

# Precise Shape Analysis using Field Sensitivity

*A Thesis Submitted  
in Partial Fulfillment of the Requirements  
for the Degree of*  
**Master of Technology**

by

**Sandeep Dasgupta**

**Y9111031**

*under the guidance of*

**Prof. Amey Karkare**

and

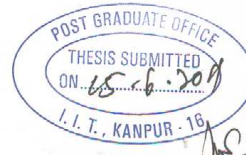
**Prof. Sanjeev K Aggarwal**



Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

June 2011

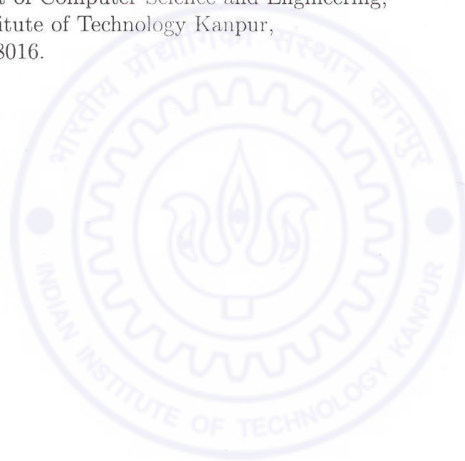


## CERTIFICATE

It is certified that the work contained in this thesis entitled "Precise Shape Analysis using Field Sensitivity", by Sandeep Dasgupta (Roll No. Y9111031), has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare /13/6/11  
(Professor Amey Karkare)  
Department of Computer Science and Engineering,  
Indian Institute of Technology Kanpur,  
Kanpur-208016.

Sanjeev K  
(Professor Sanjeev K Aggarwal)  
Department of Computer Science and Engineering,  
Indian Institute of Technology Kanpur,  
Kanpur-208016.



## **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 work, 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.

*This thesis is dedicated  
to my sister Mousumi Dasgupta.*



# Acknowledgment

I would like to express my heartiest gratitude towards my thesis supervisor **Professor Amey Karkare** for his invaluable support and encouragement. His guidance has helped me from initial till the final level of this work on the way developing a sound understanding of the topic. It would have been very difficult for me to complete this work without his innumerable innovative suggestions, the freedom that he has provided and his utmost patience in hearing my problems out. Apart from gaining a lot from his huge technical expertise, starting from gaining tips about how to write a good report to getting insights whenever I got struck, I got many learnings which I will explore throughout my life.

I wish to thank **Prof. Sanjeev K. Aggarwal**, for his immense support and invaluable suggestions that I have received from him.

I would also like to thank **Prof. Subhajit Roy** for all the enriching discussions that I had with him.

Special thanks to my friend and fellow mate **Ms. Barnali Basak** for her encouragement and enormous help and lots of thanks to my institution without which this thesis would have been a distant reality.

Finally, I wish to extend my thanks to my parents, my sister and my beloved friends, who have always been a source of inspiration and encouragement. Their love and blessings have always been a driving force in my life to carry all my works with hope and optimism.

Every effort has been made to give credit where it is due for the material contained here in. If inadvertently I have omitted giving credit to anyone, I apologize and express my gratitude for their contribution to this work.

*Sandeep Dasgupta*

*June 2011*

*Indian Institute of Technology, Kanpur*

## **Declaration**

I declare that this written submission represents my ideas in my own words except Appendix A and B whose contents are borrowed from Ghiya et. al. [GH96] and Marron et. al. [MKSH06] respectively for the ease of reference. The places where others' ideas or words have been included, I have adequately cited and referenced the original sources.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	A Brief Introduction . . . . .	1
1.2	Contributions of our Work . . . . .	2
1.3	Organization of the Thesis . . . . .	2
<b>2</b>	<b>Related Work</b>	<b>3</b>
<b>3</b>	<b>Motivation</b>	<b>5</b>
<b>4</b>	<b>Definitions and Notations</b>	<b>9</b>
<b>5</b>	<b>Analysis</b>	<b>15</b>
5.1	Analysis of Basic Statements . . . . .	16
5.2	Inter-procedural Analysis . . . . .	21
<b>6</b>	<b>Properties of our Analysis</b>	<b>23</b>
6.1	Need for Boolean Variables . . . . .	23
6.2	Termination . . . . .	24
6.3	Storage Requirement . . . . .	24
<b>7</b>	<b>Comparison with Other Approaches</b>	<b>27</b>
7.1	Comparison with Ghiya et. al. [GH96] . . . . .	27
7.1.1	Inserting an internal node in a singly linked list. . . . .	27
7.1.2	Swapping Two Nodes of a Singly Linked List. . . . .	27
7.1.3	Recursively Swapping Binary Tree. . . . .	28
7.2	Comparison with Marron et. al. [MKSH06] . . . . .	29
<b>8</b>	<b>Conclusion and Future Work</b>	<b>33</b>
<b>A</b>	<b>Analysis of Ghiya et. al. [GH96]</b>	<b>35</b>
<b>B</b>	<b>Analysis of Marron et. al. [MKSH06]</b>	<b>41</b>





# List of Figures

3.1	A motivating example . . . . .	6
3.2	Field Sensitive information for the code in Figure 3.1(a) . . . . .	6
4.1	Paths in a heap graph . . . . .	10
4.2	A heap graph and its field sensitive path matrices . . . . .	12
6.1	Using boolean variables to improve precision . . . . .	24
6.2	The termination of computation of boolean functions . . . . .	24
7.1	Insertion of an internal node in a singly linked list . . . . .	28
7.2	Swapping two nodes of a singly linked list . . . . .	29
7.3	Computing mirror image of a binary tree. “ <i>l</i> ” and “ <i>r</i> ” denotes respectively the left and right fields of tree. . . . .	30
A.1	Example Direction and Interference Matrices . . . . .	36
A.2	Example Demonstrating Shape Estimation . . . . .	37
A.3	The Overall Struture of the Analysis . . . . .	38
A.4	Analysis Rules . . . . .	39



# List of Tables

4.1	Determining shape from boolean attributes . . . . .	11
4.2	Path removal, intersection and union operations, where $\gamma$ denotes any other path.	13
4.3	Multiplication by a scalar . . . . .	13



# Chapter 1

## Introduction

### 1.1 A Brief 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 [SRW96, SYKS03, DOY06, HR05] are not practical as they do not scale to the size of complex heap manipulating programs. To achieve scalability, the practical shape analysis algorithms [CWZ90, GH96, MKSH06] trade precision for speed.

In this report, 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.

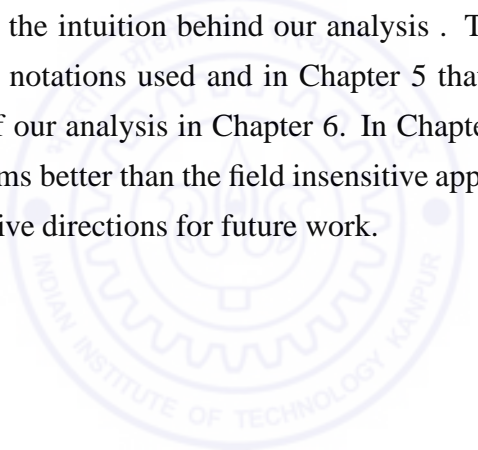
The field sensitivity information is captured in two ways: (a) we use field based boolean variables to remember the direct connections between two pointer variables, and (b) we compute field sensitive matrices that store the approximate path information between two pointer variable. We generate boolean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

## 1.2 Contributions of our Work

Our work contributes in the area of analysing pointers that points to dynamically allocated objects (typically in the heap). Inferring the shape of the data structure pointed to by heap allocated objects can be used for disambiguating heap accesses originating from that pointer. We proposed a field sensitive shape analysis technique that helps in precise inference of the shape of the heap data structures. As any field sensitive shape analysis algorithm must remember all paths between pointers. Our analysis uses certain approximations to remember such paths. They include: (a) DF :Modified direction matrix that stores the first fields of the paths between two pointers; (b) IF : 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. ‘

## 1.3 Organization of the Thesis

We discuss some of the prior works on shape analysis in Chapter 2. A motivating example is used in Chapter 3 to explain the intuition behind our analysis . The analysis is formalized in Chapter 4 that describes the notations used and in Chapter 5 that gives the analysis rules. We describe some properties of our analysis in Chapter 6. In Chapter 7 we show some of the cases where our analysis performs better than the field insensitive approaches. We conclude the presentation in Chapter 8 and give directions for future work.



