# Precise Shape Analysis using Field Sensitivity

Sandeep Dasgupta and Amey Karkare

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, Kanpur, India
{dsand,karkare}cse.iitk.ac.in
http://www.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.

**Keywords:** Compilers, formal methods, logics of programs, program analysis

## 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 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 [15, 18, 5, 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, 14] 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 conservative shape to 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 Section 2. A motivating example is used in Section 3 to explain the intuition behind our analysis . The analysis is formalized in Section 4 that describes the notations used and in Section 5 that gives the analysis rules. We conclude the presentation in Section **??** and give 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 resulted 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->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. [15–17] 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 runtime in the number of predicates, and therefore not suitable for large programs.

Distefano et al. [5] 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.).

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 inter-procedural 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. Each class of heap structure is then summarized. 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. [14]. 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 also demonstrated the practical applications of their analysis [6, 8, 9] and shown that it works well on practical programs. However, the main shortcoming of this approach is 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. So it has to to conservatively identify the shapes.

Marron et. al. [14] 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 [16, 17], the re-materialization is approximate and may result in loss of precision.

Our method is also data flow analysis framework, that uses matrices and boolean equations as data flow values. We use field sensitive connectivity 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

For each pointer variable, our analysis computes the shape attribute of the data structure pointed to by the variable. Following the existing literature [7, 15, 6, 14], we define the shape attribute p.shape for a pointer p as follows:

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

Where the heap is visualized as a directed graph, and cycle and DAG have there natural graph-theoretic meanings.

