is approximate and may result in loss of precision.

Our method is based on data flow analysis that uses matrices and boolean functions as data flow values. We use field sensitive matrices to store path information, and boolean variables to record field updates. By incorporating field sensitivity information, we are able to improve the precision without much impact on efficiency. The next section presents a simplified view of our approach before we explain it in full details.

3. A MOTIVATING EXAMPLE

Following the literature [7, 14, 13], we define the shape attribute for a pointer p as:

$$p.\text{shape} = \begin{cases} \texttt{Cycle} & \text{If a cycle can be reached from } p \\ \texttt{Dag} & \text{Else if a DAG can be reached from } p \\ \texttt{Tree} & \text{Otherwise} \end{cases}$$

where the heap is visualized as a directed graph, and cycle and DAG have there natural graph-theoretic meanings. For each pointer variable, our analysis computes the shape attribute of the data structure pointed to by the variable. We use the code fragment in Fig. 1(a) to motivate the need for a field sensitive shape analysis.

EXAMPLE 1. *Consider the code segment in the Fig. 1(a), At* S4*, a DAG is created that is reachable from p. At* S5*, a cycle is created that is reachable from both p and q. This cycle is destroyed at line* S6 *and the DAG is destroyed at* S7.

Field insensitive shape analysis algorithms use conservative kill information and hence they are, in general, unable to infer the shape transition from cycle to DAG or from DAG to Tree. For example, the algorithm by Ghiya et. al. [7] can correctly report the shape transition from DAG to cycle (at S5*), but fails to infer the shape transition from cycle to DAG (at* S6*) and from DAG to* Tree *(at* S7*). This is evident from Fig. 1(b) that shows the Direction (D) and Interference (I) matrices computed using their algorithm. We get conservative shape information at* S6 *and* S7 *because the kill-effect of statements* S6 *and* S7 *are not taken into account for computing D and I.* □

We now show how we have incorporated limited field sensitivity at each program point in our shape analysis. The details of our analysis will be presented later (Sect. 5).

EXAMPLE 2. *The statement at* S4 *creates a new DAG structure reachable from p, because there are two paths ($p \rightarrow f$ and $p \rightarrow h$) reaching q. Any field sensitive shape analysis algorithm must remember all paths from p to q. Our analysis approximates any path between two variables by the first field that is dereferenced on the path. Further, as there may be an unbounded number of paths between two variables, we use k-limiting to approximate the number of paths starting at a given field.*

Our analysis remembers the path information using the following: (a) D_F: Modified direction matrix that stores the first fields of the paths between two pointers; (b) I_F: Modified interference matrix that stores the pairs of first fields corresponding to the pairs of interfering paths, and (c) Boolean Variables that remember the fields directly connecting two pointer variables.

Figures 1(c) and 1(d) show the values computed by our analysis for the example program. In this case, the fact that the shape of the variable p becomes DAG after S4 *is captured by the following*

```
S1.  p = malloc();
S2.  q = malloc();
S3.  p→f = q;
S4.  p→h = q;
S5.  q→g = p;
S6.  q→g = NULL;
S7.  p→h = NULL;
```

(a) A code fragment

After Stmt	f_{pq}	h_{pq}	g_{qp}
S1	false	false	false
S2	false	false	false
S3	true	false	false
S4	true	true	false
S5	true	true	true
S6	true	true	false
S7	true	false	false

(d) Boolean Variables and their values

After Stmt	D			I		
		p	q		p	q
S1	p	1	0	p	1	0
	q	0	0	q	0	0
S2		p	q		p	q
	p	1	0	p	1	0
	q	0	1	q	0	1
S3		p	q		p	q
	p	1	1	p	1	1
	q	0	1	q	1	1
S4		p	q		p	q
	p	1	1	p	1	1
	q	0	1	q	1	1
S5		p	q		p	q
	p	1	1	p	1	1
	q	1	1	q	1	1
S6		p	q		p	q
	p	1	1	p	1	1
	q	1	1	q	1	1
S7		p	q		p	q
	p	1	1	p	1	1
	q	1	1	q	1	1

(b) Direction (D) and Interference (I) matrices [7]

After Stmt	D_F			I_F		
		p	q		p	q
S1	p	$\{\varepsilon\}$	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\emptyset$
	q	$\emptyset$	$\emptyset$	q	$\emptyset$	$\emptyset$
S2		p	q		p	q
	p	$\{\varepsilon\}$	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\emptyset$
	q	$\emptyset$	$\{\varepsilon\}$	q	$\emptyset$	$\{(\varepsilon,\varepsilon)\}$
S3		p	q		p	q
	p	$\{\varepsilon\}$	$\{f\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f,\varepsilon)\}$
	q	$\emptyset$	$\{\varepsilon\}$	q	$\{(\varepsilon,f)\}$	$\{(\varepsilon,\varepsilon)\}$
S4		p	q		p	q
	p	$\{\varepsilon\}$	$\{f,h\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f,\varepsilon),(h,\varepsilon)\}$
	q	$\emptyset$	$\{\varepsilon\}$	q	$\{(\varepsilon,f),(\varepsilon,h)\}$	$\{(\varepsilon,\varepsilon)\}$
S5		p	q		p	q
	p	$\{\varepsilon,f,h\}$	$\{f,h\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f,\varepsilon),(h,\varepsilon),(\varepsilon,g)\}$
	q	$\{g\}$	$\{\varepsilon,g\}$	q	$\{(\varepsilon,f),(\varepsilon,h),(g,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$
S6		p	q		p	q
	p	$\{\varepsilon\}$	$\{f,h\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f,\varepsilon),(h,\varepsilon)\}$
	q	$\emptyset$	$\{\varepsilon\}$	q	$\{(\varepsilon,f),(\varepsilon,h)\}$	$\{(\varepsilon,\varepsilon)\}$
S7		p	q		p	q
	p	$\{\varepsilon\}$	$\{f\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f,\varepsilon)\}$
	q	$\emptyset$	$\{\varepsilon\}$	q	$\{(\varepsilon,f)\}$	$\{(\varepsilon,\varepsilon)\}$

(c) Field Sensitive Direction (D_F) and Interference (I_F) matrices.

