Innovations Syst Softw Eng (2013) 9:79–93
DOI 10.1007/s11334-013-0198-7

SI: SAC-SVT'12

Precise shape analysis using field sensitivity

Sandeep Dasgupta · Amey Karkare · Vinay Kr Reddy

Received: 17 July 2012 / Accepted: 5 March 2013 / Published online: 3 April 2013

Abstract We present a static shape analysis technique to infer the shapes of the heap structures created by a program at run time. Our technique is field sensitive in that it uses field information to compute the shapes. The shapes of the heap structures are computed using two components: (a) Boolean functions that capture the shape transitions due to the update of a field in a structure, and (b) through path matrices that store approximate path information between two pointer variables. We classify the shapes as one of *Tree, Directed Acyclic Graph (DAG)* and *Cycle*. The novelty of our approach lies in the way we use field information to remember the fields that cause a heap structure to have a particular shape (Tree, DAG or Cycle). This allows us to easily identify the field updates that cause shape transitions from Cycle to DAG, from Cycle to Tree and from DAG to Tree. This makes our analysis more precise as compared to earlier shape analyses that ignore the fields participating in the formation of a shape. We implemented our analysis in GCC as a dynamic plug-in as an interprocedural data-flow analysis and evaluated it on some standard benchmarks against a field-insensitive shape analysis technique as a baseline approach. We are able to achieve significant precision as compared to the baseline analysis (at the cost of increase in analysis time). In particular, we are able to infer more precise shapes for 4 out 7 *Olden* benchmarks, and never detect more cycles than the baseline analysis. We further suggest enhancements to improve the precision of our analysis under some constraints and to improve the analysis time at the cost of precision.

Keywords Compilers · Dataflow analysis · Static analysis · Shape analysis · Heap analysis

S. Dasgupta
Intel Technology India Pvt. Ltd., Banglore, India
e-mail: sandigame.123@gmail.com

A. Karkare (✉) · V. K. Reddy
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, Kanpur, India
e-mail: karkare@cse.iitk.ac.in

V. K. Reddy
e-mail: vinayp@cse.iitk.ac.in

1 Introduction

Hardware and software revolutions in past few years have brought in two significant challenges: (a) High performance computing is now accessible to end users (scientists and engineers) in the forms of multi-core and GPU processors at a very low cost. However, these end users, who are domain experts, find it difficult to acquire expertise to take advantage of the compute power available at their disposal. As a result, the compute power remains significantly underutilized. The tasks of helping programmers "think parallel" and providing them with working parallel environment are considered the top challenges in parallel computing research [21,28]. (b) The cost of failure of software is increasing with the growing complexity of software. The cost of software failure was estimated to be USD 6.18 Trillion in the year 2009 [26]. Therefore, it is becoming more and more important to verify the correctness of a software program before its deployment. The complexity of hardware and software systems requires that the solutions to the above challenges have to be automated. We need parallelizing compilers that can automatically convert sequential programs to parallel one, and verifying compilers that can prove the correctness of programs.

Programs in all modern programming languages use heap intensively. Any non-trivial analysis of these programs requires precise reasoning about the heap structures. The reasoning is complex because the heap structures are not static but are manipulated dynamically during the execution of the program. *Shape analysis* is the term for the class of static analysis techniques that are used to infer useful properties about heap structures and the programs manipulating the heap. The shape information of the heap data structures can be used by 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, thus making them suitable either for verification or for optimization, but not both. Precise shape analysis algorithms [10,14,24,27] are not practical for optimizing compilers as they scale poorly to the large heap manipulating programs. To achieve scalability, the optimizing compilers use shape analysis algorithms [6,11,20] that trade precision for speed by ignoring certain properties of heap structures (for example, the calling context or the field connectivity between the pointers).

In this paper, we present a field-sensitive, context-sensitive shape analysis technique that uses field-based path information to infer the shapes of 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). 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.

Our analysis captures the field sensitivity information using two components: (a) Field-based Boolean variables to remember the direct connections between two pointer variables, and (b) Path matrices that store the approximate paths between pointer variables. The shape of a pointer variable at a given program point is inferred from these two components.

This paper makes the following contributions:

1. We present a novel field-based shape analysis technique that uses limited path information to infer precisely the shapes of heap structures. We propose the analysis as an instance of a forward data flow analysis framework, and describe in details the components of the framework.
2. We describe a flow-sensitive, context-sensitive implementation of our analysis as a plug-in for GCC version 4.5.0 [1]. We present experimental evaluation and comparison of our analysis with an existing field-insensitive approach [11] on several benchmark programs.
3. We propose some enhancements to our analysis: (a) A *Field-subset*-based analysis to improve precision when auxiliary fields[1] are present in data structures, and (b) shape-based context-sensitive analysis to improve memory footprint and speed when the number of contexts is huge.

[1] Fields that are used only for diagnostic or debugging purpose, or unused by a significant part of the program.

The paper is organized as follows: We demonstrate the working of our analysis using a motivating example in Sect. 2, and explain intuitively the key concepts. Section 3 presents the notations and definitions used by our analysis and Sect. 4 describes the details of the analysis. Section 5 describes the baseline approach (field-insensitive shape analysis [11]), the benchmark programs and our experimental evaluation. Section 6 describes some enhancements over our current scheme. We discuss some of the prior works on shape analysis in Sect. 7. Section 8 concludes the paper and gives directions for future work.

We start out with motivating examples that also illustrate the workings of our analysis.

2 A motivating example

Following the literature [11,20,24], 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 their 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 to motivate the need for a field-sensitive shape analysis.

Example 1 The program in Fig. 1a creates a binary tree rooted at `root` (not shown) and uses the function `mirror` to create its mirror image in situ. The program then calls `treeAdd` on the mirrored tree to perform the additions of its left and right subtrees recursively.

If it can be inferred that the shape of the argument to the function `treeAdd` is indeed `Tree`, then a parallelizing compiler can schedule the two recursive calls to `treeAdd` on lines *S24* and *S26* in parallel. This is possible because these two calls do not access any common region of heap, and hence they can proceed independently.

The inference of the shape of the argument of the `treeAdd` depends on the execution of `mirror`. Even when called with an argument that has a shape of `Tree`, the function `mirror` temporarily changes the shape to a `Dag` (due to assignments on line *S12* and *S15*, both `t`→`left` and `t`→`right` point to the same node `tr`). The shape is reverted back to `Tree` at line *S16*.

Field-insensitive shape analysis algorithms use conservative kill information and hence they are, in general, unable to

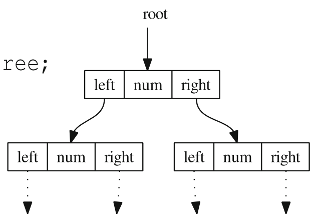
```
        int main() {
S1:       tree root;
S2:       ...
S3:       mirror(root);
S4:       treeAdd(root);
S5:       ...
        }

        void mirror(tree t) {
S11:      tl = t->left;
S12:      tr = t->right;
S13:      mirror(tl);
S14:      mirror(tr);
S15:      t->left = tr;
S16:      t->right = tl;
        }

        int treeAdd(tree t) {
S21:      if (t == NULL)
S22:        return 0;
S23:      tl = t→left;
S24:      tl_num = treeAdd(tl);
S25:      tr = t→right;
S26:      tr_num = treeAdd(tr);
S27:      t→num = tl_num + tr_num;
S28:      return t→num;
        }
```

(a)

```
typedef struct treenode tree;
struct treenode {
  int num;
  tree *left;
  tree *right;
};
```



(b)

Fig. 1 A motivating example. The program **a** manipulates the heap structures created by using the basic data structure as shown in **b**

infer the shape transition from `Cycle` to `Dag` or from `Dag` to `Tree`. For example, the algorithm by Ghiya et al. [11] can correctly report the shape transition from `Tree` to `Dag` (at *S15*) but fails to infer the shape transition from `Dag` to `Tree` (at *S16*). Our analysis, on the other hand, keeps track of the fields involved in creation of the DAG at *S15* (`t→left` and `t→right`), and therefore it is able to restore the shape to Tree when `t→right` is updated.