We use the code fragment in Figure 1(a) to motivate the need for a field sensitive shape analysis.

```
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 | D | | | I | | |
|---|---|---|---|---|---|---|
| | | p | q | | p | q |
| S1 | p | 1 | 0 | p | 1 | 0 |
| | q | 0 | 0 | q | 0 | 0 |
| | | p | q | | p | q |
| S2 | p | 1 | 0 | p | 1 | 0 |
| | q | 0 | 1 | q | 0 | 1 |
| | | p | q | | p | q |
| S3 | p | 1 | 1 | p | 1 | 1 |
| | q | 0 | 1 | q | 1 | 1 |
| | | p | q | | p | q |
| S4 | p | 1 | 1 | p | 1 | 1 |
| | q | 0 | 1 | q | 1 | 1 |
| | | p | q | | p | q |
| S5, S6, S7 | p | 1 | 1 | p | 1 | 1 |
| | q | 1 | 1 | q | 1 | 1 |

(b) Direction ($D$) and Interference ($I$) matrices as computed by [7]

**Fig. 1.** A motivating example

*Example 1*. Consider the code segment in the Figure 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 Figure 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 (Section 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 the paths from p to q. However, as there may be an unbounded number of paths between two pointer variables, we need to conservatively approximate the path information.

Our analysis approximates the paths between two variable by the first field that is dereferenced on the path. The analysis remembers this limited 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.

Figure 2 shows 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

| **After Stmt** | $D_F$ | | | $I_F$ | | |
|---|---|---|---|---|---|---|
| S1 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\phi$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\phi$ |
| | q | $\phi$ | $\phi$ | q | $\phi$ | $\phi$ |
| S2 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\phi$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\phi$ |
| | q | $\phi$ | $\{\varepsilon\}$ | q | $\phi$ | $\{(\varepsilon,\varepsilon)\}$ |
| S3 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\{f\}$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\{(f,\varepsilon)\}$ |
| | q | $\phi$ | $\{\varepsilon\}$ | q | $\{(\varepsilon,f)\}$ | $\{(\varepsilon,\varepsilon)\}$ |
| S4 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\{f,h\}$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\{(f,\varepsilon),(h,\varepsilon)\}$ |
| | q | $\phi$ | $\{\varepsilon\}$ | q | $\{(\varepsilon,f),(\varepsilon,h)\}$ | $\{(\varepsilon,\varepsilon)\}$ |
| S5 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\{f,h\}$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\{(f,\varepsilon),(h,\varepsilon),(\varepsilon,g)\}$ |
| | q | $\{g\}$ | $\{\varepsilon\}$ | 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 | $\phi$ | $\{\varepsilon\}$ | q | $\{(\varepsilon,f),(\varepsilon,h)\}$ | $\{(\varepsilon,\varepsilon)\}$ |
| S7 | | p | q | | p | q |
| | p | $\{\varepsilon\}$ | $\{f\}$ | p | $\{(\varepsilon,\varepsilon)\}$ | $\{(f,\varepsilon)\}$ |
| | q | $\phi$ | $\{\varepsilon\}$ | q | $\{(\varepsilon,f)\}$ | $\{(\varepsilon,\varepsilon)\}$ |

(a) Direction ($D_F$) and Interference ($I_F$) Matrices.

| **After Stmt** | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|---|---|---|---|---|---|---|---|
| **Boolean Vars** | | | | | | | |
| $f_{pq}$ | false | false | true | true | true | true | true |
| $h_{pq}$ | false | false | false | true | true | true | false |
| $g_{qp}$ | false | false | false | false | true | false | false |

(b) Values for boolean variables corresponding to relevant pointer fields.

**Fig. 2.** Field Sensitive information for the code in Figure 1(a)

the following boolean equations[1] :

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

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 equations 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 $\texttt{p}_{\texttt{Dag}}$ to become false.

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 equations for detecting cycle on p after S5 are:

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

Here, the equations captures the fact that cycle on p is 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 ($g_{qp} = \textbf{False}$). The latter occurs after S6 in Figure 1(a). □

In the rest of the paper, we formalize the intuitions presented above and describe our analysis in details.

## 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 the 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

[1] The equations shown here are simplified versions of the actual equations generated by the analysis. Simplification is done manually to avoid clutter in the presentation.

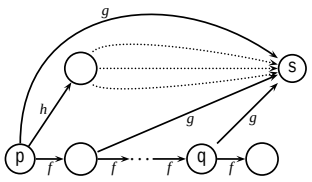
[2] The decision to use only first field is guided by the fact that in our language, a statement can use at most one field, i.e. p→f = ... or ...= p→f. In a language that allows use of more than one fields, we can use fixed $k$ length prefixes. However, this does not make any fundamental change to our analysis.

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 direct path and $I$ for 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, with at most one direct path starting at a given field $f$. We want to keep track of all such possible paths. The number of such paths may be unbounded. However, 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 $l$ on number of repetitions of a particular prefix. If the number goes beyond $l$, we treat the number of paths with that prefix as $\infty$. This approach is similar to the approach of $k$-limiting [11] and $sl$-limiting [13].

```
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.

**Fig. 3.** Paths in a heap graph

*Example 3.* Figure 3(a) shows a code fragment and Figure 3(b) shows a possible heap graph at line lineNum. 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 $l \geq 3$, this information can be represented by the set $\{g^D, f^{I\infty}, h^{I3}\}$. On the other hand, if $l < 3$, then the set would be $\{g^D, f^{I\infty}, h^{I\infty}\}$. □

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 $\mathtt{p}, \mathtt{q} \in \mathcal{H}, f \in \mathcal{F}$:*

$$
\begin{aligned}
\varepsilon &\in D_F[\mathtt{p},\mathtt{p}]\ . \\
f^D &\in D_F[\mathtt{p},\mathtt{q}] \quad \textit{if there is a direct path } f \textit{ from } \mathtt{p} \textit{ to } \mathtt{q}\ . \\
f^{Ik} &\in D_F[\mathtt{p},\mathtt{q}] \quad \textit{if there are } k \textit{ indirect paths starting with field } f \textit{ from } \mathtt{p} \textit{ to } \mathtt{q} \textit{ and } k \leq l\ . \\
f^{I\infty} &\in D_F[\mathtt{p},\mathtt{q}] \quad \textit{if there are } k \textit{ indirect paths starting with field } f \textit{ from } \mathtt{p} \textit{ to } \mathtt{q} \textit{ and } k > l\ .
\end{aligned}
$$