# Chapter 2

## Related Work

The shape-analysis problem was initially studied in the context of functional languages. Jones and Muchnick [JM79] 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. [CWZ90] introduced the concept of limited reference count to classify heap objects into different shapes. They also classified the nodes in concrete and summary nodes, where summary nodes were used to guarantee termination. Using the reference count and concreteness information of the node, they were able to kill relations (*strong updates*) for assignments of the form  $p \rightarrow f = q$  in some cases. However, this information is not insufficient to compute precise shape, and detects false cycle even in case of simple algorithms like destructive list reversal.

Sagiv et. al. [SRW96, SRW99, SRW02] 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. [DOY06] 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. [CRB10] extended this analysis to support nonlinear data structure (tree, graph etc.).

Berdine et al. [BCC<sup>+</sup>07] 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 [HR05] 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. Chorem et. al. [CR07] use the local abstraction scheme of [HR05] to generate local invariants to accurately compute shape information for complex data structures.

Jump and McKinley [JM09] 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. [GH96] and by Marron et. al. [MKSH06]. Ghiya et al. [GH96] 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 [Ghi96, GH98, GHZ98] 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 conservatively identify the shapes.

Marron et. al. [MKSH06] presents a data flow framework that uses heap graphs to model data flow values. They presented an abstract heap model that can represent information on aliasing, shape, logical data structures, sharing between variables, and sharing between data elements in collections. They introduce a restricted version of refinement, based on the ideas presented by Sagiv, Reps and Wilhelm. Using this restricted notion of refinement, they demonstrate how this model can be used to accurately simulate important program events such as list copying, sorting, reversal, and various other destructive operations. The analysis uses technique similar to re-materialization, but unlike parametric shape analysis techniques [SRW99, SRW02], 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 functions 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 chapter presents a simplified view of our approach before we explain it in full details.

As our work is closest to the work of Ghiya et. al. [GH96] we present a brief summary of their analysis in Appendix A.

# Chapter 3

## Motivation

For each pointer variable, our analysis computes the shape attribute of the data structure pointed to by the variable. Following the existing literature [GH96, SRW96, Ghi96, MKSH06], we define the shape attribute  $p.\text{shape}$  for a pointer  $p$  as follows:

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

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

We use the code fragment in Fig. 3.1(a) to motivate the need for a field sensitive shape analysis.

**Example 3.1.** Consider the code segment in the Fig. 3.1(a), At  $S_4$ , a DAG is created that is reachable from  $p$ . At  $S_5$ , a cycle is created that is reachable from both  $p$  and  $q$ . This cycle is destroyed at line  $S_6$  and the DAG is destroyed at  $S_7$ .

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. [GH96] can correctly report the shape transition from DAG to cycle (at  $S_5$ ), but fails to infer the shape transition from cycle to DAG (at  $S_6$ ) and from DAG to Tree (at  $S_7$ ). This is evident from Fig. 3.1(b) that shows the Direction ( $D$ ) and Interference ( $I$ ) matrices computed using their algorithm. We get conservative shape information at  $S_6$  and  $S_7$  because the kill-effect of statements  $S_6$  and  $S_7$  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 (Chapter 5).

**Example 3.2.** The statement at  $S_4$  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



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

(a) A code fragment

After Stmt	D			I		
		<i>p</i>	<i>q</i>		<i>p</i>	<i>q</i>
S1	<i>p</i>	1	0	<i>p</i>	1	0
	<i>q</i>	0	0	<i>q</i>	0	0
S2	<i>p</i>	1	0	<i>p</i>	1	0
	<i>q</i>	0	1	<i>q</i>	0	1
S3	<i>p</i>	1	1	<i>p</i>	1	1
	<i>q</i>	0	1	<i>q</i>	1	1
S4	<i>p</i>	1	1	<i>p</i>	1	1
	<i>q</i>	0	1	<i>q</i>	1	1
S5, S6, S7	<i>p</i>	1	1	<i>p</i>	1	1
	<i>q</i>	1	1	<i>q</i>	1	1

(b) Direction (*D*) and Interference (*I*) matrices as computed by [GH96]

Figure 3.1: A motivating example

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

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

After Stmt	S1	S2	S3	S4	S5	S6	S7
<b>Boolean Vars</b>							
$f_{pq}$	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.

Figure 3.2: Field Sensitive information for the code in Figure 3.1(a)

remember all paths from  $p$  to  $q$ . Our analysis approximates any path between two variables by the first field that is dereferenced on the path. Further, as there may be an unbounded number of paths between two variables, we use  $k$ -limiting to approximate the number of paths starting at a given field.

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

Figures 3.2(a) and 3.2(b) show the values computed by our analysis for the example program. In this case, the fact that the shape of the variable  $p$  becomes DAG after S4 is captured by the following boolean functions<sup>1</sup>:

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

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

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

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

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

Here, the functions captures the fact that cycle on  $p$  consists of field  $g$  from  $q$  to  $p$  ( $g_{qp}$ ) and some path from  $p$  to  $q$  ( $|D_F[p, q]| \geq 1$ ). This cycle is broken either when the path from  $p$  to  $q$  is broken ( $|D_F[p, q]| = 0$ ) or when the link  $g$  changes ( $g_{qp} = \mathbf{False}$ ). The latter occurs after S6 in Fig. 3.2(a). □

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

---

<sup>1</sup>The functions and values shown in this example and in Fig. 3.2 are simplified to avoid references to concepts not defined yet.



# Chapter 4

## Definitions and Notations

We view the heap structure at a program point as a directed graph, the nodes of which represent the allocated objects and the edges represent the connectivity through pointer fields. Pictorially, inside a node we show all the relevant pointer variables that can point to the heap object corresponding to that node. The edges are labeled by the name of the corresponding pointer field. In this report, 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<sup>1</sup>. To distinguish between a path of length one (direct path) from a path of length greater than one (indirect path) that start at the same field, we use the superscript  $D$  for a direct path and  $I$  for an indirect path. In pictures, we use solid edges for direct paths, and dotted edges for indirect paths.

It is also possible to have multiple paths between two pointers starting at a given field  $f$ , with at most one direct path  $f^D$ . However, the number of indirect paths  $f^I$  may be unbounded. As there can only be a finite number of first fields, we store first fields of paths, including the count for the indirect paths, between two pointer variables in a set. To bound the size of the set, we put a limit  $k$  on number of repetitions of a particular field. If the number goes beyond  $k$ , we treat the number of paths with that field as  $\infty$ . This approach is similar to the approach of  $k$ -limiting [JM79]. and  $sl$ -limiting [LH88].

**Example 4.1.** Figure 4.1(a) shows a code fragment and Fig. 4.1(b) shows a possible heap graph at a program point after line S5. In any execution, there is one path between  $p$  and  $q$ , starting with field  $f$ , whose length is statically unknown. This information is stored by our analysis as

---

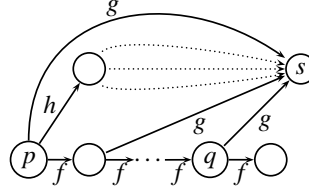
<sup>1</sup>The decision to use only first field is guided by the fact that in our language, a statement can use at most one field, i.e.  $p \rightarrow f = \dots$  or  $\dots = p \rightarrow f$ . While it is possible to use prefixes of any fixed length in case of languages using more than one fields, it does not make any fundamental change to our analysis.

```

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

```

(a) A code fragment



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

Figure 4.1: Paths in a heap graph

the set  $\{f^{l1}\}$ . 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  $\{g^D, f^{l\infty}, h^{l3}\}$ . On the other hand, if  $k < 3$ , then the set would be  $\{g^D, f^{l\infty}, h^{l\infty}\}$ .  $\square$

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

**Definition 4.1.** *Field sensitive Direction matrix  $D_F$  is a matrix that stores information about paths between two pointer variables. Given  $p, q \in \mathcal{H}, f \in \mathcal{F}$ :*

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

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

$$\varepsilon \sqsubseteq \varepsilon, \quad f^D \sqsubseteq f^D, \quad f^{l\infty} \sqsubseteq f^{l\infty}, \quad f^{lm} \sqsubseteq f^{l\infty}, \quad f^{ln} \sqsubseteq f^{lm}.$$

The partial order is extended to set of paths  $S_{P_1}, S_{P_2}$  as<sup>2</sup>:

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

For pair of paths:

$$(\alpha, \beta) \sqsubseteq (\alpha', \beta') \Leftrightarrow (\alpha \sqsubseteq \alpha') \wedge (\beta \sqsubseteq \beta')$$