Figure 1: A motivating example

boolean functions[1] *:*

$$p_{\texttt{Dag}} = (h_{pq} \wedge (|I_F[p,q]| > 1)) \vee (f_{pq} \wedge (|I_F[p,q]| > 1)),$$
$$h_{pq} = \textbf{\textit{True}}.$$

Where h_{pq} is a boolean variable that is true if h field of p points to q, f_{pq} is a boolean variable that is true if f field of p points to q, I_F is field sensitive interference matrix, $|I_F[p,q]|$ is the count of number of interfering paths between p and q.

The functions simply say that variable p reaches a DAG because there are more than one paths ($|I_F[p,q]| > 1$) from p to q. It also keeps track of the paths (f_{pq} and h_{pq} in this case). Later, at statement S7*, the path due to h_{pq} is broken, causing $|I_F[p,q]| = 1$. This causes $p_{\texttt{Dag}}$ to become false. Note that we* do not *evaluate the boolean functions immediately, but associate the unevaluated functions with the statements. When we want to find out the shape at a given statement, only then we evaluate the function using the D_F and I_F matrices, and the values of boolean variables at that statement.*

Our analysis uses another attribute Cycle *to capture the cycles reachable from a variable. For our example program, assuming the absence of cycles before* S5*, the simplified functions for detecting cycle on p after* S5 *are:*

$$p_{\texttt{Cycle}} = g_{qp} \wedge (|D_F[p,q]| \geq 1),$$
$$g_{qp} = \textbf{\textit{True}}.$$

*Here, the functions captures the fact that cycle on p consists of field g from q to p (g_{qp}) and some path from p to q ($|D_F[p,q]| \geq 1$). This cycle is broken either when the path from p to q is broken ($|D_F[p,q]| = 0$) or when the link g changes ($g_{qp} =$ **False**). The latter occurs after* S6 *in Fig. 1(a).* □

We now formalize the intuitions presented in the example above.

4. DEFINITIONS AND NOTATIONS

We view the heap structure at a program point as a directed graph, the nodes of which represent the allocated objects and the edges represent the connectivity through pointer fields. Pictorially, inside a node we show all the relevant pointer variables that can point to the heap object corresponding to that node. The edges are labeled by the name of the corresponding pointer field. In this paper, we only label nodes and edges that are relevant to the discussion, to avoid clutter.

Let $\mathcal{H}$ denotes the set of all heap directed pointers at a particular program point and $\mathcal{F}$ denotes the set of all pointer fields at that program point. Given two heap-directed pointers $p, q \in \mathcal{H}$, a path from p to q is the sequence of pointer fields that need to be traversed in the heap to reach from p to q. The length of a path is defined as the number of pointer fields in the path. As the path length between two heap objects may be unbounded, we keep track of only the first field of a path[2] To distinguish between a path of length one (direct path) from a path of length greater than one (indirect path) that start at the same field, we use the superscript D for a direct path and I for an indirect path. In pictures, we use solid edges for direct paths, and dotted edges for indirect paths.

It is also possible to have multiple paths between two pointers starting at a given field f, with at most one direct path f^D. However, the number of indirect paths f^I may be unbounded. As there can only be a finite number of first fields, we store first fields of paths, including the count for the indirect paths, between two pointer variables in a set. To bound the size of the set, we put a limit k on number of repetitions of a particular field. If the number goes beyond k, we treat the number of paths with that field as ∞.

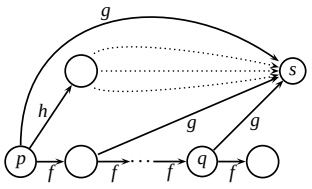
EXAMPLE 3. *Figure 2(a) shows a code fragment and Fig. 2(b) shows a possible heap graph at a program point after line* S5*. In any execution, there is one path between p and q, starting with field f, whose length is statically unknown. This information is stored by our analysis as the set $\{f^{I1}\}$. Further, there are unbounded number of paths between p and s, all starting with field f. There is also a direct path from p to s using field g, and 3 paths starting with field h between p and s. Assuming the limit $k \geq 3$, this information can be*

[1]The functions and values shown in this example and in Fig. 1(c) are simplified to avoid references to concepts not defined yet.

[2]The decision to use only first field is guided by the fact that in our language, a statement can use at most one field, i.e. `p→f = ...` or `...= p→f`. Using prefixes of a fixed length > 1 increases the precision of the analysis at the cost of increased complexity. However, it does not make any fundamental change to the analysis.

```
S1.  q = p;
S2.  while(...)  {
S3.      q→g = s;
S4.      q = q→f;
S5.  }
```

(a) A code fragment



(b) A possible heap graph for code in (a). Solid edges are the direct paths, dotted edges are the indirect paths.

Figure 2: Paths in a heap graph

represented by the set $\{g^D, f^{I\infty}, h^{I3}\}$. *On the other hand, if* $k < 3$, *then the set would be* $\{g^D, f^{I\infty}, h^{I\infty}\}$. □

For brevity, we use f^* for the cases when we do not want to distinguish between direct or indirect path starting at the first field f. We now define the field sensitive matrices used by our analysis.