*where* $\varepsilon$ *denotes the empty path.*

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

$$\begin{aligned} &\varepsilon \sqsubseteq \varepsilon, f^D \sqsubseteq f^D, f^{I\infty} \sqsubseteq f^{I\infty} . \\ &f^{Ik} \sqsubseteq f^{I\infty} \quad k \in \mathbb{N} . \\ &f^{Ik} \sqsubseteq f^{Ij} \quad j,k \in \mathbb{N}, k \leq j . \end{aligned}$$

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

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

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

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

$$D_F[\mathtt{p}, \mathtt{s}] \times D_F[\mathtt{q}, \mathtt{s}] \sqsubseteq I_F[\mathtt{p}, \mathtt{q}] . \tag{1}$$

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 Equation 1 to compute $I_F$, computing both explicitly results in better approximations for $I_F$.

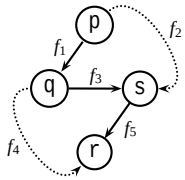
*Example 4.* Figure 4 shows a heap graph and the corresponding field sensitive matrices as computed by our analysis. □

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

The shape of $\mathtt{p}$, $\mathtt{p}$.shape, can be obtained by evaluating the equations for the attributes $\mathtt{p}_{\mathtt{Cycle}}$ and $\mathtt{p}_{\mathtt{Dag}}$, and using the following tabletable caption above table:

| $\mathtt{p}_{\mathtt{Cycle}}$ | $\mathtt{p}_{\mathtt{Dag}}$ | p.shape |
|---|---|---|
| **True** | Don't Care | Cycle |
| **False** | **True** | DAG |
| **False** | **False** | Tree |

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




(a) A heap graph

| $D_F$ | p | q | s | r |
|---|---|---|---|---|
| p | {ε} | $\{f_1^D\}$ | $\{f_1^{I1}, f_2^{I1}\}$ | $\{f_1^{I2}, f_2^{I1}\}$ |
| q | ∅ | {ε} | $\{f_3^D\}$ | $\{f_3^{I1}, f_4^{I1}\}$ |
| s | ∅ | ∅ | {ε} | $\{f_5^D\}$ |
| r | ∅ | ∅ | ∅ | {ε} |

(b) Direction Matrix

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

(c) Interference Matrix

**Fig. 4.** A heap graph and its field sensitive path matrices

## 5 Analysis Rules

For $\{\mathtt{p},\mathtt{q}\} \subseteq \mathcal{H}$, $f \in \mathcal{F}$, $k \in \mathbb{N}$ and $\mathtt{op} \in \{+,-\}$, we have the following eight basic statements that can access or modify the heap structures.

1. Allocations
   (a) `p = malloc();`
2. Pointer Assignments
   (a) `p = NULL;`
   (b) `p = q;`
   (c) `p = q` $\rightarrow$ `f;`
   (d) `p = &(q` $\rightarrow$ `f);`
   (e) `p = q op k;`
3. Structure Updates
   (a) `p` $\rightarrow$ `f = q;`
   (b) `p` $\rightarrow$ `f = NULL;`