<sup>2</sup>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  $f^l$

Table 4.1: Determining shape from boolean attributes

$p_{\text{Cycle}}$	$p_{\text{Dag}}$	$p.\text{shape}$
<b>True</b>	Don't Care	Cycle
<b>False</b>	<b>True</b>	DAG
<b>False</b>	<b>False</b>	Tree

For set of pairs of paths  $R_{P_1}, R_{P_2}$ :

$$R_{P_1} \sqsubseteq R_{P_2} \Leftrightarrow \forall (\alpha, \beta) \in R_{P_1}, \exists (\alpha', \beta') \in R_{P_2} \text{ s.t. } (\alpha, \beta) \sqsubseteq (\alpha', \beta')$$

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

**Definition 4.2.** *Field sensitive Interference matrix  $I_F$  between two pointers captures the ways in which these pointers are interfering. For  $p, q, s \in \mathcal{H}, p \neq q$ , the following relation holds for  $D_F$  and  $I_F$ :*

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

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

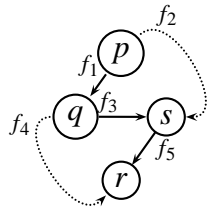
$$(\alpha, \beta) \in I_F[p, q] \Leftrightarrow (\beta, \alpha) \in I_F[q, p] .$$

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

**Example 4.2.** Figure 4.2 shows a heap graph and the corresponding field sensitive matrices as computed by our analysis. □

As mentioned earlier, for each variable  $p \in \mathcal{H}$ , our analysis uses attributes  $p_{\text{Dag}}$  and  $p_{\text{Cycle}}$  to store boolean functions telling whether  $p$  can reach a DAG or cycle respectively in the heap. The boolean functions consist of the values from matrices  $D_F$ ,  $I_F$ , and the field connectivity information. For  $f \in \mathcal{F}, p, q \in \mathcal{H}$ , field connectivity is captured by boolean variables of the form  $f_{pq}$ , which is true when  $f$  field of  $p$  points directly to  $q$ . The shape of  $p$ ,  $p.\text{shape}$ , can be obtained by evaluating the functions for the attributes  $p_{\text{Cycle}}$  and  $p_{\text{Dag}}$ , and using Table 4.1.

We use the following operations in our analysis. Let  $S$  denote the set of approximate paths



(a) Heap graph

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

(b) Direction Matrix

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

(c) Interference Matrix

Figure 4.2: A heap graph and its field sensitive path matrices

between two nodes,  $P$  denote a set of pair of paths, and  $k \in \mathcal{N}$  denotes the limit on maximum indirect paths stored for a given field. Then,

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

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

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

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

$$|S| = \sum_{\alpha \in S} |\alpha|$$

- 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 4.2. 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:

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

Table 4.2: Path removal, intersection and union operations, where  $\gamma$  denotes any other path.

(a) Path removal						(b) Intersection							
$\ominus$	$\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{lj}\}$	$\{f^{l\infty}\}$	$\{\gamma\}$	$\cap$	$\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{lj}\}$	$\{f^{l\infty}\}$	$\{\gamma\}$
$\{\alpha\}$							$\{\alpha\}$						
$\{\varepsilon\}$		$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\varepsilon$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^D\}$		$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\{f^D\}$	$\{f^D\}$	$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^{li}\}$		$\{f^{li}\}$	$\emptyset$	$\{f^{lm}\}$	$\emptyset$	$\{f^{li}\}$	$\{f^{li}\}$	$\emptyset$	$\emptyset$	$\{f^{lm}\}$	$\{f^{li}\}$	$\emptyset$	$\emptyset$
$\{f^{l\infty}\}$		$\{f^{l\infty}\}$	$\emptyset$	$\{f^{l\infty}\}$	$\{f^{l\infty}\}$	$\{f^{l\infty}\}$	$\{f^{l\infty}\}$	$\emptyset$	$\emptyset$	$\{f^{lj}\}$	$\{f^{l\infty}\}$	$\emptyset$	$\emptyset$

(c) Union						
$\cup$	$\{\beta\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{lj}\}$	$\{f^{l\infty}\}$	$\{\gamma\}$
$\{\alpha\}$						
$\{\varepsilon\}$		$\{\varepsilon\}$	$\{\varepsilon, f^D\}$	$\{\varepsilon, f^{lj}\}$	$\{\varepsilon, f^{l\infty}\}$	$\{\varepsilon, \gamma\}$
$\{f^D\}$		$\{f^D, \varepsilon\}$	$\{f^D\}$	$\{f^D, f^{lj}\}$	$\{f^D, f^{l\infty}\}$	$\{f^D, \gamma\}$
$\{f^{li}\}$		$\{f^{li}, \varepsilon\}$	$\{f^{li}, f^D\}$	$\{f^{li}\}$	$\{f^{l\infty}\}$	$\{f^{li}, \gamma\}$
$\{f^{l\infty}\}$		$\{f^{l\infty}, \varepsilon\}$	$\{f^{l\infty}, f^D\}$	$\{f^{l\infty}\}$	$\{f^{l\infty}\}$	$\{f^{l\infty}, \gamma\}$

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

Table 4.3: Multiplication by a scalar

$\star$	$\alpha$	$\varepsilon$	$f^D$	$f^{lj}$	$f^{l\infty}$
$i$					
$i$		$\varepsilon$	$f^{li}$	$f^{lm}, m = \begin{cases} i * j & \text{if } i * j \leq k \\ \infty & \text{Otherwise} \end{cases}$	$f^{l\infty}$
$\infty$		$\varepsilon$	$f^{l\infty}$	$f^{l\infty}$	$f^{l\infty}$

- Multiplication by a scalar( $\star$ ): Let  $i, j \in \mathcal{N}, i \leq k, j \leq k$ . Then, for a path  $\alpha$ , the multiplication by a scalar  $i$ ,  $i \star \alpha$  is defined in Table 4.3. The operation is extended to set of paths as:

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





# Chapter 5

## Analysis

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

### 1. Allocations

(a)  $p = \text{malloc}();$

### 2. Pointer Assignments

(a)  $p = \text{NULL};$

(b)  $p = q;$

(c)  $p = q \rightarrow f;$

(d)  $p = \&(q \rightarrow f);$

(e)  $p = q \text{ op } n;$

### 3. Structure Updates

(a)  $p \rightarrow f = q;$

(b)  $p \rightarrow f = \text{NULL};$

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

denote the values coming along two paths,

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

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

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

Field connectivity information is updated directly by the statement.

## 5.1 Analysis of Basic Statements

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

1.  $p = \text{malloc}()$ : After this statement all the existing relationships of  $p$  get killed and it will point to a newly allocated object. We will consider that  $p$  can have an empty path to itself and it can interfere with itself using empty paths (or  $\varepsilon$  paths).

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

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

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

2.  $p = \text{NULL}$ : This statement only kills the existing relations of  $p$ .

$$\begin{array}{ll} p_{\text{Cycle}}^{\text{kill}} = p_{\text{Cycle}}^{\text{in}} & p_{\text{Dag}}^{\text{kill}} = p_{\text{Dag}}^{\text{in}} \\ p_{\text{Cycle}}^{\text{gen}} = \mathbf{False} & p_{\text{Dag}}^{\text{gen}} = \mathbf{False} \end{array}$$

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

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

3.  $p = q$ ,  $p = \&(q \rightarrow f)$ ,  $p = q \text{ op } n$ : In our analysis we consider these three pointer assignment statements as equivalent. After this statement all the existing relationships of  $p$  gets killed and it will point to same heap object as pointed to by  $q$ . In case  $q$  currently points to null,  $p$  will also points to null after the statement. So  $p$  will have the same field sensitive Direction and Interference relationships as  $q$ . The kill effect of this statement is same as that of the previous statement. The generated boolean functions for heap object  $p$  corresponding to DAG or Cycle attribute will be same as that of  $q$ , with all occurrences of  $q$  replaced by  $p$ .

$$\begin{array}{ll} p_{\text{Cycle}}^{\text{kill}} = p_{\text{Cycle}}^{\text{in}} & p_{\text{Dag}}^{\text{kill}} = p_{\text{Dag}}^{\text{in}} \\ p_{\text{Cycle}}^{\text{gen}} = q_{\text{Cycle}}^{\text{in}}[q/p] & p_{\text{Dag}}^{\text{gen}} = q_{\text{Dag}}^{\text{in}}[q/p] \end{array}$$