DEFINITION 1. *Field sensitive Direction matrix* D_F *is a matrix that stores information about paths between two pointer variables. Given* $p,q \in \mathcal{H}, f \in \mathcal{F}$*:*

ε	$\in D_F[p,p]$	*where* ε *denotes the empty path.*
f^D	$\in D_F[p,q]$	*if there is a direct path* f *from* p *to* q.
f^{Im}	$\in D_F[p,q]$	*if there are* m *indirect paths starting with field* f *from* p *to* q *and* $m \leq k$.
$f^{I\infty}$	$\in D_F[p,q]$	*if there are* m *indirect paths starting with field* f *from* p *to* q *and* $m > k$.

Let $\mathcal{N}$ denote the set of natural numbers. We define the following partial order for approximate paths used by our analysis. For $f \in \mathcal{F}$, $m,n \in \mathcal{N}$, $n \leq m$:

$$\varepsilon \sqsubseteq \varepsilon, \quad f^D \sqsubseteq f^D, \quad f^{I\infty} \sqsubseteq f^{I\infty}, \quad f^{Im} \sqsubseteq f^{I\infty}, \quad f^{In} \sqsubseteq f^{Im}.$$

The partial order is extended to set of paths S_{P_1}, S_{P_2} as[3]:

$$S_{P_1} \sqsubseteq S_{P_2} \quad \Leftrightarrow \quad \forall \alpha \in S_{P_1}, \exists \beta \in S_{P_2}\ s.t. \alpha \sqsubseteq \beta$$

For pair of paths: $(\alpha,\beta) \sqsubseteq (\alpha',\beta') \Leftrightarrow (\alpha \sqsubseteq \alpha') \wedge (\beta \sqsubseteq \beta')$, and for set of pairs of paths R_{P_1}, R_{P_2}: $R_{P_1} \sqsubseteq R_{P_2} \Leftrightarrow \forall (\alpha,\beta) \in R_{P_1}, \exists (\alpha',\beta') \in R_{P_2}\ s.t. (\alpha,\beta) \sqsubseteq (\alpha',\beta')$.

Two pointers $p,q \in \mathcal{H}$ are said to interfere if there exists $s \in \mathcal{H}$ such that both p and q have paths reaching s. Note that s could be p (or q) itself, in which case the path from p (from q) is ε.

DEFINITION 2. *Field sensitive Interference matrix* I_F *between two pointers captures the ways in which these pointers are interfering. For* $p,q,s \in \mathcal{H}, p \neq q$, *the following relation holds for* D_F *and* I_F*:*

$$D_F[p,s] \times D_F[q,s] \quad \sqsubseteq \quad I_F[p,q]$$

Our analysis computes over-approximations for the matrices D_F and I_F at each program point. While it is possible to compute only D_F and use above equation to compute I_F, computing both explicitly results in better approximations for I_F. Note that interference relation is symmetric, i.e.,

$$(\alpha,\beta) \in I_F[p,q] \Leftrightarrow (\beta,\alpha) \in I_F[q,p]$$

While describing the analysis, we use the above relation to show the computation of only one of the two entries.

[3]Note that for our analysis, for a given field f, these sets contain at most one entry of type f^D and at most one entry of type f^I

Table 1: Determining shape from boolean attributes

$p_{\texttt{Cycle}}$	$p_{\texttt{Dag}}$	p.shape
True	Don't Care	Cycle
False	**True**	DAG
False	**False**	Tree

EXAMPLE 4. *Figure 3 shows a heap graph and the corresponding field sensitive matrices as computed by our analysis.* □

As mentioned earlier, for each variable $p \in \mathcal{H}$, our analysis uses attributes $p_{\texttt{Dag}}$ and $p_{\texttt{Cycle}}$ to store boolean functions telling whether p can reach a DAG or cycle respectively in the heap. The boolean functions consist of the values from matrices D_F, I_F, and the field connectivity information. For $f \in \mathcal{F}, p,q \in \mathcal{H}$, field connectivity is captured by boolean variables of the form f_{pq}, which is true when f field of p points directly to q. The shape of p, p.shape, can be obtained by evaluating the functions for the attributes $p_{\texttt{Cycle}}$ and $p_{\texttt{Dag}}$, and using Table 1.

We use the following operations in our analysis. Let S denote the set of approximate paths between two nodes, P denote a set of pair of paths, and $k \in \mathcal{N}$ denotes the limit on maximum indirect paths stored for a given field. Then,

- Projection: For $f \in \mathcal{F}$, $S \triangleright f$ extracts the paths starting at field f.

$$S \triangleright f \equiv S \cap \{f^D, f^{I1}, \ldots, f^{Ik}, f^{I\infty}\}$$

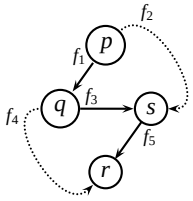
- Counting: The count on the number of paths is defined as :

$$\begin{aligned} |\varepsilon| &= 1 \\ |f^D| &= 1 \\ |f^{I\infty}| &= \infty \\ |f^{Ij}| &= j \text{ for } j \in \mathcal{N} \\ |S| &= \sum_{\alpha \in S} |\alpha| \end{aligned}$$

- Path removal, intersection and union over set of approximate paths : For singleton sets of paths $\{\alpha\}$ and $\{\beta\}$, path removal ($\{\alpha\} \ominus \{\beta\}$), intersection ($\{\alpha\} \cap \{\beta\}$) and union($\{\alpha\} \cup \{\beta\}$) operations are defined as given in Table 2. These definitions are extended to set of paths in a natural way [5].