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. 4).

Example 2 The statement at *S15* creates a new DAG structure reachable from `t`, because there are two paths (`t→left` and `t→right`) reaching `tr`. A field-sensitive shape analysis algorithm must remember all paths from `t` to `tr`.

Stmt	Path matrix P_F after the stmt			
		t	tl	tr
S11	t	$\emptyset$	$\{\texttt{left}^{D}\}$	$\emptyset$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S12	t	$\emptyset$	$\{\texttt{left}^{D}\}$	$\{\texttt{right}^{D}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S15	t	$\emptyset$	$\emptyset$	$\{\texttt{right}^{D},\texttt{left}^{D}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$
		t	tl	tr
S16	t	$\emptyset$	$\{\texttt{right}^{D}\}$	$\{\texttt{left}^{D}\}$
	tl	$\emptyset$	$\emptyset$	$\emptyset$
	tr	$\emptyset$	$\emptyset$	$\emptyset$

Fig. 2 Paths computed by our analysis for the program in Fig. 1a as path matrix P_F. The entry $P_F[x, y]$ lists the paths between pointer variables x and y

Our analysis approximates a path between two pointer 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 [16] to approximate the number of paths starting at a given field.

Our analysis remembers the path information using the following: (a) P_F: Path matrix that stores the first fields of the paths between two pointers and (b) Boolean variables that remember the fields directly connecting two pointer variables. Figure 2 shows the values computed for P_F and Boolean variables by our analysis for the example program in Fig. 1a. In this case, the fact that the shape of the variable `t` becomes `Dag` after *S15* is captured by the following Boolean functions:[2]

$$\texttt{t}_{\texttt{Dag}} = (\texttt{left}_{\texttt{t tr}} \wedge (|P_F[\texttt{t}, \texttt{tr}]| > 1)),$$
$$\texttt{left}_{\texttt{t tr}} = \textbf{True}.$$

where $\texttt{left}_{\texttt{t tr}}$ is a Boolean variable that is **True** because the `left` field of `t` points to `tr`, P_F is field-sensitive Path matrix, $|P_F[\texttt{t}, \texttt{tr}]|$ is the count of number of paths between `t` and `tr`.

The functions simply say that variable `t` reaches a DAG because there are more than one paths ($|P_F[\texttt{t}, \texttt{tr}]| > 1$) from `t` to `tr`. It also keeps track of the path $\texttt{left}_{\texttt{t tr}}$ that is

[2] The functions and values shown in this example and in Fig. 2 are simplified to avoid references to concepts not defined yet.

involved in the creation of the DAG. Later, at statement *S16*, the path due `t→ right` between `t` and `tr` is broken, causing $|P_F[\texttt{t}, \texttt{tr}]| = 1$. This causes $\texttt{t}_{\texttt{Dag}}$ 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 values from P_F and the Boolean variables at that statement.

We now formalize the intuitions presented in the example above, starting with the concepts necessary to describe our field-sensitive shape analysis technique.

3 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.

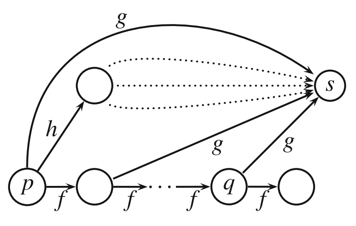
Let $\mathcal{H}$ denote the set of all heap-directed pointers at a particular program point and $\mathcal{F}$ denotes the set of all pointer fields at that program point. Given two heap-directed pointers $p, q \in \mathcal{H}$, a path from p to q is the sequence of pointer fields that need to be traversed 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.[3] 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. There can be at most one such direct path f^D. However, the number of indirect paths starting at a field f, f^I, may be unbounded. We include the count for the indirect paths between two pointer variables in the path set. To bound the size of the set, we put a limit k on the number of repetitions of a particular field. If the number goes beyond k, we treat the number of paths with that field as ∞. The following example illustrates these concepts.

[3] The decision to use only first field is guided by the fact that in many intermediate languages (for example, GIMPLE for GCC), a statement is allowed to use at most one field, i.e. `p→f` = … or …= `p→f`. Therefore, a long path is broken into several small paths. While it is possible to use prefixes of any fixed length by reconstructing the path, the process is complex and does not add any fundamental value to our analysis.

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

(a)



(b)

Fig. 3 Paths in a heap graph. *Graph* in **b** is a possible heap graph for code in **a**. *Solid edges* are the direct paths, *dotted edges* are the indirect paths

Example 3 Figure 3a shows a code fragment and Fig. 3b 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 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 field f. We now define the field-sensitive path matrix used by our analysis.

Definition 1 Field-sensitive path matrix P_F is a matrix that stores information about paths between two pointer variables in the following form: Given $p, q \in \mathcal{H}$, $f \in \mathcal{F}$:

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

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

The partial order is extended to set of paths S_{P_1}, S_{P_2} as:[4]

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

[4] 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.

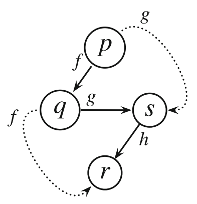
Two pointers $p, q \in \mathcal{H}$ are said to interfere at a program point if there exists $s \in \mathcal{H}$ such that both p and q have paths reaching s at that point. Note that s could be p (or q) itself, in which case the path from p (from q) is ϵ. Thus, the interference relation between p and q, $I_F[p, q]$ can be defined in terms of path matrix (P_F) as:

$$I_F[p, q] \Rightarrow \exists s. P_F[p, s] \wedge P_F[q, s] \tag{1}$$

As we will see later in the data-flow equations (Sect. 4), we are only interested in the *maximum count* of pair of paths that are interfering for two pointers at a given heap node. We use $|I_F[p, q]|$ to denote this count for interfering paths for nodes p and q. Our analysis computes over-approximations for the P_F matrix at each program point, and uses Eq. (1) to compute an over-approximation for $|I_F|$. This can result in a conservative shape (Cycle or DAG instead of a Tree, Cycle instead of a DAG) for a pointer, which is a safe inference.

Example 4 Figure 4 shows a heap graph and the corresponding path matrix 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 P_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. It is important to note that we are only interested in those Boolean functions that evaluate to true for attributes $p_{\texttt{Cycle}}$ and $p_{\texttt{Dag}}$. This is because we want to capture the transitions of shape from `Cycle` to `Dag`, from `Cycle` to `Tree`, and from `Dag` to `Tree`. If the function for $p_{\texttt{Cycle}}$ ($p_{\texttt{Dag}}$) is already false, then such transitions cannot take place. Therefore, we only store Boolean functions if they can potentially evaluate to true value. Further, for these functions, we store the functions themselves as Binary Decision Diagrams (BDDs) [4], and not the evaluated values. The evaluation of functions takes place only when the shape of the corresponding pointer is required by some client of the analysis or by the analysis itself. We call this *lazy* evaluation of Boolean functions.