The intend is to determine $D_F$ and $I_F$ matrices and the sets `B`, `TrueVars` at each program point. The problem is a forward data flow analysis problem with $\langle D_F, I_F, \mathtt{B}, \mathtt{TrueVars}\rangle$ as the data flow value at each program point with each basic block containing a single statement only. Let $\langle subDF^x, I_F^x, \mathtt{B}^x, \mathtt{TrueVars}^x\rangle$ and $\langle D_F^y, I_F^y, \mathtt{B}^y, \mathtt{TrueVars}^y\rangle$ are two

data flow values at program points $x$ and $y$, the confluence operator, Merge, is defined as follows:

$$\begin{aligned}
&Merge(\mathtt{TrueVars}^x, \mathtt{TrueVars}^y) \Rightarrow \\
&\qquad \mathtt{TrueVars}^x = \mathtt{TrueVars}^x \cup \mathtt{TrueVars}^y. \\
&Merge(\mathtt{B}^x, \mathtt{B}^y) \Rightarrow \\
&\qquad \forall \mathtt{p} \in \mathcal{H},\ \mathtt{B}^x.\mathtt{p}_{\mathtt{Cycle}} = \mathtt{B}^x.\mathtt{p}_{\mathtt{Cycle}} \cup \mathtt{B}^y.\mathtt{p}_{\mathtt{Cycle}} \\
&\qquad \forall \mathtt{p} \in \mathcal{H},\ \mathtt{B}^x.\mathtt{p}_{\mathtt{Dag}} = \mathtt{B}^x.\mathtt{p}_{\mathtt{Dag}} \cup \mathtt{B}^y.\mathtt{p}_{\mathtt{Dag}} \\
&Merge(D_F^x, D_F^y) \Rightarrow \\
&\qquad \forall \mathtt{p},\mathtt{q} \in \mathcal{H},\ D_F^x[\mathtt{p},\mathtt{q}] = D_F^x[\mathtt{p},\mathtt{q}] \quad \cup \quad D_F^y[\mathtt{p},\mathtt{q}] \\
&Merge(I_F^x, I_F^y) \Rightarrow \\
&\qquad \forall \mathtt{p},\mathtt{q} \in \mathcal{H},\ I_F^x[\mathtt{p},\mathtt{q}] = I_F^x[\mathtt{p},\mathtt{q}] \quad \cup \quad I_F^y[\mathtt{p},\mathtt{q}]
\end{aligned} \tag{2}$$

Now for each statement $st$, we compute both the gen set ($gen_{st}$) and kill set ($kill_{st}$) corresponding to each data flow value and then applying the following transfer function $f$ we compute the data flow values after the statement $st$.

$$out_{st} = \mathtt{f}(in_{st}) = (in_{st} - kill_{st}) \cup gen_{st}$$

where $in_{st}$ and $out_{st}$ corresponds the data flow values before and after the statement $st$ respectively.

### 5.1 Analysis of Basic Statements

For any $\mathtt{p} \in \mathcal{H}$, $p^{gen}_{\mathtt{Cycle}}$ and $p^{gen}_{\mathtt{Dag}}$ ($p^{kill}_{\mathtt{Cycle}}$ and $p^{kill}_{\mathtt{Dag}}$) denotes the gen (kill) sets corresponding to $\mathtt{B}.\mathtt{p}_{\mathtt{Cycle}}$ and $\mathtt{B}.\mathtt{p}_{\mathtt{Dag}}$ respectively. $\mathtt{gen}_{\mathtt{TrueVars}}$ ($\mathtt{kill}_{\mathtt{TrueVars}}$) denotes the gen (kill) set corresponding to `TrueVars`. Also for each basic statements we propose algorithm to determine the gen and kill information corresponding to $D_F$ and $I_F$. Also we consider $\langle D_F^{in}, I_F^{in}, \mathtt{B}^{in}, \mathtt{TrueVars}^{in} \rangle$ and $\langle D_F^{out}, I_F^{out}, \mathtt{B}^{out}, \mathtt{TrueVars}^{out} \rangle$ as the data flow values before and after a particular statement under consideration.

$$D_F^{out}[\mathtt{p},\mathtt{q}] = D_F^{in}[\mathtt{p},\mathtt{q}] \ominus D_F^{kill}[\mathtt{p},\mathtt{q}] \cup D_F^{gen}[\mathtt{p},\mathtt{q}] \tag{3}$$

$$\ldots \tag{4}$$

We now present our analysis of each kind of statement.