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

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

$$\begin{array}{ll} f_{ps} = f_{qs}, & f_{sp} = f_{sq} \\ D_F^{\text{kill}}[p, s] = D_F^{\text{in}}[p, s] & D_F^{\text{gen}}[p, s] = D_F^{\text{in}}[q, s] \\ D_F^{\text{kill}}[s, p] = D_F^{\text{in}}[s, p] & D_F^{\text{gen}}[s, p] = D_F^{\text{in}}[s, q] \\ D_F^{\text{kill}}[p, p] = D_F^{\text{in}}[p, p] & D_F^{\text{gen}}[p, p] = D_F^{\text{in}}[q, q] \\ I_F^{\text{kill}}[p, s] = I_F^{\text{in}}[p, s] & I_F^{\text{gen}}[p, s] = I_F^{\text{in}}[q, s] \\ I_F^{\text{kill}}[p, p] = I_F^{\text{in}}[p, p] & I_F^{\text{gen}}[p, p] = I_F^{\text{in}}[q, q] \end{array}$$

4.  $p \rightarrow f = \text{null}$ : This statement breaks the existing link  $f$  emanating from  $p$ , thus killing relations of  $p$ , that are due to the link  $f$ . The statement does not generate any new rela-

tions. The killed relationships involve breaking of the all the direct links of  $p$ , which is manifested by setting  $f_{pq}$  to 0,  $\forall q \in \mathcal{H}$  as well as the indirect links of  $p$ , which is done by updating the  $D_F$  and  $I_F$  matrices. Other links between pointers which come up due to field  $f$  of  $p$  are conservatively approximated to exist even after the statement. The relation  $I_F[p, p]$  is not much relevant to our analysis, so the corresponding kill information is ignored.

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

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

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

5.  $p \rightarrow f = q$ : 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  $p \rightarrow f = \text{null}$ . We only describe the generated relationships here.

The fact that the shape of the variable  $p$  becomes DAG after the statement is captured by the boolean functions  $p_{\text{Dag}}^{\text{gen}}$  and  $f_{pq}$ . The functions simply say that variable  $p$  reaches a DAG because there are more than one paths ( $|I_F[p, q]| > 1$ ) from  $p$  to  $q$ . It also keeps track of the path  $f_{pq}$  in this case. The function  $q_{\text{Cycle}}^{\text{gen}}$  (or  $p_{\text{Cycle}}^{\text{gen}}$ ) 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  $p$  ( $|D_F[q, p]| \geq 1$ ). The function  $p_{\text{Cycle}}^{\text{gen}}$  also captures the fact that cycle on  $p$  can be due to the link  $f_{pq}$  reaching an already existing cycle on  $q$ . These are summarized as follows:

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

For nodes  $s \in \mathcal{H}$  other than  $p$  or  $q$ , the function  $s_{\text{Cycle}}^{\text{gen}}$  captures the fact that cycle on  $s$  consists of some path from  $s$  to  $p$  (or  $q$ ) i.e.  $|D_F[s, p]| \geq 1$  (or  $|D_F[s, q]| \geq 1$ ) and the fact that a Cycle on  $p$  (or  $q$ ) has just created due to the statement. Again the function  $s_{\text{Dag}}^{\text{gen}}$  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  $D_F[s, p]$  in this case.

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

After the statement, all the nodes which have paths towards  $p$  (including  $p$ ) will have path towards all the nodes reachable from  $q$  (including  $q$ ). Again all nodes having paths to  $p$  can potentially interfere with all the node  $q$  interferes with. Thus the relations generated for  $D_F$  and  $I_F$  are as follows.

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

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

6.  $p = q \rightarrow f$ : The relations killed by the statement are same as that in case of  $p = \text{NULL}$ . The relations created by this statement are heavily approximated by our analysis. After this statement  $p$  points to the heap object which is accessible from pointer  $q$  through  $f$  link. The only inference we can draw is that  $p$  is reachable from any pointer  $r$  such that  $r$

reaches  $q \rightarrow f$  before the assignment. This information is available because  $I_F^{in}[q, r]$  will have an entry of the form  $(f^D, \alpha)$  for some  $\alpha$ .

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

$$p_{\text{Cycle}}^{gen} = q_{\text{Cycle}}^{in} \quad p_{\text{Dag}}^{gen} = q_{\text{Dag}}^{in}$$

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

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

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

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

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

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

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

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

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

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

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

Finally, we compute the  $I_F$  and  $D_F$  relations as:

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

## 5.2 Inter-procedural Analysis

To handle procedure calls we use simple inter-procedural analysis that works by creating an invocation graph of the program. To handle recursive calls, for which the invocation structure is statically unknown, we use approximate summary for one of the nodes involved in the recursion chain, and use it to break the cycle. At each call site, the two matrices ( $D_F$  and  $I_F$ ) along with the boolean functions are fed as input to the called procedure, after proper mapping between formal and actual arguments. The called procedure is then analyzed to create the corresponding output matrices and the boolean functions. This approach is similar to the work by Ghiya et. al. [GH96].





# Chapter 6

## Properties of our Analysis

In this chapter 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 how we guarantee termination of our analysis. We also give bounds corresponding to the storage requirement of our analysis.

### 6.1 Need for Boolean Variables

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

**Example 6.1.** Fig. 6.1(a) shows a program fragment, and Fig. 6.1(b) shows a possible heap graph at a program point before statement S1. At S2, a DAG is created that is reachable from  $r$  which gets destroyed after S3. Fig. 6.1(c) shows the Direction ( $D_F$ ) and Interference ( $I_F$ ) matrices at various program points. After statement S2 we conservatively approximated the entries  $D_F[r, p]$  and  $I_F[r, q]$  using the universal path  $\mathcal{U}$  and as a consequence those entries will not be affected by the kill-effects of statement S3. However, using boolean variables, the fact that the shape of the variable  $r$  becomes DAG after S2 is captured by the following boolean functions:

$$r_{\text{Dag}} = (f_{pq} \wedge (|I_F[r, q]| > 1)) \quad f_{pq} = \mathbf{True}$$

After S2,  $r_{\text{Dag}}$  becomes true, thus implying that  $r.\text{shape} == \text{DAG}$ . Later, at statement S3, the path due to  $f_{pq}$  is broken. Even though the inequality  $|I_F[r, q]| > 1$  still holds (because of  $\mathcal{U}$ ), we can still infer the shape transition from DAG to Tree because the boolean variable  $f_{pq}$  becomes false thus setting  $r_{\text{Dag}}$  to false.  $\square$

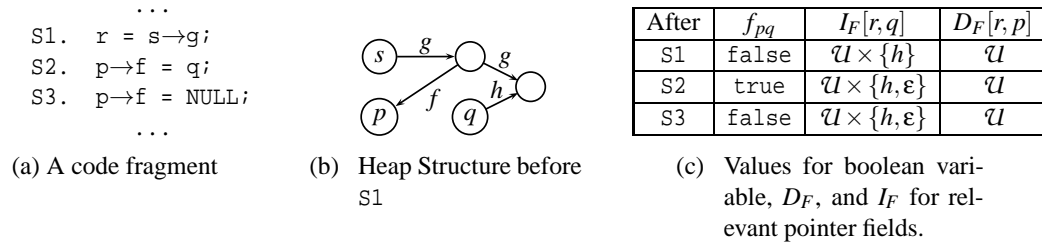


Figure 6.1: Using boolean variables to improve precision

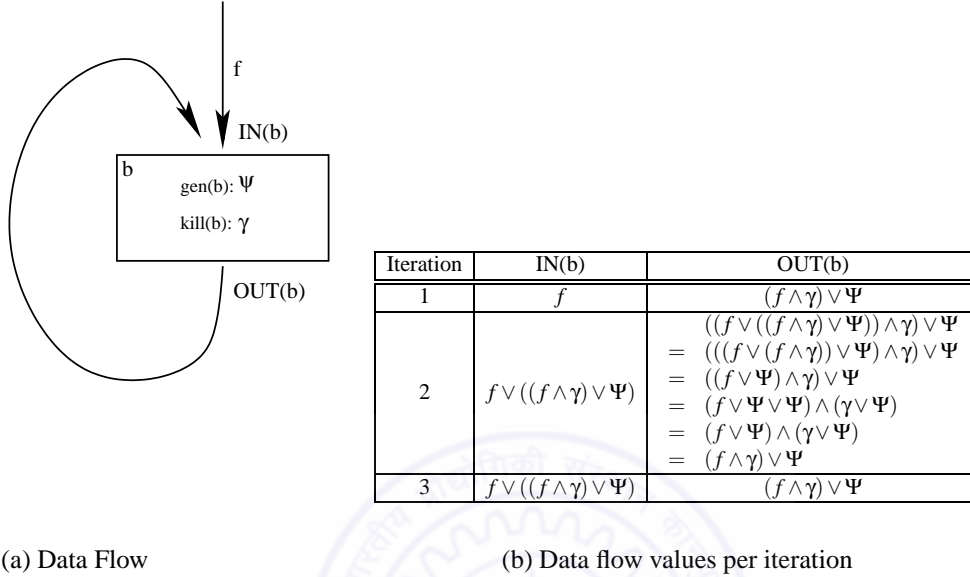


Figure 6.2: The termination of computation of boolean functions

## 6.2 Termination

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

Consider Fig. 6.2(a) which shows a basic block  $b$  with its in and out sets,  $IN(b)$  and  $OUT(b)$  respectively. The boolean formulas  $\Psi$  and  $\gamma$  respectively denote the gen and kill sets corresponding to basic block  $b$ . Let  $f$  denotes the boolean formula at a program point before executing  $b$ . Figure 6.2(b) shows the in and out sets generated in each iteration of the data flow analysis. As depicted in Fig. 6.2(b), the boolean functions acquire fixed-point after the second iteration.