(a)

P_F	p	q	s	r
p	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{I1}, g^{I1}\}$	$\{f^{I2}, g^{I1}\}$
q	$\emptyset$	$\{\varepsilon\}$	$\{g^D\}$	$\{g^{I1}, f^{I1}\}$
s	$\emptyset$	$\emptyset$	$\{\varepsilon\}$	$\{h^D\}$
r	$\emptyset$	$\emptyset$	$\emptyset$	$\{\varepsilon\}$

(b)

Fig. 4 A heap graph **a** and its field-sensitive path matrix **b**

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

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 $k \in \mathcal{N}$ denote 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:

$$|\epsilon| = 1, \quad \left|f^D\right| = 1, \quad \left|f^{I\infty}\right| = \infty$$
$$\left|f^{Ij}\right| = j \quad \text{for } j \in \mathcal{N}$$
$$|S| = \sum_{\alpha \in S} |\alpha|$$

– 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 can be extended to set of paths in a natural way. For example, for general sets of paths, S_1 and S_2, the definition of removal can be extended as:

$$S_1 \ominus S_2 = \bigcap_{\beta \in S_2} \bigcup_{\alpha \in S_1} (\{\alpha\} \ominus \{\beta\})$$

– Multiplication by a scalar($\star$): Let $i, j \in \mathcal{N}$, $i \le k$, $j \le k$. 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:

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

4 Our analysis

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

Table 2 Path operations

$i, j \in \mathcal{N}, \quad m = \mathtt{max}(i - j, 0), \quad n = \mathtt{min}(i, j),$

$\tau = \begin{cases} i + j & \text{if } i + j \le k \\ \infty & \text{Otherwise} \end{cases}$

$\eta = \begin{cases} i * j & \text{if } i * j \le k \\ \infty & \text{Otherwise} \end{cases}$

	$\{\epsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	g^*
(a) Path removal					
$\ominus \quad \{\beta\}$					
$\{\alpha\}$					
$\{\epsilon\}$	$\emptyset$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
$\{f^D\}$	$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\{f^D\}$	$\{f^D\}$
$\{f^{Ii}\}$	$\{f^{Ii}\}$	$\emptyset$	$\{f^{Im}\}$	$\emptyset$	$\{f^{Ii}\}$
$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\emptyset$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$
(b) Intersection					
$\cap \quad \{\beta\}$					
$\{\alpha\}$					
$\{\epsilon\}$	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					
$\cup \quad \{\beta\}$					
$\{\alpha\}$					
$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon, f^D\}$	$\{\epsilon, f^{Ij}\}$	$\{\epsilon, f^{I\infty}\}$	$\{\epsilon, g^*\}$
$\{f^D\}$	$\{f^D, \epsilon\}$	$\{f^D\}$	$\{f^D, f^{Ij}\}$	$\{f^D, f^{I\infty}\}$	$\{f^D, g^*\}$
$\{f^{Ii}\}$	$\{f^{Ii}, \epsilon\}$	$\{f^{Ii}, f^D\}$	$\{f^{I\tau}\}$	$\{f^{I\infty}\}$	$\{f^{Ii}, g^*\}$
$\{f^{I\infty}\}$	$\{f^{I\infty}, \epsilon\}$	$\{f^{I\infty}, f^D\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}, g^*\}$
(d) Multiplication by a Scalar					
$\star \quad \alpha$	ϵ	f^D	f^{Ij}	$f^{I\infty}$	
i					
i	ϵ	f^{Ii}	$f^{I\eta}$	$f^{I\infty}$	
∞	ϵ	$f^{I\infty}$	$f^{I\infty}$	$f^{I\infty}$	

1. Allocations

   (a) `p = malloc();`

2. Pointer Assignments

   (a) `p = NULL;`
   (b) `p = q;`
   (c) `p = q→ f;`
   (d) `p = &(q→ f);`
   (e) `p = q op n;`

3. Structure Updates

   (a) `p→ f = q;`
   (b) `p→ f = NULL;`

Our intend is to determine, at each program point, the path matrix P_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 Boolean variables and the path matrix. 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,

$$\text{merge}\left(f^x_{pq}, f^y_{pq}\right) = f^x_{pq} \vee f^y_{pq}, \quad f \in \mathcal{F}, \quad p, q \in \mathcal{H}$$
$$\text{merge}\left(p^x_{\mathtt{Cycle}}, p^y_{\mathtt{Cycle}}\right) = p^x_{\mathtt{Cycle}} \vee p^y_{\mathtt{Cycle}}, \quad p \in \mathcal{H}$$
$$\text{merge}\left(p^x_{\mathtt{Dag}}, p^y_{\mathtt{Dag}}\right) = p^x_{\mathtt{Dag}} \vee p^y_{\mathtt{Dag}}, \quad p \in \mathcal{H}$$
$$\text{merge}\left(P^x_F, P^y_F\right) = P^{xy}_F \quad \text{where } P^{xy}_F[p, q] = P^x_F[p, q] \cup P^y_F[p, q], \quad \forall p, q \in \mathcal{H}$$

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

$$P_F^{\text{out}}[p,q] = \left(P_F^{\text{in}}[p,q] \ominus P_F^{\text{kill}}[p,q]\right) \cup P_F^{\text{gen}}[p,q]$$
$$p_{\texttt{Cycle}}^{\text{out}} = \left(p_{\texttt{Cycle}}^{\text{in}} \wedge \neg p_{\texttt{Cycle}}^{\text{kill}}\right) \vee p_{\texttt{Cycle}}^{\text{gen}}$$
$$p_{\texttt{Dag}}^{\text{out}} = \left(p_{\texttt{Dag}}^{\text{in}} \wedge \neg p_{\texttt{Dag}}^{\text{kill}}\right) \vee p_{\texttt{Dag}}^{\text{gen}}$$

Update of field connectivity information and the computation of the gen and kill components of various data flow values depend on the type of statement. This is described in details next.

4.1 Analysis of basic statements

We now present our analysis for each kind of statement that can modify the heap structures.

4.1.1 `p = NULL`

This statement only kills the existing values of p. The heap node pointed-to by p is no longer reachable by p. However, it is possible that the node is part of a DAG or a Cycle reachable from some other node q. To keep the analysis safe, we need to first evaluate all terms that involve $p(f_{pq}^*,\ P_F[q,p]$, etc.) and use the resulting value in the equations using them. Then, we kill the values of $p$as given by the equations:

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

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

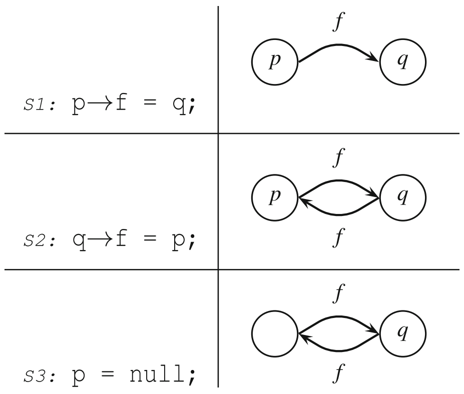
$$\begin{array}{ll} f_{ps} = \textbf{False} & f_{sp} = \textbf{False} \\ P_F^{\text{kill}}[p,s] = P_F^{\text{in}}[p,s] & P_F^{\text{gen}}[p,s] = \emptyset \\ P_F^{\text{kill}}[s,p] = P_F^{\text{in}}[s,p] & P_F^{\text{gen}}[s,p] = \emptyset \end{array}$$

Example 5 In the code segment Fig. 5a, after statement *S2* there is a `Cycle` on p and q. The cycle on $q$must remain after `S3`. Figure 5b, presents the relevant data-flow values after *S2*. Before *S3*, we evaluate terms containing p, i.e. f_{qp} and $|P_F[p,q] \geq 1|$ both to true, and use these values in the equation $q_{\texttt{Cycle}}^{\text{in}}$ so as to get $q_{\texttt{Cycle}}^{\text{in}}$ as true. Then we kill all the equations of p. After *S3*, we still have $q_{\texttt{Cycle}}^{\text{out}}$ as true, thus inferring the shape of q as `Cycle`. This would not be the case if we do not eagerly evaluate the terms containing p at *S3*.