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

$$\begin{aligned}
p^{kill}_{\mathtt{Cycle}} &= p^{in}_{\mathtt{Cycle}} & p^{kill}_{\mathtt{Dag}} &= p^{in}_{\mathtt{Dag}} \\
p^{gen}_{\mathtt{Cycle}} &= \mathbf{False} & p^{gen}_{\mathtt{Dag}} &= \mathbf{False}
\end{aligned}$$

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

$$\begin{aligned} D_F^{kill}[\mathtt{p},\mathtt{s}] &= D_F^{in}[\mathtt{p},\mathtt{s}] & D_F^{gen}[\mathtt{p},\mathtt{s}] &= \emptyset \\ D_F^{kill}[\mathtt{s},\mathtt{p}] &= D_F^{in}[\mathtt{s},\mathtt{p}] & D_F^{gen}[\mathtt{s},\mathtt{p}] &= \emptyset \\ D_F^{kill}[\mathtt{p},\mathtt{p}] &= D_F^{in}[\mathtt{p},\mathtt{p}] & D_F^{gen}[\mathtt{p},\mathtt{p}] &= \{\varepsilon\} \\ I_F^{kill}[\mathtt{p},\mathtt{s}] &= I_F^{in}[\mathtt{p},\mathtt{s}] & I_F^{gen}[\mathtt{p},\mathtt{s}] &= \emptyset \\ I_F^{kill}[\mathtt{s},\mathtt{p}] &= I_F^{in}[\mathtt{s},\mathtt{p}] & I_F^{gen}[\mathtt{s},\mathtt{p}] &= \emptyset \\ I_F^{kill}[\mathtt{p},\mathtt{p}] &= I_F^{in}[\mathtt{p},\mathtt{p}] & I_F^{gen}[\mathtt{p},\mathtt{p}] &= \{\varepsilon,\varepsilon\} \end{aligned}$$

2. `p = NULL`: This statement does not cause any generate new relations, but kills the relations of `p`.

$$\begin{aligned} p_{\mathtt{Cycle}}^{kill} &= p_{\mathtt{Cycle}}^{in} & p_{\mathtt{Dag}}^{kill} &= p_{\mathtt{Dag}}^{in} \\ p_{\mathtt{Cycle}}^{gen} &= \mathbf{False} & p_{\mathtt{Dag}}^{gen} &= \mathbf{False} \end{aligned}$$

$\forall \mathtt{s} \in \mathcal{H}$ :

$$\begin{aligned} D_F^{kill}[\mathtt{p},\mathtt{s}] &= D_F^{in}[\mathtt{p},\mathtt{s}] & D_F^{gen}[\mathtt{p},\mathtt{s}] &= \emptyset \\ D_F^{kill}[\mathtt{s},\mathtt{p}] &= D_F^{in}[\mathtt{s},\mathtt{p}] & D_F^{gen}[\mathtt{s},\mathtt{p}] &= \emptyset \\ I_F^{kill}[\mathtt{p},\mathtt{s}] &= I_F^{in}[\mathtt{p},\mathtt{s}] & I_F^{gen}[\mathtt{p},\mathtt{s}] &= \emptyset \\ I_F^{kill}[\mathtt{s},\mathtt{p}] &= I_F^{in}[\mathtt{s},\mathtt{p}] & I_F^{gen}[\mathtt{s},\mathtt{p}] &= \emptyset \end{aligned}$$