## 6.3 Storage Requirement

The memory requirement of our analysis consists of the storage space for the matrices ( $D_F$  and  $I_F$ ) and the boolean functions. Let  $n \in \mathcal{N}$  denotes 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 pointer, for each field, we only use: (a) the ( $k$ -limited) count of the number of indirect paths starting at that given field, and (b) if there is a direct path using that field, the storage requirement for the matrices is bounded as:

$$\text{Space requirement for } D_F : O(n^2 * m)$$

$$\text{Space requirement for } I_F : O(n^2 * m^2)$$

The boolean functions at each program point are stored in an expression tree. 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. By carefully reusing the trees for the subexpressions in an expression, it is possible to store the boolean functions efficiently.





# Chapter 7

## Comparison with Other Approaches

For comparison purpose the test-cases must involve shape transitions like Cycle to DAG, Cycle to Tree, and DAG to Tree. The transition like Tree to DAG, DAG to Cycle, or Tree to Cycle are not of much importance as these can be detected by any of the field insensitive approaches. Following are the cases that meet our requirement and better demonstrate the accuracy of our analysis as compared to field insensitive analysis (like Ghiya et. al. [GH96]).

### 7.1 Comparison with Ghiya et. al. [GH96]

#### 7.1.1 Inserting an internal node in a singly linked list.

Consider the code fragment Fig. 7.1(a) that is a simplified version of insertion of an internal node in a linked list. Field insensitive approach like that of Ghiya et. al. [GH96] cannot detect the kill information due to the change of the field  $f$  of  $p$  at S4 and finds  $p$  to have an additional path to  $q$  via  $r$  (which is now actually the only path). So they report the shape attribute of  $p$  as DAG. Figure 7.1(c) shows the  $D_F$  and  $I_F$  matrices corresponding to our analysis at each program point. Consider the following boolean function generated after S4 using our approach.

$$\begin{aligned} p_{\text{Dag}} &= (f_{pq} \wedge (|I_F[pq]| > 1)) \vee (f_{pr} \wedge (|I_F[pr]| > 1)), \\ f_{pr} &= \mathbf{True}, & f_{pq} &= \mathbf{False} . \end{aligned}$$

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

#### 7.1.2 Swapping Two Nodes of a Singly Linked List.

Consider the code fragment Fig. 7.2(a) which swaps the two pointers  $p \rightarrow f$  (say  $n1$ ) and  $p \rightarrow f \rightarrow f$  (say  $n2$ ) in a singly linked list  $L$  with link field as  $f$ , given the pointer  $p$ . Also let  $t$  be the node following the heap object  $n2$  using  $f$  link. After S1, a temporary Cycle is created

After Stmt	Actual Shape	Field Insensitive Analysis	Field Sensitive Analysis
S1. $p \rightarrow f = q;$	Tree	Tree	Tree
S2. $r = \text{malloc}();$	Tree	Tree	Tree
S3. $r \rightarrow f = q;$	Tree	Tree	Tree
S4. $p \rightarrow f = r;$	Tree	DAG(at $p$ )	Tree

(a) A code fragment

(b) Shape Inference

After Stmt	$D_F$				$I_F$			
	$p$	$r$	$q$		$p$	$r$	$q$	
S1	$p$	0	0	$\{f^D\}$	$p$	0	0	$\{(f^D, \epsilon)\}$
	$r$	0	0	0	$r$	0	0	0
	$q$	0	0	0	$q$	$\{(\epsilon, f^D)\}$	0	0
S2	$p$	0	0	$\{f^D\}$	$p$	0	0	$\{(f^D, \epsilon)\}$
	$r$	0	$\{\epsilon\}$	0	$r$	0	$\{(\epsilon, \epsilon)\}$	0
	$q$	0	0	0	$q$	$\{(\epsilon, f^D)\}$	0	0
S3	$p$	0	0	$\{f^D\}$	$p$	0	0	$\{(f^D, \epsilon)\}$
	$r$	0	$\{\epsilon\}$	$\{f^D\}$	$r$	0	$\{(\epsilon, \epsilon)\}$	$\{(f^D, \epsilon)\}$
	$q$	0	0	0	$q$	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^D)\}$	0
S4	$p$	0	$\{f^D\}$	$\{f^D\}$	$p$	0	$\{(f^D, \epsilon)\}$	$\{(f^D, \epsilon)\}$
	$r$	0	$\{\epsilon\}$	$\{f^D\}$	$r$	$\{(\epsilon, f^D)\}$	$\{(\epsilon, \epsilon)\}$	$\{(f^D, \epsilon)\}$
	$q$	0	0	0	$q$	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^D)\}$	0

(c) Direction ( $D_F$ ) and Interference ( $I_F$ ) matrices

Figure 7.1: Insertion of an internal node in a singly linked list

on  $p, n1$  and  $n2$ , which get destroyed after the statement S2. This temporary shape transition is detected by our analysis. The table in Fig. 7.2(b) shows the comparison between the shape decision given by our approach and the field insensitive approaches. Consider the following boolean functions generated after S1 using our approach.

$$\begin{aligned}
 n2_{\text{Cycle}} &= (f_{n2, n1} \wedge |D_F[n1, n2]| \geq 1) \\
 n1_{\text{Cycle}} &= (f_{n2, n1} \wedge |D_F[n1, n2]| \geq 1) \\
 p_{\text{Cycle}} &= (|D_F[p, n2]| \geq 1 \wedge f_{n2, n1} \wedge |D_F[n1, n2]| \geq 1) \\
 p_{\text{Cycle}} &= (|D_F[p, n1]| \geq 1 \wedge f_{n2, n1} \wedge |D_F[n1, n2]| \geq 1) \\
 f_{n2, n1} &= \mathbf{True}
 \end{aligned}$$

As depicted in Fig. 7.2(c) which summarises the  $D_F$  and  $I_F$  matrices computed using our analysis, beyond S2 the condition  $|D_F[n1, n2]| \geq 1$  becomes **False** and thus the shape transition from Cycle to Tree is reported.

### 7.1.3 Recursively Swapping Binary Tree.

Consider the code fragment Fig. 7.3(a) which creates a mirror image of a binary tree rooted at  $T$ . While swapping the left and right sub-tree a temporary DAG is created (after statement S5), which gets destroyed after the very next statement S6. Consider the following boolean functions

After	Actual Shape	Field Insensitive Analysis	Field Sensitive Analysis
S1. $n2 \rightarrow f = n1;$	Cycle (at p, n1, n2)	Cycle (at p, n1, n2)	Cycle (at p, n1, n2)
S2. $n1 \rightarrow f = t;$	Tree	Cycle (at p, n1, n2)	Tree
S3. $p \rightarrow f = n2;$	Tree	Cycle (at p, n1, n2)	Tree