4.1.2 `p = malloc()`

After this statement all the existing data-flow values of p get killed and $p$starts pointing to a newly allocated object. The kill effect is exactly same as that of `p = NULL`. After the statement, p can only have an empty path to itself.

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



(a)

$$P_F[p,q] = \{f^D\} \qquad P_F[q,p] = \{f^D\}$$
$$f_{pq} = \textbf{True} \quad f_{qp} = \textbf{True}$$
$$q_{\texttt{Cycle}} = f_{qp} \wedge (|P_F[p,q] \geq 1|)$$

(b)

$$P_F[p,q] = \emptyset \qquad P_F[q,p] = \emptyset$$
$$f_{pq} = \textbf{False} \quad f_{qp} = \textbf{False}$$
$$q_{\texttt{Cycle}} = \textbf{True}$$

(c)

Fig. 5 The case for eager evaluation of Boolean functions involving p for `p = NULL`. **a** the updates of a heap graph. **b** and **c** data-flow values and Boolean equations before and after *S3*. Without eager evaluation, the equation for $q_{\texttt{Cycle}}$ will remain $f_{qp} \wedge (|P_F[p,q] \geq 1|)$ and hence evaluate to false after *S3*

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

$$\begin{array}{ll} f_{ps} = \textbf{False} & f_{sp} = \textbf{False} \\ P_F^{\text{kill}}[p,s] = P_F^{\text{in}}[p,s] & P_F^{\text{gen}}[p,s] = \emptyset \\ P_F^{\text{kill}}[s,p] = P_F^{\text{in}}[s,p] & P_F^{\text{gen}}[s,p] = \emptyset \\ P_F^{\text{kill}}[p,p] = P_F^{\text{in}}[p,p] & P_F^{\text{gen}}[p,p] = \{\epsilon\} \end{array}$$

4.1.3 `p = q, p = &(q→f), p = q op n`

In our analysis, we consider these three pointer assignment statements as equivalent. After this statement all the existing values of p get killed and it will point to same heap object (or NULL) as pointed to by q. So p will have the same paths and field connections as those of q. The kill effect of this statement is same as that of the previous cases. The generated Boolean functions for heap object p corresponding to DAG or Cycle attribute will also be same as those of q, with all occurrences of q replaced by p.[5]

[5] The notation $X[q/p]$ means a copy of Boolean equation X with all occurrences of $q$replaced by p.

$$p^{\text{kill}}_{\texttt{Cycle}} = p^{\text{in}}_{\texttt{Cycle}} \qquad p^{\text{kill}}_{\texttt{Dag}} = p^{\text{in}}_{\texttt{Dag}}$$
$$p^{\text{gen}}_{\texttt{Cycle}} = q^{\text{in}}_{\texttt{Cycle}}[q/p] \quad p^{\text{gen}}_{\texttt{Dag}} = q^{\text{in}}_{\texttt{Dag}}[q/p]$$

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

$$\begin{aligned} f_{ps} &= f_{qs} & f_{sp} &= f_{sq} \\ P^{\text{kill}}_F[p,s] &= P^{\text{in}}_F[p,s] & P^{\text{gen}}_F[p,s] &= P^{\text{in}}_F[q,s] \\ P^{\text{kill}}_F[s,p] &= P^{\text{in}}_F[s,p] & P^{\text{gen}}_F[s,p] &= P^{\text{in}}_F[s,q] \\ P^{\text{dkill}}_F[p,p] &= P^{\text{in}}_F[p,p] & P^{\text{gen}}_F[p,p] &= P^{\text{in}}_F[q,q] \end{aligned}$$

4.1.4 `p→f = NULL`

This statement breaks the existing link f emanating from p, thus killing equations of $p$that are due to link f. The statement does not generate new equations.

$$p^{\text{kill}}_{\texttt{Cycle}} = \textbf{False}, \quad p^{\text{kill}}_{\texttt{Dag}} = \textbf{False}$$
$$p^{\text{gen}}_{\texttt{Cycle}} = \textbf{False}, \quad p^{\text{gen}}_{\texttt{Dag}} = \textbf{False}$$

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

$$f_{pq} = \textbf{False}$$
$$P^{\text{kill}}_F[p,q] = P^{\text{in}}_F[p,q] \triangleright f \quad P^{\text{kill}}_F[s,q] = \emptyset$$

4.1.5 `p→f = q`

This statement first breaks the existing link f and then relinks 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 values here.

The fact that the shape of the variable p becomes DAG after the statement is captured by the Boolean functions $p^{\text{gen}}_{\texttt{Dag}}$. The function 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 path f_{pq} in this case. The function $q^{\text{gen}}_{\texttt{Cycle}}$ (or $p^{\text{gen}}_{\texttt{Cycle}}$) captures the fact that cycle on q (or p) consists of field f from p to q (f_{pq}) and some path from q to p ($|P_F[q,p]| \geq 1$). The function $p^{\text{gen}}_{\texttt{Cycle}}(p^{\text{gen}}_{\texttt{Dag}})$ also captures the fact that cycle (DAG) on p can be due to the link f_{pq} reaching an already existing cycle (DAG) on q. These are summarized as follows:

$$\begin{aligned} p^{\text{gen}}_{\texttt{Cycle}} &= (f_{pq} \wedge q^{\text{in}}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (|P_F[q,p]| \geq 1)) \\ p^{\text{gen}}_{\texttt{Dag}} &= (f_{pq} \wedge q^{\text{in}}_{\texttt{Dag}}) \vee (f_{pq} \wedge (|I_F[p,q]| > 1) \\ q^{\text{gen}}_{\texttt{Cycle}} &= f_{pq} \wedge (|P_F[q,p]| \geq 1) \\ q^{\text{gen}}_{\texttt{Dag}} &= \textbf{False} \\ f_{pq} &= \textbf{True} \end{aligned}$$

For nodes $s \in \mathcal{H}$ other than p or q, the function $s^{\text{gen}}_{\texttt{Cycle}}$ captures the fact that cycle on s consists of some path from s to p (or q), i.e. $|P_F[s,p]| \geq 1$ (or $|P_F[s,q]| \geq 1$) and the fact that a Cycle on p (or q) has just created due to the statement. Again the function $s^{\text{gen}}_{\texttt{Dag}}$ simply say that variable s reaches a DAG because there are more than one way of interference between s and q, i.e. $|I_F[s,q]| > 1$. It also keeps track of the paths f_{pq} and $P_F[s,p]$ in this case.

$$\begin{aligned} s^{\text{gen}}_{\texttt{Cycle}} = {} & ((|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge q^{\text{in}}_{\texttt{Cycle}}) \\ & \vee ((|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|P_F[q,p]| \geq 1)) \\ & \vee ((|P_F[s,q]| \geq 1) \wedge f_{pq} \wedge (|P_F[q,p]| \geq 1)), \\ & \forall s \in \mathcal{H}, \quad s \neq p, \quad s \neq q \\ s^{\text{gen}}_{\texttt{Dag}} = {} & (|P_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|I_F[s,q]| > 1), \\ & \forall s \in \mathcal{H}, \quad s \neq p, \quad s \neq q \end{aligned}$$