3. `p = q, p = &(q→f), p = q op k:` . 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`.
$\forall \mathtt{s} \in \mathcal{H}, \mathtt{s} \neq \mathtt{p}, \forall f \in \mathcal{F}$

$$\begin{aligned} p_{\mathtt{Cycle}}^{kill} &= p_{\mathtt{Cycle}}^{in} & p_{\mathtt{Dag}}^{kill} &= p_{\mathtt{Dag}}^{in} \\ p_{\mathtt{Cycle}}^{gen} &= q_{\mathtt{Cycle}}^{in}[q/p] & p_{\mathtt{Dag}}^{gen} &= q_{\mathtt{Dag}}^{in}[q/p] \\ f\mathtt{ps} &= f\mathtt{qs} & f\mathtt{sp} &= f\mathtt{sq} \end{aligned}$$

$$\begin{aligned} D_F^{kill}[\mathtt{p},\mathtt{s}] &= D_F^{in}[\mathtt{p},\mathtt{s}] & D_F^{gen}[\mathtt{p},\mathtt{s}] &= D_F^{in}[\mathtt{q},\mathtt{s}] \\ D_F^{kill}[\mathtt{s},\mathtt{p}] &= D_F^{in}[\mathtt{s},\mathtt{p}] & D_F^{gen}[\mathtt{s},\mathtt{p}] &= D_F^{in}[\mathtt{s},\mathtt{q}] \\ D_F^{kill}[\mathtt{p},\mathtt{p}] &= D_F^{in}[\mathtt{p},\mathtt{p}] & D_F^{gen}[\mathtt{p},\mathtt{p}] &= D_F^{in}[\mathtt{q},\mathtt{q}] \\ I_F^{kill}[\mathtt{p},\mathtt{s}] &= I_F^{in}[\mathtt{p},\mathtt{s}] & I_F^{gen}[\mathtt{p},\mathtt{s}] &= I_F^{in}[\mathtt{q},\mathtt{s}] \\ I_F^{kill}[\mathtt{s},\mathtt{p}] &= I_F^{in}[\mathtt{s},\mathtt{p}] & I_F^{gen}[\mathtt{s},\mathtt{p}] &= I_F^{in}[\mathtt{s},\mathtt{q}] \\ I_F^{kill}[\mathtt{p},\mathtt{p}] &= I_F^{in}[\mathtt{p},\mathtt{p}] & I_F^{gen}[\mathtt{p},\mathtt{p}] &= I_F^{in}[\mathtt{q},\mathtt{q}] \end{aligned}$$

where $X[\mathtt{q}/\mathtt{p}]$ creates a copy of $X$ with all occurrences of `q` replaced by `p`.

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

$$\begin{aligned} p^{kill}_{Cycle} &= \textbf{False} & p^{kill}_{Dag} &= \textbf{False} \\ p^{gen}_{Cycle} &= \textbf{False} & p^{gen}_{Dag} &= \textbf{False} \\ fpq &= \textbf{False} \end{aligned}$$

$$\begin{aligned} D^{kill}_F[p,q] &= D^{in}_F[p,q] \triangleright f & D^{gen}_F[p,q] &= \emptyset \\ D^{kill}_F[s,q] &= |D^{in}_F[p,q] \triangleright f| \star D^{in}_F[s,p] & D^{gen}_F[s,q] &= \emptyset \\ I^{kill}_F[r,q] &= D^{in}_F[r,p] \times I^{in}_F[p,q].\text{fst} \triangleright f & I^{gen}_F[r,q] &= \emptyset \end{aligned}$$

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:

$$\begin{aligned} p^{gen}_{Cycle} &= (fpq \wedge q^{in}_{Cycle}) \vee (fqp \wedge (|D^{in}_F[q,p]| \geq 1)) \\ q^{gen}_{Cycle} &= fqp \wedge (|D^{in}_F[q,p]| \geq 1) \\ p^{gen}_{Dag} &= fqp \wedge (|I^{in}_F[q,p]| > 1) \\ fpq &= \textbf{True} \end{aligned}$$