- Multiplication by a scalar($\star$): Let $i,j \in \mathcal{N}, i \leq k, j \leq k$. Then, for a path α, the multiplication by a scalar i, $i \star \alpha$ is defined in Table 2(d). The operation is extended to set of paths as:

$$i \star S \quad = \quad \begin{cases} \emptyset & i = 0 \\ \{i \star \alpha \mid \alpha \in S\} & i \in \mathcal{N} \cup \{\infty\} \end{cases}$$



(a) Heap graph

D_F	p	q	s	r
p	$\{\varepsilon\}$	$\{f_1^D\}$	$\{f_1^{I1}, f_2^{I1}\}$	$\{f_1^{I2}, f_2^{I1}\}$
q	$\emptyset$	$\{\varepsilon\}$	$\{f_3^D\}$	$\{f_3^{I1}, f_4^{I1}\}$
s	$\emptyset$	$\emptyset$	$\{\varepsilon\}$	$\{f_5^D\}$
r	$\emptyset$	$\emptyset$	$\emptyset$	$\{\varepsilon\}$

(b) Direction Matrix

I_F	p	q	s	r
p	$\{\varepsilon,\varepsilon\}$	$\{(f_1^D,\varepsilon), (f_2^{I1}, f_3^D), (f_2^{I1}, f_4^{I1})\}$	$\{(f_1^{I1},\varepsilon), (f_2^{I1},\varepsilon), (f_1^{I1}, f_5^D)\}$	$\{(f_1^{I2},\varepsilon), (f_2^{I1},\varepsilon)\}$
q	$\{(\varepsilon, f_1^D), (f_3^D, f_2^{I1}), (f_4^{I1}, f_2^{I1})\}$	$\{\varepsilon,\varepsilon\}$	$\{(f_3^D,\varepsilon), (f_4^{I1}, f_5^D)\}$	$\{(f_3^{I1},\varepsilon), (f_4^{I1},\varepsilon)\}$
s	$\{(\varepsilon, f_1^{I1}), (\varepsilon, f_2^{I1})\}$ $(f_5^D, f_1^{I1})\}$	$\{(\varepsilon, f_3^D), (f_5^D, f_4^{I1})\}$	$\{\varepsilon,\varepsilon\}$	$\{(f_5^D,\varepsilon)\}$
r	$\{(\varepsilon, f_1^{I2}), (\varepsilon, f_2^{I1})\}$	$\{(\varepsilon, f_3^{I1}), (\varepsilon, f_4^{I1})\}$	$\{(\varepsilon, f_5^D)\}$	$\{\varepsilon,\varepsilon\}$

(c) Interference Matrix

Figure 3: A heap graph and its field sensitive path matrices

Table 2: Path operations

(a) Path removal

$\ominus$ $\{\alpha\}$ \ $\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	Any other path ($\{\gamma\}$)
$\{\varepsilon\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$
$\{f^D\}$	$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\{f^D\}$	$\{f^D\}$
$\{f^{Ii}\}$	$\{f^{Ii}\}$	$\emptyset$	$\{f^{Im}\}$	$\emptyset$	$\{f^{Ii}\}$
$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\emptyset$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$

(b) Intersection

$\cap$ $\{\alpha\}$ \ $\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	Any other path ($\{\gamma\}$)
$\{\varepsilon\}$	ε	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^{Ii}\}$	$\emptyset$	$\emptyset$	$\{f^{In}\}$	$\{f^{Ii}\}$	$\emptyset$
$\{f^{I\infty}\}$	$\emptyset$	$\emptyset$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	$\emptyset$

(c) Union

$\cup$ $\{\alpha\}$ \ $\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	Any other path ($\{\gamma\}$)
$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon, f^D\}$	$\{\varepsilon, f^{Ij}\}$	$\{\varepsilon, f^{I\infty}\}$	$\{\varepsilon,\gamma\}$
$\{f^D\}$	$\{f^D,\varepsilon\}$	$\{f^D\}$	$\{f^D, f^{Ij}\}$	$\{f^D, f^{I\infty}\}$	$\{f^D,\gamma\}$
$\{f^{Ii}\}$	$\{f^{Ii},\varepsilon\}$	$\{f^{Ii}, f^D\}$	$\{f^{It}\}$	$\{f^{I\infty}\}$	$\{f^{Ii},\gamma\}$
$\{f^{I\infty}\}$	$\{f^{I\infty},\varepsilon\}$	$\{f^{I\infty}, f^D\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty},\gamma\}$

$$i, j \in \mathcal{N},\ m = \texttt{max}(i-j, 0),\ n = \texttt{min}(i, j) \text{ and } t = \begin{cases} i+j & \text{if } i+j \le k \\ \infty & \text{Otherwise} \end{cases}.$$

(d) Multiplication by a scalar

$\star$ i \ α	ε	f^D	f^{Ij}	$f^{I\infty}$
i	ε	f^{Ii}	f^{Im}, $m = \begin{cases} i*j & \text{if } i*j \le k \\ \infty & \text{Otherwise} \end{cases}$	$f^{I\infty}$
∞	ε	$f^{I\infty}$	$f^{I\infty}$	$f^{I\infty}$