After the statement, all the nodes that have paths towards p (including p) will have path towards all the nodes reachable from q (including q). Thus,

For $r, s \in \mathcal{H}$:

$$\begin{aligned} P^{\text{gen}}_F[r,s] &= \left|P^{\text{in}}_F[q,s]\right| \star P^{\text{in}}_F[r,p], \quad s \neq p,\ r \notin \{p,q\} \\ P^{\text{gen}}_F[r,p] &= \left|P^{\text{in}}_F[q,p]\right| \star P^{\text{in}}_F[r,p], \quad r \neq p \\ P^{\text{gen}}_F[p,r] &= \left|P^{\text{in}}_F[q,r]\right| \star \left(P^{\text{in}}_F[p,p] \ominus \{\epsilon\} \cup \{f^{I1}\}\right), \\ & \quad r \neq q \\ P^{\text{gen}}_F[p,q] &= \{f^D\} \cup \left(\left|P^{\text{in}}_F[q,q] - \{\epsilon\}\right| \star \{f^{I1}\}\right) \cup \\ & \quad \left(\left|P^{\text{in}}_F[q,q]\right| \star \left(P^{\text{in}}_F[p,p] \ominus \{P^{\text{in}}_F[p,p] \triangleright f \cup \{\epsilon\}\}\right)\right) \\ P^{\text{gen}}_F[q,q] &= 1 \star P^{\text{in}}_F[q,p] \\ P^{\text{gen}}_F[q,r] &= \left|P^{\text{in}}_F[q,r]\right| \star P^{\text{in}}_F[q,p], \quad r \notin \{p,q\} \end{aligned}$$

4.1.6 `p = q→f`

The values killed by the statement are the same as those in the case of `p = NULL`. The values 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.

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

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

Note that shape of no other pointer variable gets affected by this statement.

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.

$$f_{qp} = \textbf{True}$$
$$h_{pr} = \left|P^{\text{in}}_F[q,r] \triangleright f\right| \geq 1 \quad \forall h \in \mathcal{F}, \quad \forall r \in \mathcal{H}$$

The equations to compute the generated values for P_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, this information 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 $P_F[p, s]$. The set $\mathcal{U}$is defined as:

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

Because it is also not possible to determine if there exists 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.shapeevaluates to `Cycle`). These observations result in the following equations:

$$\forall s \in \mathcal{H}, \quad s \neq p \wedge P_F^{\text{in}}[q, s] \triangleright f \neq \emptyset$$

$$P_1[p, s] = \begin{cases} \{\epsilon\} & (P_F^{\text{in}}[q, s] \triangleright f = f^D) \\ & \wedge\, q\text{.shape evaluates to } \texttt{Tree} \text{ or } \texttt{Dag} \\ \mathcal{U} & \text{Otherwise} \end{cases}$$

$$P_1[p, p] = s \begin{cases} \mathcal{U} & q\text{.shape evaluates to } \texttt{Cycle} \\ \{\epsilon\} & \text{Otherwise} \end{cases}$$

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

$$P_2[s, p] = \infty \star P_F^{\text{in}}[s, q] \quad \forall s \in \mathcal{H},\ s \notin \{p, q\}$$

If $p \neq q$, then we record the path from $q$to $p$as:

$$P_2[q, p] = \begin{cases} \{f^D\} & q\text{. shape evaluates to } \texttt{Tree} \\ \mathcal{U} & \text{Otherwise} \end{cases}$$

The third category of nodes to consider are those that interfere with the node corresponding to $q \to f$, without going through q. Any such node $s$will have paths to p after the statement. Thus, we have:

$$\forall r, \quad s \in \mathcal{H},\ r \notin \{p, q\}$$

$$P_3[s, p] = \bigcup_r \left\{\alpha \mid f^D \in P_F^{\text{in}}[q, r] \wedge \alpha \in P_F^{\text{in}}[s, r] \ominus P_F^{\text{in}}[s, q]\right\}$$

Finally, we compute the entries generated for P_F as:

$$P_F^{\text{gen}}[r, s] = P_1[r, s] \cup P_2[r, s] \cup P_3[r, s] \quad \forall r, s \in \mathcal{H}$$

4.2 Properties

In this section we discuss some properties of our analysis. First we discuss the need of introducing Boolean variables on the top of field-sensitive matrices. We then discuss termination guarantees for our analysis. We finally give bounds corresponding to the storage requirement of our analysis.

4.2.1 Need for Boolean variables

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

Example 6 Figure 6 shows a program fragment, and heap graph at the program point before statement *S1*. At *S2*, a DAG is created that is reachable from $r$which gets destroyed after *S3*. Figure 6c shows the path information between $r$and $p$and the Boolean variable f_{pq} at various program points. First we note that after statement *S2*, the analysis conservatively approximates the path $P_F[r, p]$ using the universal path $\mathcal{U}$. As a consequence, more than one pair of paths from $q$and $r$are considered interfering at t. Further, these entries will not be affected by the kill effects of statement *S3*. Thus, a `Dag` inferred using only the P_F matrix will continue to exist even after *S3*.

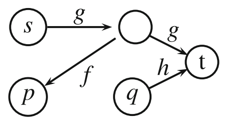
The use of field-based Boolean variables avoid this situation as follows. The fact that the shape of the variable $r$becomes `Dag` after *S2* is captured by the following Boolean function and variable:

$$\texttt{r}_{\texttt{Dag}} = (|P_F[r, p]| \geq 1) \wedge f_{pq} \wedge (|I_F[r, q]| > 1)$$
$$f_{pq} = \textbf{True}$$

After *S2*, $\texttt{r}_{\texttt{Dag}}$ becomes **True**, thus implying that r.shape is `Dag`. Later, at statement *S3*, the path due to f_{pq} is broken. Even though $|P_F[r, p]| \geq 1|$ and $|I_F[r, q]| > 1$ still hold

```
        . . .
S1:  r = s→g;
S2:  p→f = q;
S3:  p→f = NULL;
        . . .
```

(a)



(b)

After	f_{pq}	$P_F[r,p]$	$\lvert I_F[r,p]\rvert > 1$
S1	**False**	$\mathcal{U}$	**True**
S2	**True**	$\mathcal{U}$	**True**
S3	**False**	$\mathcal{U}$	**True**

(c)

Fig. 6 Using Boolean variables to improve precision. **a** program fragment, **b** the heap structure at the start of the fragment (before *S1*) , and **c** the values for the Boolean variables and paths between variables of interest

(because of presence of $\mathcal{U}$), we can still infer the shape transition from `Dag` to `Tree` because the Boolean variable f_{pq} becomes **False**, setting $\mathtt{r}_{\mathtt{Dag}}$ to **False**.

4.2.2 Termination

The computation of P_F matrices follows from the fact that the data flow functions are monotonic and the sets of approximate paths are bounded. The number of Boolean variables for a program is bounded by the number of pointer variables, and the fields in the program. Thus, the set of Boolean functions is also bounded as it uses a fixed set of Boolean variables and fixed operators ($\vee$ and $\wedge$). The termination of computation of Boolean functions for `Cycle` and `Dag` follows from the monotonicity of the flow functions [15, 18].