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

$$\begin{aligned} s^{gen}_{Cycle} = &((|D^{in}_F[s,p]| \geq 1) \wedge fpq \wedge q^{in}_{Cycle}) \\ &\vee ((|D^{in}_F[s,p]| \geq 1) \wedge fqp \wedge (|D^{in}_F[q,p]| \geq 1)) \\ &\vee ((|D^{in}_F[s,q]| \geq 1) \wedge fqp \wedge (|D^{in}_F[q,p]| \geq 1)) \\ s^{gen}_{Dag} = &(|D^{in}_F[s,p]| \geq 1) \wedge fqp \wedge (|I^{in}_F[q,p]| > 1) \end{aligned}$$

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

$$\begin{aligned} D^{gen}_F[s,r] &= |D^{in}_F[q,r]| \star D^{in}_F[s,p] \quad \text{if } s \neq p \\ D^{gen}_F[p,r] &= |D^{in}_F[q,r]| \star (D^{in}_F[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}) \\ &\quad \text{if } r \neq q \\ D^{gen}_F[p,q] &= |D^{in}_F[q,q]| \star (D^{in}_F[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}) \\ &\quad \bigcup \{f^D\} \end{aligned}$$

$$
\begin{aligned}
I_F^{gen}[\mathtt{s},\mathtt{r}] &= D_F^{in}[\mathtt{s},\mathtt{p}] \times I_F^{in}[\mathtt{q},\mathtt{r}].\mathrm{snd} \\
&\qquad \text{if } \mathtt{s} \neq \mathtt{p} \text{ and } D_F^{in}[\mathtt{r},\mathtt{p}] = \emptyset \\
I_F^{gen}[\mathtt{p},\mathtt{r}] &= |D_F^{in}[\mathtt{q},\mathtt{r}]| \star (D_F^{in}[\mathtt{p},\mathtt{p}] \ominus \{\varepsilon\} \cup \{f^{I1}\}) \\
&\qquad \text{if } \mathtt{r} \neq \mathtt{q} \\
I_F^{gen}[\mathtt{p},\mathtt{q}] &= |D_F^{in}[\mathtt{q},\mathtt{q}]| \star (D_F^{in}[\mathtt{p},\mathtt{p}] \ominus \{\varepsilon\} \cup \{f^{I1}\}) \\
&\qquad \bigcup \{f^D\}
\end{aligned}
$$

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 $\mathtt{q} \rightarrow f$ before the assignment. This information is available because $I_F^{in}[\mathtt{q},\mathtt{r}]$ will have an entry of the form $(f^D,\alpha)$ for some $\alpha$. Therefore, all attributes, except $D_F^{gen}[\mathtt{r},\mathtt{p}]$ are computed approximately as explained next.
   As `p` could potentially point to a cycle(DAG) reachable from `q`, we set:

$$p^{gen}_{\mathtt{Cycle}} = q^{in}_{\mathtt{Cycle}} \qquad p^{gen}_{\mathtt{Dag}} = q^{in}_{\mathtt{Dag}}$$

   We record the fact that `q` reaches `p` through the path $f$. Further, we mark any object reachable from `q` as directly accessible from `p` through any possible field.

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

   The equations to compute the generated relations for $D_F$ can be divided into three components. We explain each of the component, and give corresponding 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 $D_F[\mathtt{p},\mathtt{s}]$. The set $\mathcal{U}$ is defined as:

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

   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:

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

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 ($\infty$) as an approximation:

$$D_2[\mathsf{s},\mathsf{p}] = \infty \star D_F^{in}[\mathsf{s},\mathsf{q}] \quad \forall \mathsf{s} \in \mathcal{H}, \mathsf{s} \neq \mathsf{q}$$
$$D_2[\mathsf{q},\mathsf{p}] = \{f^D\} \cup (D_F^{in}[\mathsf{q},\mathsf{q}] \ominus \{\varepsilon\})$$

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. These paths can be determined as:

$$D_3[\mathsf{s},\mathsf{p}] = \{\alpha \mid (f^D,\alpha) \in I_F^{in}[\mathsf{q},\mathsf{s}]\}$$