5. OUR ANALYSIS

Our intent is to determine, at each program point, the field sensitive matrices D_F and I_F, and the boolean variables capturing field connectivity. We formulate the problem as an instance of forward data flow analysis, where the data flow values are the matrices and the boolean variables as mentioned above. For simplicity, we construct basic blocks containing a single statement each. Before presenting the equations for data flow analysis, we define the confluence operator (merge) for various data flow values as used by our analysis. Using the superscripts x and y to denote the values coming along two paths,

$$\begin{aligned}
\text{merge}(f_{pq}^x, f_{pq}^y) &= f_{pq}^x \vee f_{pq}^y, f \in \mathcal{F}, p, q \in \mathcal{H} \\
\text{merge}(p_{\texttt{Cycle}}^x, p_{\texttt{Cycle}}^y) &= p_{\texttt{Cycle}}^x \vee p_{\texttt{Cycle}}^y, p \in \mathcal{H} \\
\text{merge}(p_{\texttt{Dag}}^x, p_{\texttt{Dag}}^y) &= p_{\texttt{Dag}}^x \vee p_{\texttt{Dag}}^y, p \in \mathcal{H} \\
\text{merge}(D_F^x, D_F^y) &= D_F \text{ where } D_F[p,q] = \\
&\quad D_F^x[p,q] \cup D_F^y[p,q], \forall p, q \in \mathcal{H} \\
\text{merge}(I_F^x, I_F^y) &= I_F \text{ where } I_F[p,q] = \\
&\quad I_F^x[p,q] \cup I_F^y[p,q], \forall p, q \in \mathcal{H}
\end{aligned}$$

The transformation of data flow values due to a statement *st* is captured by the following set of equations:

$$\begin{aligned}
D_F^{out}[p,q] &= (D_F^{in}[p,q] \ominus D_F^{kill}[p,q]) \cup D_F^{gen}[p,q] \\
I_F^{out}[p,q] &= (I_F^{in}[p,q] \ominus I_F^{kill}[p,q]) \cup I_F^{gen}[p,q] \\
p_{\texttt{Cycle}}^{out} &= (p_{\texttt{Cycle}}^{in} \wedge \neg p_{\texttt{Cycle}}^{kill}) \vee p_{\texttt{Cycle}}^{gen} \\
p_{\texttt{Dag}}^{out} &= (p_{\texttt{Dag}}^{in} \wedge \neg p_{\texttt{Dag}}^{kill}) \vee p_{\texttt{Dag}}^{gen}
\end{aligned}$$

Field connectivity information is updated directly by the statement.

5.1 Analysis of Basic Statements

We now present the basic statements that can access or modify the heap structures, and our analysis of each kind of statements.

1. `p = malloc()`: After this statement p points to a newly allocated object. Thus,

$$p_{\texttt{Cycle}}^{kill} = p_{\texttt{Cycle}}^{in} \qquad p_{\texttt{Dag}}^{kill} = p_{\texttt{Dag}}^{in}$$
$$p_{\texttt{Cycle}}^{gen} = \textbf{False} \qquad p_{\texttt{Dag}}^{gen} = \textbf{False}$$

$\forall s \in \mathcal{H},\ s \neq p,$

$$D_F^{kill}[p,s] = D_F^{in}[p,s] \qquad D_F^{gen}[p,s] = \emptyset$$
$$D_F^{kill}[s,p] = D_F^{in}[s,p] \qquad D_F^{gen}[s,p] = \emptyset$$
$$D_F^{kill}[p,p] = D_F^{in}[p,p] \qquad D_F^{gen}[p,p] = \{\varepsilon\}$$
$$I_F^{kill}[p,s] = I_F^{in}[p,s] \qquad I_F^{gen}[p,s] = \emptyset$$
$$I_F^{kill}[p,p] = I_F^{in}[p,p] \qquad I_F^{gen}[p,p] = \{\varepsilon,\varepsilon\}$$

2. `p = NULL`: This statement only kills the existing relations of p.

$$p_{\texttt{Cycle}}^{kill} = p_{\texttt{Cycle}}^{in} \qquad p_{\texttt{Dag}}^{kill} = p_{\texttt{Dag}}^{in}$$
$$p_{\texttt{Cycle}}^{gen} = \textbf{False} \qquad p_{\texttt{Dag}}^{gen} = \textbf{False}$$

$\forall s \in \mathcal{H},$

$$D_F^{kill}[p,s] = D_F^{in}[p,s] \qquad D_F^{gen}[p,s] = \emptyset$$
$$D_F^{kill}[s,p] = D_F^{in}[s,p] \qquad D_F^{gen}[s,p] = \emptyset$$
$$I_F^{kill}[p,s] = I_F^{in}[p,s] \qquad I_F^{gen}[p,s] = \emptyset$$

3. `p = q, p = &(q→f), p = q op n`: In our analysis we consider these three pointer assignment statements as equivalent. After these statements, p is assumed to point to the same heap object as pointed by q.

$$p_{\texttt{Cycle}}^{kill} = p_{\texttt{Cycle}}^{in} \qquad p_{\texttt{Dag}}^{kill} = p_{\texttt{Dag}}^{in}$$
$$p_{\texttt{Cycle}}^{gen} = q_{\texttt{Cycle}}^{in}[q/p] \qquad p_{\texttt{Dag}}^{gen} = q_{\texttt{Dag}}^{in}[q/p]$$

where $X[q/p]$ creates a copy of X with all occurrences of q replaced by p.