4.2.3 Storage requirement

The memory requirement of our analysis consists of the storage space for the path matrices (P_F) and the Boolean functions. Let $n \in \mathcal{N}$ denote the cardinality of the set $\mathcal{H}$ at a program point. Obviously n is bounded by the total number of pointer variables in the program. Let $m \in \mathcal{N}$ denotes the maximum number of possible distinct pointer fields emanating from a heap-directed pointer, which is again a bounded quantity. Between two pointers, for each field, we only use: (a) the (k-limited) count of the number of indirect paths starting at that given field, and (b) a direct path using that field (if there is one). The storage requirement for the path matrices P_F is thus bounded by $O(n^2 * m)$.

In our empirical evaluation, we have observed that the path matrices are sparse for large programs, as a pointer typically has paths only to a small set of heap nodes. Therefore, a sparse matrix-based representation can be used to store path matrices efficiently.

The Boolean functions at each program point are stored as bdds, using the *BuDDy* [19] library. As can be seen from the equations for the Boolean functions, the height and width of the expression tree for a function is polynomial in the number of pointer instructions in the program. However, because BDDs keep the Boolean functions in canonical form, it allow reusing the BDDs for the expressions at different program points. This results in an efficient storage of the Boolean functions.

5 Implementation and experimental results

We have implemented our technique as a context-sensitive, flow-sensitive interprocedural analysis [2] in C for GCC version 4.5.0 [1] as a dynamic plug-in. To achieve context sensitivity, we use *Call-strings*-based approach [22].

The analysis works on *GIMPLE*, the intermediate representation used by GCC. GIMPLE is a 3-address representation with at max one load/store per statement. To simplify the analysis, we treat each statement as a basic block in itself, except the call statement which is split into two blocks: call block and return block [22]. Our analysis computes the path matrix, Boolean field variables and Boolean shape functions at the IN and OUT of each basic block.

The analysis is preceded by a pre-processing phase that collects the information about heap pointers present in the program, for example the data type, number of fields, scope of the pointer variable. The analysis uses a worklist-based implementation of data flow analysis.

The data structures used by the analysis are as follows. We use dynamically allocated two dimensional sparse matrices to store the path information (P_F) at each program point. The field variables are stored in `bool` data type. We represent Boolean functions as Binary Decision Diagrams (BDDs) [4], using the *BuDDY* [19] package.

We have also implemented an existing field-insensitive shape analysis [11]. It is implemented as a context-sensitive, flow-sensitive analysis, and is used as baseline approach to compare against.

5.1 Benchmarks

We have experimented with benchmarks obtained from two sources. The *List* benchmarks [23] implement basic operations on linked lists in C++. This include the recursive and iterative implementations of the following operations on linked lists: insert, remove, delete all, search, append, merge and reverse. The *Olden* benchmarks [5] are used to show how our analysis performs on real life code with large number of complex function calls, including recursive calls. The method and results of our experiments are described in the rest of the section.

5.2 Methodology

We ran our experiments on a Intel® Xeon® 2.40 GHz CPU with 8 GB of RAM and 8 GB of swap space. The effectiveness of analysis is measured in terms of the following parameters:

Shapes inferred. We counted the number of each type of shapes (`Tree`, `Dag`, `Cycle`) reported by our analysis and the baseline approach. The analysis which reports more `Trees` and fewer `Dags` and `Cycles` is considered better.

Time for the analysis. We computed the total time taken by each of the analysis. While it is expected that the baseline approach will always have better time (due to fewer computations involved), we would like to have an estimate of the slowdown to better understand the scalability of our analysis.

Peak memory usage. Again, the baseline approach will always perform better in terms of peak memory usage for non-trivial programs because it does not store equations and

Table 3 Comparison of our analysis with baseline analysis [11] for List [23] and Olden [5] benchmarks

Benchmark	Baseline analysis			Our analysis			Ratio	
	#Shape (`Tree, Dag, Cycle`)	Time (T1) (MilliSec)	Memory (M1) (MB)	#Shape (`Tree, Dag, Cycle`)	Time (T2) (MilliSec)	Memory (M2) (MB)	Time T2/T1	Memory M2/M1
List Benchmarks								
100_create_iter	(99, 0, 0)	0.747	6.91	(99, 0, 0)	20.948	3.19	28.043	0.46
100_create_recur	(110, 0, 0)	1.124	6.12	(110, 0, 0)	24.098	8.26	21.440	1.35
200_delall_iter_create_fixed	(319, 0, 0)	1.671	5.33	(319, 0, 0)	25.23	6.82	15.099	1.28
200_delall_iter_create_iter	(154, 0, 0)	1.349	5.59	(154, 0, 0)	24.451	9.58	18.125	1.71
200_delall_recur_create_fixed	(308, 0, 0)	1.767	5.33	(308, 0, 0)	29.952	9.18	16.951	1.72
200_delall_recur_create_iter	(143, 0, 0)	1.626	4.8	(143, 0, 0)	24.366	8.26	14.985	1.72
300_insert_iter_create_fixed	(509, 51, 0)	3.398	6.64	(551, 9, 0)	46.745	6.82	13.757	1.03
300_insert_iter_create_iter	(269, 51, 0)	3.109	5.33	(311, 9, 0)	43.451	7.11	13.976	1.33
300_insert_recur_create_fixed	(429, 0, 0)	2.794	5.33	(425, 4, 0)	34.941	7.09	12.506	1.33
300_insert_recur_create_iter	(266, 0, 0)	2.653	9.75	(266, 0, 0)	38.683	7.12	14.581	0.73
400_remove_iter_create_fixed	(616, 0, 40)	2.348	7.17	(643, 13, 0)	28.161	8.40	11.994	1.17
400_remove_iter_create_iter	(360, 0, 40)	1.994	8.24	(387, 13, 0)	34.019	6.03	17.061	0.73
400_remove_recur_create_fixed	(540, 0, 0)	2.364	8.23	(540, 0, 0)	26.682	9.45	11.287	1.15
400_remove_recur_create_iter	(315, 0, 0)	2.07	6.91	(315, 0, 0)	27.923	4.98	13.489	0.72
500_search_iter_create_fixed	(403, 0, 0)	1.573	8.13	(403, 0, 0)	25.777	4.98	16.387	0.61
500_search_iter_create_iter	(208, 0, 0)	1.388	5.36	(208, 0, 0)	23.573	6.29	16.983	1.17
500_search_recur_create_fixed	(462, 0, 0)	1.862	5.85	(462, 0, 0)	30.201	10.28	16.220	1.76
500_search_recur_create_iter	(252, 0, 0)	1.531	8.91	(252, 0, 0)	31.247	6.82	20.410	0.76
600_append_iter_create_fixed	(544, 0, 0)	2.231	5.32	(537, 7, 0)	31.072	7.95	13.927	1.49
600_append_iter_create_iter	(304, 0, 0)	1.87	6.64	(297, 7, 0)	35.381	7.09	18.920	1.07
600_append_recur_create_fixed	(578, 0, 0)	2.718	5.32	(578, 0, 0)	31.57	12.17	11.615	2.29
600_append_recur_create_iter	(323, 0, 0)	2.196	5.86	(323, 0, 0)	34.458	6.82	15.691	1.16
700_merge_iter_create_fixed	(641, 0, 124)	3.93	6.91	(719, 46, 0)	44.398	7.08	11.297	1.02
700_merge_iter_create_iter	(432, 0, 138)	6.362	6.39	(452, 54, 64)	75.186	10.11	11.818	1.58
700_merge_recur_create_fixed	(840, 0, 0)	6.635	6.12	(840, 0, 0)	79.291	8.74	11.950	1.43
700_merge_recur_create_iter	(525, 0, 0)	6.238	5.59	(525, 0, 0)	78.413	7.09	12.570	1.27
800_reverse_iter_create_fixed	(470, 0, 55)	2.33	7.17	(495, 30, 0)	30.353	6.82	13.027	0.95
800_reverse_iter_create_iter	(245, 0, 55)	2.074	5.32	(270, 30, 0)	32.384	9.05	15.614	1.70
800_reverse_recur_create_fixed	(880, 0, 0)	4.253	6.38	(880, 0, 0)	44.728	6.82	10.517	1.07
800_reverse_recur_create_iter	(550, 0, 0)	3.804	5.36	(550, 0, 0)	46.117	8.00	12.123	1.49
Olden Benchmarks								
Health	(2800, 0, 216)	99.654	55.98	(3005, 4, 7)	3,184.563	211.77	31.956	3.78
MST	(1439, 6, 165)	40.341	6.18	(1665, 83, 0)	559.043	83.64	13.858	13.53
Power	(397, 63, 0)	13.774	4.93	(460, 0, 0)	458.805	106.11	33.309	21.52
em3d	(1008, 0, 0)	16.192	6.17	(1008, 0, 0)	164.311	61.30	10.148	9.94
Perimeter	(850, 0, 0)	30.451	5.45	(850, 0, 0)	30,095.358	2,477.67	988.321	454.62
TreeAdd	(63, 0, 0)	0.764	5.45	(63, 0, 0)	27.16	6.60	35.550	1.21
Tsp	(5718, 0, 277)	78.166	55.6	(5861, 0, 244)	80,144.484	6,499.13	1,025.311	116.89
Voronoi	(41318, 0, 532)	2,972.958	1,670.57	(41846, 0, 4)	43,843.493	3,548.43	14.747	2.12