(a) A code fragment

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

(b) Shape Inference

After Stmt	$D_F$				$I_F$				
	n1	n2	p	t	n1	n2	p	t	
S1	n1	$\{f^{I1}\}$	$\{f^D, f^{I1}\}$	$\{f^{I1}\}$	n1	$\{(f^{I1}, \epsilon)\}$	$\{(f^D, \epsilon), (\epsilon, f^D)\}$	$\{(\epsilon, f^D)\}$	0
	n2	$\{f^D\}$	$\{f^{I1}\}$	0	n2	$\{(f^D, \epsilon), (\epsilon, f^D)\}$	$\{(f^{I1}, \epsilon)\}$	$\{(\epsilon, f^{I1})\}$	0
	p	$\{f^D, f^{I1}\}$	$\{f^{I2}\}$	0	p	$\{(f^D, \epsilon)\}$	$\{(f^{I1}, \epsilon)\}$	0	0
	t	0	0	0	t	0	0	0	0
S2	n1	0	0	$\{f^D\}$	n1	0	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^D)\}$	$\{(f^D, \epsilon)\}$
	n2	$\{f^D\}$	$\{f^{I1}\}$	0	n2	$\{(f^D, \epsilon)\}$	0	$\{(\epsilon, f^{I1})\}$	$\{(f^{I1}, \epsilon)\}$
	p	$\{f^D, f^{I1}\}$	$\{f^D, f^{I3}\}$	0	p	$\{(f^D, \epsilon)\}$	$\{(f^{I1}, \epsilon)\}$	0	$\{(f^{I1}, \epsilon)\}$
	t	0	0	0	t	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^{I1})\}$	$\{(\epsilon, f^{I1})\}$	0
S3	n1	0	0	$\{f^D\}$	n1	0	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^{I1})\}$	$\{(f^D, \epsilon)\}$
	n2	$\{f^D\}$	$\{f^{I1}\}$	0	n2	$\{(f^D, \epsilon)\}$	0	$\{(\epsilon, f^D)\}$	$\{(f^{I1}, \epsilon)\}$
	p	$\{f^{I1}\}$	$\{f^D, f^{I1}\}$	0	p	$\{(f^{I1}, \epsilon)\}$	$\{(f^D, \epsilon)\}$	0	$\{(f^{I1}, \epsilon)\}$
	t	0	0	0	t	$\{(\epsilon, f^D)\}$	$\{(\epsilon, f^{I1})\}$	$\{(\epsilon, f^{I1})\}$	0

(c) Direction ( $D_F$ ) and Interference ( $I_F$ ) matrices

Figure 7.2: Swapping two nodes of a singly linked list

generated after S5 using our approach.

$$T_{\text{Dag}} = (\text{left}_{T,R} \wedge |I_F[T,R]| > 1)$$

$$\text{left}_{T,R} = \mathbf{True}$$

Figure 7.3(c) shows the  $D_F$  and  $I_F$  matrices corresponding to our analysis at each program point. Due to the inclusion of the approximation ( $\mathcal{U}$ ) for the statements S1 and S2, the condition  $|I_F[T,R]| > 1$  still holds beyond S6 and thus the analysis reports shape of  $T$  as DAG, which is actually a Tree. Considering the fact that shape attribute of  $T$  is a Tree, and thus statements S1 and S2 must include just one path between  $T$  and  $L$  or  $T$  and  $R$ , we obtain Figure 7.3(d) which shows the refined  $D_F$  and  $I_F$  matrices at each program point, using which we can infer the shape of  $T$  after S6 as Tree because the condition  $|I_F[T,R]| > 1$  evaluated to **False** after S6. Figure 7.3(b) shows the shape transition at each program point using this refined analysis.

## 7.2 Comparison with Marron et. al. [MKSH06]

Using the property as mentioned in Appendix B, we inferred the shape of the data structure corresponding to each region for the test-cases that we have chosen, and concluded that their shape inferencing comply exactly with the actual shapes that should be reported.

To eliminate the state explosion that is possible with refinement, they apply refinement to only those cases where there exist a unique way of materializing new nodes. This limits the amount of precision that can be achieved as there exists cases where refinement into multiple



```

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

```

(a) A code fragment

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

(b) Shape Inference

After Stmt	$D_F$			$I_F$				
	T	L	R	T	L	R		
S1	T	$\emptyset$	$\{l^D\} \cup \mathcal{U}$	$\emptyset$	T	$\emptyset$	$\{l^D\} \cup \mathcal{U} \times \{\epsilon\}$	$\emptyset$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon\} \times (\{l^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\emptyset$	$\emptyset$	$\emptyset$
S2	T	$\emptyset$	$\{l^D\} \cup \mathcal{U}$	$\{r^D\} \cup \mathcal{U}$	T	$\emptyset$	$(\{l^D\} \cup \mathcal{U}) \times \{\epsilon\}$	$(\{r^D\} \cup \mathcal{U}) \times \{\epsilon\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon\} \times (\{r^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon\} \times (\{r^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$
S5	T	$\emptyset$	$\mathcal{U}$	$\{r^D, l^D\} \cup \mathcal{U}$	T	$\emptyset$	$\mathcal{U} \times \{\epsilon\}$	$(\{r^D, l^D\} \cup \mathcal{U}) \times \{\epsilon\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon\} \times \mathcal{U}$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon\} \times (\{r^D, l^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$
S6	T	$\emptyset$	$\{r^D\} \cup \mathcal{U}$	$\{l^D\} \cup \mathcal{U}$	T	$\emptyset$	$(\{r^D\} \cup \mathcal{U}) \times \{\epsilon\}$	$(\{l^D\} \cup \mathcal{U}) \times \{\epsilon\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon\} \times (\{r^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon\} \times (\{l^D\} \cup \mathcal{U})$	$\emptyset$	$\emptyset$

(c) Direction ( $D_F$ ) and Interference ( $I_F$ ) matrices

After Stmt	$D_F$			$I_F$				
	T	L	R	T	L	R		
S1	T	$\emptyset$	$\{l^D\}$	$\emptyset$	$\{l^D, \epsilon\}$	$\emptyset$		
	L	$\emptyset$	$\emptyset$	$\emptyset$	$\{\epsilon, l^D\}$	$\emptyset$		
	R	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$		
S2	T	$\emptyset$	$\{l^D\}$	$\{r^D\}$	T	$\emptyset$	$\{l^D, \epsilon\}$	$\{r^D, \epsilon\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon, l^D\}$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon, r^D\}$	$\emptyset$	$\emptyset$
S5	T	$\emptyset$	$\emptyset$	$\{r^D, l^D\}$	T	$\emptyset$	$\emptyset$	$\{(r^D, \epsilon), (l^D, \epsilon)\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\emptyset$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon, r^D\}, \{\epsilon, l^D\}$	$\emptyset$	$\emptyset$
S6	T	$\emptyset$	$\{r^D\}$	$\{l^D\}$	T	$\emptyset$	$\{(r^D, \epsilon)\}$	$\{(l^D, \epsilon)\}$
	L	$\emptyset$	$\emptyset$	$\emptyset$	L	$\{\epsilon, r^D\}$	$\emptyset$	$\emptyset$
	R	$\emptyset$	$\emptyset$	$\emptyset$	R	$\{\epsilon, l^D\}$	$\emptyset$	$\emptyset$

(d) Refined Direction ( $D_F$ ) and Interference ( $I_F$ ) matrices

Figure 7.3: Computing mirror image of a binary tree. “ $l$ ” and “ $r$ ” denotes respectively the left and right fields of tree.

possibilities is needed to get results with the desired accuracy.





# Chapter 8

## Conclusion and Future Work

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

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



# Appendix A

## Analysis of Ghiya et. al. [GH96]<sup>1</sup>

Most of the definitions and technical terms used in this chapter are borrowed from the aforementioned paper. The proposed shape analysis composed of three store-less abstractions that are computed together at each program point. For each heap directed pointer they approximated the attribute shape and for each pair of heap directed pointers they approximated the direction and interference relationships between them. These three abstractions are defined formally as follows:

**Definition A.1.** *Given any heap-directed pointer  $p$ , the shape attribute  $p.shape$  is Tree, if in the data structure accessible from  $p$  there is a unique (possibly empty) access path between any two nodes (heap objects) belonging to it. It is considered to be DAG (directed acyclic graph), if there can be more than one path between any two nodes in this data structure, but there is no path from a node to itself (i. e, it is acyclic). If the data structure contains a node having a path to itself,  $p.shape$  is considered to be Cycle. Note that as lists are special case of tree data structures, their shape is also considered as Tree.*

**Definition A.2.** *Given two heap directed pointers  $p$  and  $q$ , the direction matrix  $D$  captures the following relationships between them:*