$\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F}$

$$f_{ps} = f_{qs} \qquad f_{sp} = f_{sq}$$
$$D_F^{kill}[p,s] = D_F^{in}[p,s] \qquad D_F^{gen}[p,s] = D_F^{in}[q,s]$$
$$D_F^{kill}[s,p] = D_F^{in}[s,p] \qquad D_F^{gen}[s,p] = D_F^{in}[s,q]$$
$$D_F^{kill}[p,p] = D_F^{in}[p,p] \qquad D_F^{gen}[p,p] = D_F^{in}[q,q]$$
$$I_F^{kill}[p,s] = I_F^{in}[p,s] \qquad I_F^{gen}[p,s] = I_F^{in}[q,s]$$
$$I_F^{kill}[p,p] = I_F^{in}[p,p] \qquad I_F^{gen}[p,p] = I_F^{in}[q,q]$$

4. `p→f = null`: This statement breaks the existing link f emanating from p, thus killing relations of p, that are due to link f. The statement does not generate new relations.

$$p_{\texttt{Cycle}}^{kill} = \textbf{False}, \qquad p_{\texttt{Dag}}^{kill} = \textbf{False}$$
$$p_{\texttt{Cycle}}^{gen} = \textbf{False}, \qquad p_{\texttt{Dag}}^{gen} = \textbf{False}$$

$\forall q,s \in \mathcal{H}, s \neq p,$

$$\begin{aligned}
f_{pq} &= \textbf{False} \\
D_F^{kill}[p,q] &= D_F^{in}[p,q] \triangleright f \qquad D_F^{kill}[s,q] = \emptyset \\
I_F^{kill}[p,s] &= \{(\alpha,\beta) \mid (\alpha,\beta) \in I_F^{in}[p,s],\ \alpha \equiv f^*\} \\
I_F^{kill}[q,s] &= \emptyset \text{ if } q \neq p \qquad I_F^{kill}[p,p] = \emptyset
\end{aligned}$$

5. `p→f = q`: This statement first breaks the existing link f and then re-links the the heap object pointed to by p to the heap object pointed to by q. The kill effects are exactly same as described in the case of `p→f = null`. We only describe the generated relationships here:

$$\begin{aligned}
p_{\texttt{Cycle}}^{gen} &= (f_{pq} \wedge q_{\texttt{Cycle}}^{in}) \vee (f_{pq} \wedge (|D_F[q,p]| \geq 1)) \\
p_{\texttt{Dag}}^{gen} &= f_{pq} \wedge (|I_F[p,q]| > 1) \\
q_{\texttt{Cycle}}^{gen} &= f_{pq} \wedge (|D_F[q,p]| \geq 1) \\
q_{\texttt{Dag}}^{gen} &= \textbf{False} \\
f_{pq} &= \textbf{True} \\
s_{\texttt{Cycle}}^{gen} &= ((|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge q_{\texttt{Cycle}}^{in}) \\
&\quad \vee ((|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|D_F[q,p]| \geq 1)) \\
&\quad \vee ((|D_F[s,q]| \geq 1) \wedge f_{pq} \wedge (|D_F[q,p]| \geq 1)), \\
&\qquad \forall s \in \mathcal{H}, s \neq p, s \neq q \\
s_{\texttt{Dag}}^{gen} &= (|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|I_F[s,q]| > 1), \\
&\qquad \forall s \in \mathcal{H}, s \neq p, s \neq q
\end{aligned}$$

The relations generated for D_F and I_F are as follows. For $r,s \in \mathcal{H}$:

$$\begin{aligned}
D_F^{gen}[r,s] &= |D_F^{in}[q,s]| \star D_F^{in}[r,p],\ s \neq p,\ r \notin \{p,q\} \\
D_F^{gen}[r,p] &= |D_F^{in}[q,p]| \star D_F^{in}[r,p],\ r \neq p \\
D_F^{gen}[p,r] &= |D_F^{in}[q,r]| \star (D_F^{in}[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}), \\
&\qquad r \neq q \\
D_F^{gen}[p,q] &= \{f^D\} \cup (|D_F^{in}[q,q] - \{\varepsilon\}| \star \{f^{I1}\}) \cup \\
&\quad (|D_F^{in}[q,q]| \star (D_F^{in}[p,p] \ominus \{\varepsilon\})) \\
D_F^{gen}[q,q] &= 1 \star D_F^{in}[q,p] \\
D_F^{gen}[q,r] &= |D_F^{in}[q,r]| \star D_F^{in}[q,p],\ r \notin \{p,q\} \\
I_F^{gen}[p,q] &= \{(f^D,\varepsilon)\} \cup ((1 \star (D_F^{in}[p,p] \ominus \{\varepsilon\})) \times \{\varepsilon\}) \\
I_F^{gen}[p,r] &= (1 \star (D_F^{in}[p,p] \ominus \{\varepsilon\})) \times \\
&\quad \{\beta \mid (\alpha,\beta) \in I_F^{in}[q,r]\} \\
&\quad \cup \{f^D\} \times \{\beta \mid (\varepsilon,\beta) \in I_F^{in}[q,r]\} \\
&\quad \cup \{f^{I1}\} \times \{\beta \mid (\alpha,\beta) \in I_F^{in}[q,r], \alpha \neq \varepsilon\}, \\
&\qquad r \notin \{p,q\} \\
I_F^{gen}[s,q] &= (1 \star D_F^{in}[s,p]) \times \{\varepsilon\},\ s \notin \{p,q\} \\
I_F^{gen}[s,r] &= (1 \star D_F^{in}[s,p]) \times \{\beta \mid (\alpha,\beta) \in I_F^{in}[q,r]\}, \\
&\qquad s \notin \{p,q\},\ r \notin \{p,q\},\ s \neq r
\end{aligned}$$

6. `p = q→f`: The relations killed by the statement are same as that in case of `p = NULL`. The relations created by this statement are heavily approximated by our analysis. After this statement p points to the heap object which is accessible from pointer q through f link. The only inference we can draw is that p is reachable from any pointer r such that r reaches $q \rightarrow f$ before the assignment. This information is available because $I_F^{in}[q,r]$ will have an entry of the form (f^D,α) for some α.