only stores Boolean values in matrices as opposed to approximate path sets stored by our analysis. This metric also gives us an idea of the scalability of our analysis.

We set the limiting factor k to 3 for our analysis.

5.3 Results and explanations

Table 3 shows the comparison of our approach with the baseline approach for the above mentioned benchmarks.

The general observation is that our analysis is costly in terms of both memory usage as well as time taken. However, for complex programs, it is able to find precise shape for the heap structures.

For some simple benchmarks (for e.g., `100_create_iter`), our memory usage is better than baseline approach. The baseline approach stores interference matrix explicitly, that we do not store. For large programs, this saving is offset by the large number of paths stored in our path matrices. The benchmark `100_create_iter` and some other programs have very few paths among the pointers. Thus, the size of path matrices is small and hence our analysis takes less memory.

Our analysis gives safe results (more `Dags` as compared to baseline approach) for 3 of the benchmarks. These are:

- `300_insert_recur_create_fixed`,
- `600_append_iter_create_fixed`, and
- `600_append_iter_create_iter`

This is due to the fact that our computation of interference from path matrix can result in some spurious interference being detected. That is why our analysis sometimes infers spurious `Dags`. Since the shape `Cycle` does not depend on interference but only on paths, we never detect more cycles than the baseline approach.

Storing the path matrices in a sparse representation has resulted in significant memory and analysis-time savings as compared to an earlier simple matrix-based implementation. This is because typical programs use a large number of pointers that do not depend on each other. As a result the sizes of path matrices are huge, but they have very few non-empty entries. Sparse matrix representation avoids the storage associated with the empty entries. The savings in analysis-time is because only few entries have to be manipulated and due to caching effects.

Even with sparse matrix-based implementation, our analysis takes very long time and large amount of memory to analyze the benchmarks `Perimeter` and `Tsp`. The profiling of our analysis on these programs revealed that the these programs use a large number of pointers that interact with each other. As a result the path matrices are dense, nullifying the advantages of using sparse representation.

6 Enhancements

In this section, we propose some enhancements to our analysis. The implementation of these enhancements is still a work in progress.

6.1 Field-subset-based analysis

Many a times data structures include auxiliary fields used mainly for traversing the data structure for debugging or diagnostic purposes. The presence of such fields, however can result in imprecise shapes. We illustrate this using the following example.

Example 7 Consider the code segment in Fig. 7 which defines a function `search`, for searching data in a binary tree and a function `insert`, for inserting a node into a binary tree. In `insert`, a cycle is getting created between `s` and `p` after *S18*. Also the field-sensitive analysis infers the shape of *p* and *s* as cycles at that point.

Note that in the `search` function, the `parent` pointer is not at all used. Therefore, the shape of the heap graph accessible to (and traversed by) the `search` function is a Tree. But, as the function is called with the variable `root`, whose shape gets evaluated to `Cycle` in the `insert` function, we infer that the shape of the root remains `Cycle` at

```
       typedef struct treenode tree;
       struct treenode {
         tree left, right, parent;
         int key;
       };

       int main() {
 S1:     tree root;
 S2:     ...
 S3:     insert(root, key);
 S4:     search(root, key);
 S5:     ...
       }

       void insert(tree t, int key) {
S11:     tree s = root;
S12:     ... /* find the place to insert key */
S13:     tree p;
S14:     p->left = NULL;
S15:     p->right = NULL;
S16:     p->key = key;
S17:     s->left = p;
S18:     p->parent = s;
       }

       int search(tree t, int key) {
S21:     if (root)
S22:       return (key == root->key) ||
S23:         search(root->left,key) ||
S24:         search(root->right,key);
S25:     return 0;
       }
```

Fig. 7 The binary search program

all program points in the `search` function. This inhibits the parallel scheduling of recursive calls to `search` on lines *S23* and *S24*.

For the above case, unlike the field-sensitive analysis, field-subset-based analysis can correctly identify the shape by considering the fields accessed within a function and using only those fields to infer the shape. We now present intuition behind the analysis for the above example.

Example 8 Let us consider the simplified equation of $\mathtt{root}_{\mathtt{Cycle}}$ along with the relevant data-flow values at the end of statement *S18* (or at the beginning of `search`).

$$\mathtt{root}_{\mathtt{Cycle}} = (\mathtt{parent}_{\mathtt{p,root}} \wedge (|P_F[\mathtt{root,p}]| \geq 1))$$
$$\mathtt{parent}_{\mathtt{p,root}} = \mathbf{True}$$
$$P_F[\mathtt{root,p}] = \{\mathtt{left}^{\mathtt{D}}\}$$

In field-subset-based analysis, we use only those fields which are accessed within a function to evaluate the Boolean equations. In `search`, the field `parent` is not accessed, hence all the data-flow values (like the path matrix entries and Boolean variables) which use this field treat it as being absent (**False**). In this case, the Boolean variable $\mathtt{parent}_{\mathtt{p,root}}$ is treated as **False**, which makes the above equation of $\mathtt{root}_{\mathtt{Cycle}}$ **False**, inferring the shape of `root` as `Tree` ($\mathtt{root}_{\mathtt{Dag}}$ is also **False** in this case). A parallelizing compiler can schedule the recursive calls on lines *S23* and *S24* in parallel.

The key motivation behind using field-subset-based analysis is that if a function does not refer to a field f at all, it can not access sharing due to those `Cycles` or `Dags` that necessarily involve the field f. We plan to implement it in near future.

6.2 Shape-based context-sensitive analysis

In context-insensitive analysis we compromise on accuracy, whereas in context-sensitive analysis we compromise on memory consumption. To compromise between the memory consumption and precision, we propose a shape-based context-sensitive analysis which is midway between the above two approaches.