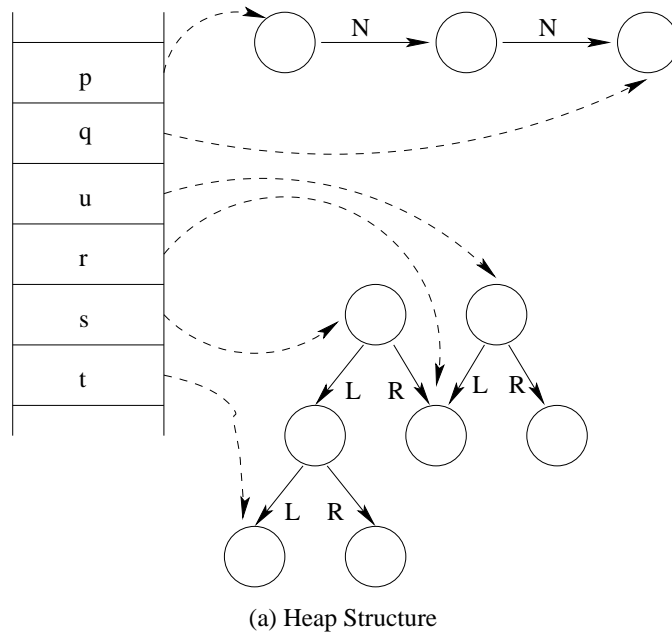
- $D[p, q] = 1$  : *An access path possibly exists in the heap, from the heap object pointed to by  $p$ , to the heap object pointed to by  $q$ . In this case we simply say that the pointer  $p$  has a path to pointer  $q$ .*
- $D[p, q] = 0$  : *No access path exists from the heap object pointed to by  $p$  to the heap object pointed to by  $q$ .*

**Definition A.3.** *Given two heap directed pointers  $p$  and  $q$ , the direction matrix  $I$  captures the following relationships between them:*

- $I[p, q] = 1$  : *A common heap object can be possibly accessed starting from pointers  $p$  and  $q$ . In this case we state that pointers  $p$  and  $q$  can interfere.*

---

<sup>1</sup>The contents of this section are borrowed from [GH96]



$D$	$p$	$q$	$r$	$s$	$t$	$u$
$p$	1	1	0	0	0	0
$q$	0	1	0	0	0	0
$r$	0	0	1	0	0	0
$s$	0	0	1	1	1	0
$t$	0	0	0	0	1	0
$u$	0	0	1	0	0	1

(b) Direction Matrix

$I$	$p$	$q$	$r$	$s$	$t$	$u$
$p$	1	1	0	0	0	0
$q$	1	1	0	0	0	0
$r$	0	0	1	1	0	1
$s$	0	0	1	1	1	1
$t$	0	0	0	1	1	0
$u$	0	0	1	1	0	1

(c) Interference Matrix

Figure A.1: Example Direction and Interference Matrices

- $I[p, q] = 0$  : No common heap object can be accessed starting from pointers  $p$  and  $q$ . In this case we state that pointers  $p$  and  $q$  do not interfere.

Direction relationships are used to actually estimate the shape attributes, where the interference relationships are used for safely calculating direction relationships.

## Illustrative Example

The direction and interference matrices are illustrated in Fig. A.1. Part (a) represents a heap structures at a program point, while parts (b) and (c) show the direction and interference matrices for it.

We now demonstrate how direction relationships help estimate the shape of the data structures. In Fig. A.2, initially we have both  $p.shape$  and  $q.shape$  as Tree. Further  $D[q, p] = 1$ , as there exists a path from  $q$  to  $p$  through next link. The statement  $p \rightarrow prev = q$ , sets up a path from  $p$  to  $q$  through the `prev` link. From direction matrix information we already know that a path exists from  $q$  to  $p$ , and now a path is being set from  $p$  to  $q$ . Thus after the statement,  $D[p, q] = 1$ ,  $D[q, p] = 1$ ,  $p.shape = Cycle$  and  $q.shape = Cycle$ .

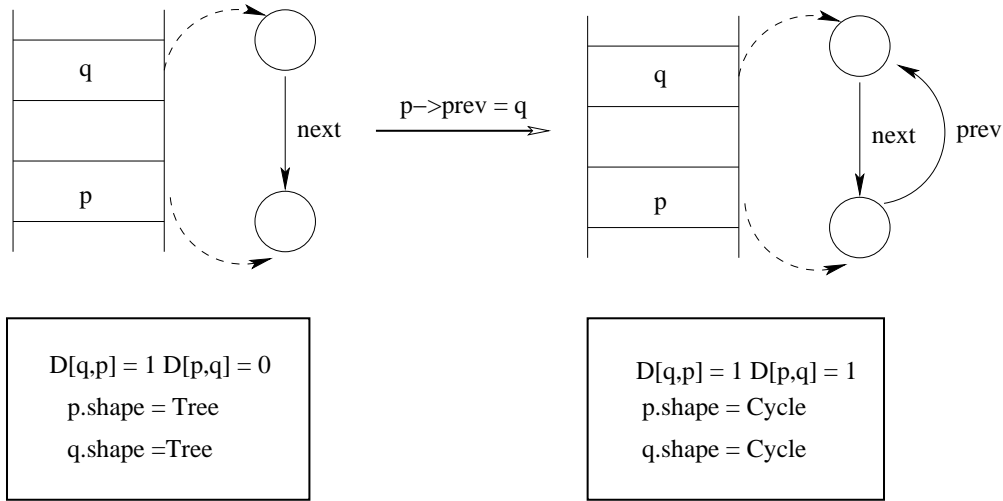


Figure A.2: Example Demonstrating Shape Estimation

### Analysis of Basic Statements

They have considered eight basic statements that can access or modify heap data structures as listed in Fig. A.3(a). Variables  $p$  and  $q$  and the field  $f$  are of pointer type, variable  $k$  is of integer type, and  $op$  denotes the  $+$  and  $-$  operations. The overall structure of the analysis is shown in Fig. A.3(b). Given the direction and the interference matrices  $D$  and  $I$  at a program point  $x$ , before the given statement, they compute the matrices  $D_n$  and  $I_n$  at a program point  $y$ . Additionally, we have the attribute matrix  $A$ , where for a pointer  $p$ ,  $A[p]$  gives its shape attribute. The attribute matrix after the statement is presented as  $A_n$ .

For each statement they compute the set of direction and interference relationships it kills and generates. Using these sets, the new matrices  $D_n$  and  $I_n$  are computed as shown in Fig. A.3(c). Note that the elements in the gen and kill sets are denoted as  $D[p, q]$  for direction relationships, and  $I[p, q]$  for interference relationships. Thus a gen set of the form  $\{D[x, y], D[y, z]\}$ , indicates that the corresponding entries in the output direction matrix  $D_n[x, y]$  and  $D_n[y, z]$  should be set to one. We also compute the set of pointers  $H_s$ , whose shape attribute can be modified by the given statement. Another attribute matrix  $A_c$  is used to store the changed attribute of pointers belonging to the set  $H_s$ . The attribute matrix  $A_n$  is then computed using the matrices  $A$  and  $A_c$  as shown in Fig. A.3(c).

Let  $H$  be the set of pointers whose relationships/attributes are abstracted by the matrices  $D$ ,  $I$  and  $A$ . Further assume that updating an interference matrix entry  $I[q, p]$ , implies identically updating the entry  $I[p, q]$ .

The actual analysis rules can be divided into three groups: (1) allocations, (2) pointer assignments, and (3) structure updates. Figure A.4 shows the gen and kill sets corresponding to each statement.



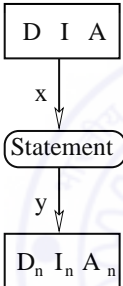
<p><b>Allocation</b></p> <ol style="list-style-type: none"> <li>1. <math>p = \text{malloc}();</math></li> </ol> <p><b>Pointer Assignments</b></p> <ol style="list-style-type: none"> <li>2. <math>p = q;</math></li> <li>3. <math>p = \&amp;(q \rightarrow f);</math></li> <li>4. <math>p = q \text{ op } k;</math></li> <li>5. <math>p = \text{NULL};</math></li> <li>6. <math>p = q \rightarrow f;</math></li> </ol> <p><b>Structure Updates</b></p> <ol style="list-style-type: none"> <li>7. <math>p \rightarrow f = q;</math></li> <li>8. <math>p \rightarrow f = \text{NULL};</math></li> </ol> <p>(a) Basic statements</p>	 <p>(b) Analysis Structure</p>	<p><b>Build the new matrices</b></p> $\forall r, s \in H, \quad D_n[r, s] = D[r, s], \quad I_n[r, s] = I[r, s]$ $\forall s \in H, \quad A_n[s] = A[s]$ <p><b>Delete Killed relationships</b></p> $\forall \text{entries } D[r, s] \in D_{\text{kill\_set}}, \quad D_n[r, s] = 0$ $\forall \text{entries } I[r, s] \in I_{\text{kill\_set}}, \quad I_n[r, s] = 0$ <p><b>Add generated relationships</b></p> $\forall \text{entries } D[r, s] \in D_{\text{gen\_set}}, \quad D_n[r, s] = 1$ $\forall \text{entries } I[r, s] \in I_{\text{gen\_set}}, \quad I_n[r, s] = 1$ <p><b>Update shape attributes of affected pointers</b></p> <p><b>Compute <math>H_s</math> and <math>A_s</math></b></p> $\forall s \in H_s, \quad A_n[s] = A[s]$ <p>(c) General Form of Analysis Rules</p>
---	--	--