As p could potentially point to a cycle(DAG) reachable from q, we set:

$$p_{\texttt{Cycle}}^{gen} = q_{\texttt{Cycle}}^{in} \qquad p_{\texttt{Dag}}^{gen} = q_{\texttt{Dag}}^{in}$$

We record the fact that q reaches p through the path f. Also, any object reachable from q using field f is marked as reachable from p through any possible field.

$$\begin{aligned} f_{qp} &= \textbf{True} \\ h_{pr} &= |D_F^{in}[q,r] \triangleright f| \geq 1 \quad \forall h \in \mathcal{F}, \forall r \in \mathcal{H} \end{aligned}$$

The equations to compute the generated relations for D_F and I_F can be divided into three components. We explain each of the component, and give the equations.

As a side-effect of the statement, any node s that is reachable from q through field f before the statement, becomes reachable from p. However, the information available is not sufficient to determine the path from p to s. Therefore, we conservatively assume that any path starting from p can potentially reach s. This is achieved in the analysis by using a universal path set $\mathcal{U}$ for $D_F[p,s]$. The set $\mathcal{U}$ is defined as:

$$\mathcal{U} = \{\varepsilon\} \cup \bigcup_{f \in \mathcal{F}} \{f^D, f^{I\infty}\}$$

Because it is also not possible to determine if there exist a path from p to itself, we safely conclude a self loop on p in case a cycle is reachable from q (i.e., q.shape evaluates to `Cycle`). These observations result in the following equations:

$$\begin{aligned} D_1[p,s] &= \mathcal{U} \quad \forall s \in \mathcal{H}, s \neq p \wedge D_F^{in}[q,s] \triangleright f \neq \emptyset \\ D_1[p,p] &= \begin{cases} \mathcal{U} & q\text{.shape evaluates to } \texttt{Cycle} \\ \{\varepsilon\} & \text{Otherwise} \end{cases} \\ I_1[p,p] &= \mathcal{U} \times \mathcal{U} \end{aligned}$$

Any node s (including q), that has paths to q before the statement, will have paths to p after the statement. However, we can not know the exact number of paths s to p, and therefore use upper limit (∞) as an approximation:

$$\begin{aligned} D_2[s,p] &= \infty \star D_F^{in}[s,q] \quad \forall s \in \mathcal{H}, s \neq q \\ D_2[q,p] &= \{f^D\} \cup (\infty \star (D_F^{in}[q,q] \ominus \{\varepsilon\})) \cup \mathcal{U} \\ I_2[s,p] &= D_2[s,p] \times \{\varepsilon\} \quad \forall s \in \mathcal{H} \end{aligned}$$

The third category of nodes to consider are those that interfere with the node reachable from q using direct path f. Such a node s will have paths to p after the statement. Also the nodes that interfere with the node reachable from q using direct or indirect path f will interfere with p after the statement. Thus, we have:

$$\begin{aligned} D_3[s,p] &= \{\alpha \mid (f^D, \alpha) \in I_F^{in}[q,s]\} \\ I_3[s,p] &= \{\alpha \mid (f^*, \alpha) \in I_F^{in}[q,s]\} \times \mathcal{U} \end{aligned}$$

Finally, we compute the I_F and D_F relations as:

$$\begin{aligned} D_F^{gen}[r,s] &= D_1[r,s] \cup D_2[r,s] \cup D_3[r,s] \quad \forall r,s \in \mathcal{H} \\ I_F^{gen}[r,s] &= I_1[r,s] \cup I_2[r,s] \cup I_3[r,s] \quad \forall r,s \in \mathcal{H} \end{aligned}$$

5.2 Interprocedural Analysis

To handle procedure calls we use simple interprocedural analysis that works by creating an invocation graph of the program. To handle recursive calls, for which the invocation structure is statically unknown, we use approximate summary for one of the nodes involved in the recursion chain, and use it to break the cycle. At each call site, the two matrices (D_F and I_F) along with the boolean functions are fed as input to the called procedure, after proper mapping between formal and actual arguments. The called procedure is then analyzed to create the corresponding output matrices and the boolean functions. This approach is similar to the work by Ghiya et. al. [7].

6. PROPERTIES OF OUR ANALYSIS

We mention some of the properties of our analysis. The details have been omitted due to space constraints, and can be found in [5]. We also show some of the cases where it performs better than the field insensitive approaches.

6.1 Need for Boolean Variables

Because we compute approximations for field sensitive matrices under certain conditions (e.g. for statement $p = q \rightarrow f$), these matrices can result in imprecise shape. Boolean variables help us retain some precision in such cases.

6.2 Termination

The computation of D_F and I_F matrices follows from the fact that the data flow functions are monotonic and the sets of approximate paths are bounded. The termination of computation of boolean functions for `Cycle` and `Dag` can be proved using the associativity and distributivity of the boolean operators ($\wedge$ and $\vee$). The details of the proof can be found in [5].

6.3 Storage Requirement

The space requirement for D_F is $O(n^2 * m)$ and that for I_F is $O(n^2 * m^2)$. The boolean functions at each program point are stored in an expression tree, the size of which is polynomial in the number of pointer instructions.

6.4 Comparison with a Field Insensitive Approach

For comparison purpose the test cases must involve shape transitions like Cycle to DAG, Cycle to Tree, and DAG to Tree. The transition like Tree to DAG, DAG to Cycle, or Tree to Cycle are not of much importance as these can be detected by any of the field insensitive approaches. Following are the cases that meet our requirement and better demonstrate the accuracy of our analysis as compared to field insensitive analysis (like Ghiya et. al. [7]).