It is evident that merging all possible calling contexts during function call leads to conservative incoming data-flow values which results in inferring imprecise shapes. In shape-based context-sensitive approach, we keep separate set of data-flow values for each possible shape at the start of functions and the merging of calling contexts is based on the shape of the heap pointer arguments during function call.

We expect that this methods helps in reducing the memory consumption as compared to context-sensitive analysis without loosing much on precision. However, we are yet to complete the implementation of this approach and evaluate it experimentally.

7 Related work

The shape-analysis problem was initially studied in the context of functional languages. Jones and Muchnick [16] 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. [6] 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 $p \to f = q$ in some cases. However, this information is not insufficient to compute precise shape, and detects false cycles even in case of simple algorithms like destructive list reversal.

Sagiv et al. [24,25] 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. [10] 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. [8] extended this analysis to support nonlinear data structure (tree, graph etc.).

Berdine et al. [3] 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 [14] 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. [7] use the local abstraction scheme of [14] to generate local invariants to accurately compute shape information for complex data structures.

Jump and McKinley [17] 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.

The present work is an extension of the analysis developed by Sandeep et al. [9]. The major improvements in the current work include (a) removal of the computation of interference matrices (I_F), which improved the run time and the memory significantly, (b) improving the precision of some data-flow equations, (c) extensions to a call-strings based context-sensitive interprocedural analysis and (d) implementation and evaluation the analysis for some standard benchmarks. The work by Sandeep et al. [9] itself was an enhancement over the work proposed by Ghiya et al. [11] and by Marron et al. [20]. The analysis Ghiya et al. [11] 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 [12,13] 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 identifies the shapes. Marron et al. [20] presents a data flow framework that uses heap graphs to model data flow values. The analysis uses a technique that is similar to re-materialization. However, unlike parametric shape analysis techniques [25], 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 considerably.

8 Conclusion and future work

In this paper we proposed a field-sensitive shape analysis technique to infer shapes of heap structures. Our approach uses field-based Boolean variables along with field-sensitive path matrices to infer the shapes of pointer variables in terms of Boolean equations. The path matrices help in remembering connectivity information of pointers, while the field-based Boolean variables help in remembering the exact updates affecting the shapes. This allows our analysis to generate precise kill information for field updates, thereby capturing the shape transitions from Cycle to DAG, from Cycle to Tree and from DAG to Tree.

We have shown some scenarios that can be handled more precisely by our analysis as compared to existing field-insensitive analyses. To show the effectiveness of our analysis, we implemented our analysis as a plug-in for GCC version 4.5.0. The implementation is an instance of call-string-based interprocedural data-flow analysis framework. We evaluated our analysis on standard benchmarks and showed that the results are more precise than an existing field-insensitive analysis. We have shown some enhancements that can be easily incorporated in our analysis to increase its effectiveness in some cases.

Our shape analysis can be used by compilers for optimizations, parallelization and verification. There is a lot of scope to improve the memory and the time required by the analysis. We plan to implement with a demand-driven variation of the analysis that can switch between precision (field sensitivity) and speed (field insensitivity) depending on the needs of the target application. We also plan to implement and evaluate the enhancements to our analysis, namely Shape-based context-sensitive interprocedural analysis and Field-subset-based analysis, in near future.

References

1. GCC, the gnu compiler collection. http://gcc.gnu.org. Last accessed July 2012
2. Aho AV, Lam MS, Sethi R, Ullman JD (2006) Compilers: principles, techniques, and tools, 2nd edn. Prentice Hall, Englewood Cliffs
3. Berdine J, Calcagno C, Cook B, Distefano D, O'hearn PW, Yang H, Mary Q (2007) Shape analysis for composite data structures. In: CAV '07, pp 178–192. Springer, Berlin
4. Bryant Randal E (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput Surv 24(3): 293–318
5. Carlisle MC (1995) Olden benchmarks. http://www.martincarlisle.com/olden_benchmarks.tar.Z
6. Chase DR, Wegman M, Kenneth Zadeck F (1990) Analysis of pointers and structures. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. ACM, New York, pp 296–310
7. Cherem S, Rugina R (2007) 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. Springer, Berlin, pp 234–250
8. Cherini R, Rearte L, Blanco J (2010) A shape analysis for non-linear data structures. In: Proceedings of the 17th international conference on static analysis. Springer, Berlin, pp 201–217
9. Dasgupta S, Karkare A (2012) Precise shape analysis using field sensitivity. In: Proceedings of the 27th annual ACM symposium on applied computing, SAC '12, ACM, New York, pp 1300–1307

10. Distefano D, O'Hearn P, Yang H (2006) A local shape analysis based on separation logic. In: TACAS '06. Springer, Berlin, pp 287–302
11. Ghiya R, Hendren LJ (1996) Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in c. In POPL '96, pp 1–15
12. Ghiya R, Hendren LJ (1998) Putting pointer analysis to work. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, pp 121–133
13. Ghiya R, Hendren LJ, Zhu Y (1998) Detecting parallelism in c programs with recursive data structures. In: CC '98, pp 159–173
14. Hackett B, Rugina R (2005) Region-based shape analysis with tracked locations. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, New York, pp 310–323
15. Hecht MS (1997) Flow analysis of computer programs. Elsevier Science Inc., New York
16. Jones ND, Muchnick SS (1979) Flow analysis and optimization of lisp-like structures. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages. ACM, New York, pp 244–256
17. Jump M, McKinley KS (2009) Dynamic shape analysis via degree metrics. In: Proceedings of the 2009 international symposium on memory management. ACM, New York, pp 119–128
18. Kam John B, Ullman Jeffrey D (1977) Monotone data flow analysis frameworks. Acta Inform 7:305–317
19. Lind-Nielsen J (2012) BuDDy: A Binary Decision Diagram library. http://buddy.sourceforge.net, last accessed July 2012
20. Marron M, Kapur D, Stefanovic D, Hermenegildo M (2006) 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. Springer, Berlin, pp 345–363
21. Mattson T, Wrinn M (2008) Parallel programming: can we please get it right this time? In: Proceedings of the 45th annual Design Automation Conference, DAC '08. ACM, New York, pp 7–11
22. Puneli MSA (1976) Two approaches to inter procedural data flow analysis. In: Program flow analysis: theory and applications. pp 189–234
23. Pavlu V (2010) Basic operations on linked lists (c++). http://www.complang.tuwien.ac.at/vpavlu/2010/list-benchmark.tgz
24. Sagiv M, Reps T, Wilhelm R (1996) Solving shape-analysis problems in languages with destructive updating. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on principles of programming languages. ACM, New York, pp 16–31
25. Sagiv Shmuel, Reps Thomas W, Wilhelm Reinhard (2002) Parametric shape analysis via 3-valued logic. ACM TOPLAS 24(3):217–298
26. Sessions R (2009) The IT Complexity Crisis: Danger and Opportunity. http://www.objectwatch.com/whitepapers/ITComplexityWhitePaper.pdf. Last accessed July 2012
27. Shaham R, Yahav E, Kolodner EK, Sagiv S (2003) Establishing local temporal heap safety properties with applications to compile-time memory management. In: Proceedings of the 10th international symposium on static analysis. Springer, London, pp 483–503
28. Wrinn M (2008) Top 10 challenges in parallel computing. http://software.intel.com/en-us/blogs/2008/12/31/top-10-challenges-in-p. Last accessed July 2012