Finally, we conservatively compute the generated $I_F$ relations. Thus,

$$D_F^{gen}[\mathsf{r},\mathsf{s}] = D_1[\mathsf{r},\mathsf{s}] \cup D_2[\mathsf{r},\mathsf{s}] \cup D_3[\mathsf{r},\mathsf{s}] \quad \forall \mathsf{r},\mathsf{s} \in \mathcal{H}$$
$$I_F^{gen}[\mathsf{r},\mathsf{p}] = I_F^{gen}[\mathsf{p},\mathsf{r}] \quad = \quad \mathcal{U} \times \mathcal{U} \quad \forall \mathsf{r} \in \mathcal{H}$$

## References


1. Berdine, J., Calcagno, C., Cook, B., Distefano, D., Ohearn, P.W., Yang, H., Mary, Q.: Shape analysis for composite data structures. In: In CAV. pp. 178–192. Springer (2007)
2. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. pp. 296–310. PLDI '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/93542.93585
3. Cherem, S., Rugina, R.: Maintaining doubly-linked list invariants in shape analysis with local reasoning. In: Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation. pp. 234–250. VMCAI'07, Springer-Verlag, Berlin, Heidelberg (2007), http://portal.acm.org/citation.cfm?id=1763048.1763073
4. Cherini, R., Rearte, L., Blanco, J.: A shape analysis for non-linear data structures. In: Proceedings of the 17th international conference on Static analysis. pp. 201–217. SAS'10, Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1882094.1882107
5. Distefano, D., OHearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 3920, pp. 287–302. Springer Berlin / Heidelberg (2006)
6. Ghiya, R.: Practical techniques for interprocedural heap analysis. Tech. rep., Master's thesis, McGill U (1996)
7. Ghiya, R., Hendren, L.J.: Is it a Tree, a DAG, or a Cyclic graph? a shape analysis for heap-directed pointers in C. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 1–15 (January 1996)
8. Ghiya, R., Hendren, L.J.: Putting pointer analysis to work. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 121–133. POPL '98, ACM, New York, NY, USA (1998), http://doi.acm.org/10.1145/268946.268957

9. Ghiya, R., Hendren, L.J., Zhu, Y.: Detecting parallelism in c programs with recursive darta structures. In: 7th International Conference on Compiler Construction. pp. 159–173 (1998)
10. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 310–323. POPL '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1040305.1040331
11. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of lisp-like structures. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 244–256. POPL '79, ACM, New York, NY, USA (1979), http://doi.acm.org/10.1145/567752.567776
12. Jump, M., McKinley, K.S.: Dynamic shape analysis via degree metrics. In: Proceedings of the 2009 international symposium on Memory management. pp. 119–128. ISMM '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1542431.1542449
13. Larus, J.R., Hilfinger, P.N.: Detecting conflicts between structure accesses. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. pp. 24–31. PLDI '88, ACM, New York, NY, USA (1988), http://doi.acm.org/10.1145/53990.53993
14. Marron, M., Kapur, D., Stefanovic, D., Hermenegildo, M.: A static heap analysis for shape and connectivity: unified memory analysis: the base framework. In: Proceedings of the 19th international conference on Languages and compilers for parallel computing. pp. 345–363. LCPC'06, Springer-Verlag, Berlin, Heidelberg (2007), http://portal.acm.org/citation.cfm?id=1757112.1757146
15. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 16–31. POPL '96, ACM, New York, NY, USA (1996), http://doi.acm.org/10.1145/237721.237725
16. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 105–118. POPL '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/292540.292552
17. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
18. Shaham, R., Yahav, E., Kolodner, E.K., Sagiv, S.: Establishing local temporal heap safety properties with applications to compile-time memory management. In: SAS '03: Proceedings of the 10th International Symposium on Static Analysis. pp. 483–503. Springer-Verlag, London, UK (2003)