Figure A.3: The Overall Structure of the Analysis

1. $p = \text{malloc}();$	$D\_kill\_set = \{D[p,s]   s \in H \wedge D[p,s]\} \cup \{D[s,p]   s \in H \wedge D[s,p]\}$ $I\_kill\_set = \{I[p,s]   s \in H \wedge I[p,s]\}$ $D\_gen\_set = \{D[p,p]\} \quad I\_gen\_set = \{I[p,p]\}$ $H_s = \{p\} \quad A_c[p] = \text{Tree}$
2. $p = q;$ 3. $p = \&(q \rightarrow f);$ 4. $p = q \text{ op } k;$	<p>Kill set same as that of <math>p = \text{malloc}();</math></p> $D\_gen\_set\_from = \{D[s,p]   s \in H \wedge s \neq p \wedge D[s,q]\}$ $D\_gen\_set\_to = \{D[p,s]   s \in H \wedge s \neq p \wedge D[q,s]\}$ $I\_gen\_set = \{I[p,s]   s \in H \wedge s \neq p \wedge I[q,s]\} \cup \{I[p,p]   I[q,q]\}$ $D\_gen\_set = D\_gen\_set\_from \cup D\_gen\_set\_to$ $H_s = \{p\} \quad A_c[p] = A[q]$
5. $p = \text{NULL};$	<p>Kill set same as that of <math>p = \text{malloc}();</math></p> $D\_gen\_set = \{\} \quad I\_gen\_set = \{\}$ $H_s = \{p\} \quad A_c[p] = \text{Tree}$
6. $p = q \rightarrow f;$	<p>Kill set same as that of <math>p = \text{malloc}();</math></p> $D\_gen\_set\_from = \{D[s,p]   s \in H \wedge s \neq p \wedge I[s,q]\}$ $D\_gen\_set\_to = \{D[p,s]   s \in H \wedge s \neq p \wedge s \neq q \wedge D[q,s]\} \cup \{D[p,q]   A[q] = \text{Cycle}\} \cup \{D[p,p]   D[q,q]\}$ $D\_gen\_set = D\_gen\_set\_from \cup D\_gen\_set\_to$ $I\_gen\_set = \{I[p,s]   s \in H \wedge s \neq p \wedge I[q,s]\} \cup \{I[p,p]   I[q,q]\}$ $A_c[p] = A[q]$
7. $p \rightarrow f = \text{NULL};$	$D\_kill\_set = \{\} \quad I\_kill\_set = \{\}$ $D\_gen\_set = \{\} \quad I\_gen\_set = \{\}$ $A_c[p] = A[p] \quad \forall p \in H$
7. $p \rightarrow f = q;$	<p>Kill set same as that of <math>p \rightarrow f = \text{NULL};</math></p> $D\_gen\_set = \{D[r,s]   r,s \in H \wedge D[r,p] \wedge D[q,s]\}$ $I\_gen\_set = \{I[r,s]   r,s \in H \wedge D[r,p] \wedge I[q,s]\}$ <p>Pointer <math>q</math> already has a path to <math>p</math>, <math>D[q,p] = 1</math></p> $\overline{H_s} = \{s   s \in H \wedge (D[s,p] \vee D[s,q])\}$ $D[q,p] \Rightarrow A_c[s] = \text{Cycle} \quad \forall s \in H_s$ <p><math>A[q] = \text{Tree}</math></p> $\overline{H_s} = \{s   s \in H \wedge (D[s,p] \vee I[s,q])\}$ $(\neg D[q,p] \wedge (A[q] = \text{Tree})) \Rightarrow A_c[s] = A[s] \bowtie \text{Dag} \quad \forall s \in H_s$ <p><math>A[q] \neq \text{Tree}</math></p> $\overline{H_s} = \{s   s \in H \wedge D[s,p]\}$ $(\neg D[q,p] \wedge (A[q] \neq \text{Tree})) \Rightarrow A_c[s] = A[s] \bowtie A[q] \quad \forall s \in H_s$

Figure A.4: Analysis Rules



# Appendix B

## Analysis of Marron et. al. [MKSH06]<sup>1</sup>

Most of the technical terms used in this chapter are borrowed from the aforementioned paper. The proposed analysis followed the abstract heap graph model that uses nodes to represent sets of concrete cells (heap allocated objects and arrays) and edges to represent sets of pointers. Each node in the abstract heap graph can be viewed as a region in memory on which certain layout predicates can be defined (Tree Layout, List Layout, Singleton Layout, Multi-Path or Cycle Layouts) which signifies what types traversal patterns a program can use to navigate through the data structures in the region. To track the concrete Structure Layout, they introduce a simple domain of layout types = {Singleton, List, Tree, Multi-Path, Cycle}. The abstract layouts can be given a simple total order: Singleton < List < Tree < Multi-Path < Cycle. This order can be interpreted as: if a node  $n$  has abstract layout  $\zeta$  then the concrete region,  $\mathcal{R} = \gamma(n)$ , where  $\gamma$  is the concretization operator, may have any of the layout properties less than or equal to  $\zeta$ . For example, if we have a node with layout type List the concrete region may have the List or Singleton layout properties. If the node has Cycle as the layout then the concrete domain may have any of the layout properties. The abstract layout for a node  $n$  represents the most general concrete layout that may be encountered by a program traversing the region that is represented by the node  $n$ .

In their analysis, sometimes refinement is necessary (after summarization the abstract heap graph) whose purpose is to transform summary representations into forms that make certain relationships explicit, so that the information in these relationships can be utilized more easily. During refinement they turn summary nodes into a number of nodes of size one so that strong updates can be performed and exact relations between variables can be maintained.

In order to eliminate the state explosion that is possible with refinement, they adopt the approach of only doing refinement in those cases in which we can be sure that there is a unique way in which new nodes can be materialized. This limits the level of details that can be achieved, but it is easy to demonstrate scenarios where refinement into multiple possibilities is needed to get results with the desired accuracy.

---

<sup>1</sup>The contents of this section are borrowed from [MKSH06]



# References

- [BCC<sup>+</sup>07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O’hearn, Hongseok Yang, and Queen Mary. Shape analysis for composite data structures. In *CAV ’07*, pages 178–192, 2007.
- [CR07] Sigmund Cherem and Radu Rugina. Maintaining doubly-linked list invariants in shape analysis with local reasoning. *VMCAI’07*, pages 234–250, 2007.
- [CRB10] Renato Cherini, Lucas Rearte, and Javier Blanco. A shape analysis for non-linear data structures. *SAS’10*, pages 201–217, 2010.
- [CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *PLDI ’90*, pages 296–310, 1990.
- [DOY06] Dino Distefano, Peter O’hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920, pages 287–302. 2006.
- [GH96] Rakesh Ghiya and Laurie J. Hendren. 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*, pages 1–15, January 1996.
- [GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. *POPL ’98*, pages 121–133, 1998.
- [Ghi96] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Technical report, Master’s thesis, McGill U, 1996.
- [GHZ98] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in c programs with recursive data structures. In *7th International Conference on Compiler Construction*, pages 159–173, 1998.
- [HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. *POPL ’05*, pages 310–323, 2005.

- [JM79] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. *POPL '79*, pages 244–256, 1979.
- [JM09] Maria Jump and Kathryn S. McKinley. Dynamic shape analysis via degree metrics. In *Proceedings of the 2009 international symposium on Memory management*, pages 119–128, 2009.
- [LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *PLDI '88*, pages 24–31, 1988.
- [MKSH06] Mark Marron, Deepak Kapur, Darko Stefanovic, and Manuel Hermenegildo. A static heap analysis for shape and connectivity: unified memory analysis: the base framework. *LCPC'06*, pages 345–363, 2006.
- [SRW96] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *POPL '96*, pages 16–31, 1996.
- [SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *POPL '99*, pages 105–118, 1999.
- [SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
- [SYKS03] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Shmuel Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. *SAS '03*, pages 483–503, 2003.