(a) **Inserting an internal node in a singly linked list:** Consider the code fragment Fig. 4(a) that is a simplified version of insertion of an internal node in a linked list. Field insensitive approach like that of Ghiya et. al. [7] cannot detect the kill information due to the change of the field f of `p` at `S4` and finds `p` to have an additional path to q via `r` (which is now actually the only path). So they report the shape attribute of `p` as DAG.

Consider the following boolean function generated after `S4` using our approach.

$$\begin{aligned} p_{\texttt{Dag}} &= (f_{pq} \wedge (|I_F[pq]| > 1)) \vee (f_{pr} \wedge (|I_F[pr]| > 1)) \\ f_{pr} &= \textbf{True} \qquad f_{pq} = \textbf{False} \end{aligned}$$

After `S4`, the condition $|I_F[p,r]| > 1$ become **False** and $p_{\texttt{Dag}}$ will get evaluated to **False**, and thus correctly detects the shape attribute of `p` to tree.

(b) **Swapping two nodes of a singly linked list:** Consider the code fragment Fig. 4(b) which swaps the two pointers p and $p \rightarrow f$ in a singly linked list L with link field as f, given

```
S1.  p→f = q;
S2.  r= malloc();
S3.  r→f = q;
S4.  p→f = r;
```

After	Actual Shape	Field Insensitive Analysis	Field Sensitive Analysis
S1	Tree	Tree	Tree
S2	Tree	Tree	Tree
S3	Tree	Tree	Tree
S4	Tree	DAG(at p)	Tree

(a) Insertion of an internal node in a singly linked list

```
S1.  n1 = p→f;
S2.  n2 = n1→f;
S3.  t = n2→f;
S4.  n2→f = n1;
S5.  n1→f = t;
S6.  p→f = n2;
```

After	Actual Shape	Field Insensitive Analysis	Field Sensitive Analysis
S1	Tree	Tree	Tree
S2	Tree	Tree	Tree
S3	Tree	Tree	Tree
S4	Cycle (at p, n1, n2)	Cycle (at p, n1, n2)	Cycle (at p, n1, n2)
S5	Tree	Cycle (at p, n1, n2)	Tree
S6	Tree	Cycle (at p, n1, n2)	Tree

(b) Swapping two nodes of a singly linked list

```
mirror(tree T) {
S1.  L = T->left;
S2.  R = T->right;
S3.  mirror(L);
S4.  mirror(R);
S5.  T->left = R;
S6.  T->right = L;
}
```

After	Actual Shape	Field Insensitive Analysis	Field Sensitive Analysis
S1	Tree	Tree	Tree
S2	Tree	Tree	Tree
S5	Dag (at T)	Dag (at T)	Dag (at T)
S6	Tree	Dag (at T)	Tree

(c) Computing mirror image of a binary tree.

Figure 4: Examples demonstrating the preciseness of our analysis

the pointer *p*. The table in Fig. 4(b) shows the comparison between the shape decision given by our approach and the field insensitive approaches.

(c) **Computing mirror image of a binary tree:** Consider the code fragment Fig. 4(c) which creates a mirror image of a binary tree rooted at *T*. While swapping the left and right sub-tree a temporary DAG is created (after statement `S6`), which gets destroyed after the very next statement `S7`. As depicted in Fig. 4(c), this shape transition is also captured in our analysis.

7. CONCLUSION AND FUTURE WORK

In this paper we proposed a field sensitive shape analysis technique. We demonstrated how boolean functions along with field sensitive matrices help in inferring the precise shape of the data structure. While field sensitive matrices help in generating the kill information for strong updates, boolean functions help in remembering the shape transition history with respect to each heap-directed pointer. We have shown some example scenarios that can be handled more precisely by our analysis as compared to an existing field insensitive analysis. Our shape analysis can be utilized by an optimizing compiler to disambiguate memory references.

We use a very simple inter procedural framework to handle function calls, that computes safe approximate summaries to reach fix point. Our next challenge is to develop a better inter procedural analysis to handle function calls more precisely. Further, we plan to extend our shape analysis technique to handle more of frequently occurring programming patterns to find precise shape for these patterns. We are developing a prototype model using GCC framework to show the effectiveness on large benchmarks. However, this work is still in very early stages, and requires manual intervention. We plan to automate the prototype in near future.

8. REFERENCES

[1] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'hearn, H. Yang, and Q. Mary. Shape analysis for composite data structures. In *CAV '07*.

[2] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90*.

[3] S. Cherem and R. Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. In *VMCAI '07*.

[4] R. Cherini, L. Rearte, and J. Blanco. A shape analysis for non-linear data structures. In *SAS '10*.

[5] S. Dasgupta. Precise shape analysis using field sensitivity. Technical report, Master's thesis, IIT Kanpur, 2011. `http://goo.gl/3U3WV`.

[6] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS '06*.

[7] R. Ghiya and L. J. Hendren. Is it a Tree, a DAG, or a Cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL '96*.

[8] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *POPL '98*.

[9] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive darta structures. In *CC '98*.

[10] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL '05*.

[11] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL '79*.

[12] M. Jump and K. S. McKinley. Dynamic shape analysis via degree metrics. In *ISMM '09*.

[13] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity: unified memory analysis: the base framework. In *LCPC'06*.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96*.

[15] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3), 2002.

[16] R. Shaham, E. Yahav, E. K. Kolodner, and S. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS '03*.