FIELD SENSITIVE SHAPE ANALYSIS: IMPLEMENTATION AND IMPROVEMENTS

by

P.Vinay Kumar Reddy

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

FIELD SENSITIVE SHAPE ANALYSIS: IMPLEMENTATION AND IMPROVEMENTS

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology (M.Tech)

by

P.Vinay Kumar Reddy

Supervised By

Dr. Amey Karkare

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

June, 2012

CERTIFICATE

It is certified that the work contained in this thesis entitled

"*Field Sensitive Shape Analysis: Implementation and Improvements*",

by *P.Vinay Kumar Reddy(Roll No. 10111026)*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare 19/6/2012

(Dr. Amey Karkare)

Department of Computer Science and Engineering,

Indian Institute of Technology Kanpur

Kanpur-208016

June, 2012

ACKNOWLEDGMENTS

I take this opportunity to thank my thesis guide **Dr. Amey Karkare** for his invaluable support and guidance. He has been utmost patient and helped me throughout in completion of my work. I express my gratitude to the faculty and staff of the Department of CSE for the beautiful academic environment they have created here.

I would like to thank **Mr. Sandeep Dasgupta** for his immense support and invaluable suggestions that I have received from him.

I would also like to thank all my friends of Mtech 2010 Batch and especially Anvesh, Ashendra, Chitti Babu, Keerthi Kumar and Saravana for making my stay at IIT Kanpur truly special.

Finally, I wish to extend my thanks to my parents, my brother for their support and encouragement. 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

P. Vinay Kumar Reddy

Dedicated to

my parents,my brother and my friends.

Abstract

Shape Analysis refers to a class of techniques used to analyze heap data structures. Several algorithms have been proposed in literature about shape analysis. These algorithms differ in the trade off they have between speed and accuracy.

In this thesis we have implemented and proposed enhancements for a field sensitive shape analysis approach. These enhancements involve modification of data flow values, and intelligent way of storing the same which makes the analysis more precise and memory efficient. This work also proposes an analysis namely *Subset Based Analysis* which infers more precise shapes depending upon which field pointers are actually accessed in a function. To handle functions we have developed a new method for interprocedural analysis called *Shape Sensitive Analysis*. This is a middle way between Context Sensitive and Context Insensitive Interprocedural Analysis. We have implemented this analysis as a plugin for gcc 4.5.0 and the results for the same are presented.

Contents

List of Figures

Chapter 1

Introduction

1.1 Brief Introduction

Shape Analysis is a static analysis technique which works on the heap to find the possible shape of the heap allocated objects. This is useful in many areas like garbage collection, parallelization, compile time optimization, instruction scheduling etc.

In this report we discuss in detail the effectiveness of the work of Sandeep [DK12] about field sensitive shape analysis. [DK12] shows its preciseness using few typical examples, but lacks deep evaluation of the approach in order to check its scalability and preciseness on large practical benchmark programs. The present work suggests several enhancements over [DK12], involving modifications of the data flow values and intelligent ways of storing the same, which makes the analysis more precise and memory efficient. This work also involves implementing the interprocedural version of the analysis as a dynamic plugin on GCC and deep evaluation of the same using large benchmarks.

This work also proposes a refined version of analysis namely subset-based field sensitive analysis (refer Chapter 4) which infers more precise shapes depending upon which field pointers are actually accessed in a function.

The interprocedural analysis is provided with two flavors: context sensitive and context insensitive; a trade-off analysis on which of these techniques would be good for this shape analysis technique is described. Also a new method of merging contexts at function calls is proposed whose complexity lies middle way between the above two, namely shape sensitive interprocedural analysis, which uses the shape information of the arguments of function to decide which context to merge with.

1.2 Organization Of Thesis

We discuss some of the prior works on shape analysis in Chapter 2. Chapter 3 discusses all the enhancements suggested to [DK12]. The Subset-based analysis is detailed in Chapter 4. Chapter 5 elaborates on the Interprocedural analysis along with its pro-

posed flavours. Implementation details along with detailed evaluation using benchmark programs are given in Chapter 6. We conclude the presentation in Chapter 7 and give directions for future work.

Chapter 2

Background

2.1 Related Work

It was for functional languages that first shape analysis was looked at. Jones and Muchnick [JM79] suggested a method for finding shape of unbounded data objects in LISP like languages using regular tree grammars. They associate with each program point a set of shape graphs and to handle termination of analysis they use k-limiting approach. So they treat all nodes whose distance is more than k from root as a single summarized node. Due to its large consumption of space and time the analysis is not practical.

Chase et al. [CWZ90] has used the concept of heap reference counts. They associate each node with a reference count and then try to find that part of the heap where all nodes have reference count as one, such portions are said to be a tree or list. They have tried to tackle the problems with k-limiting to some extent, however their work fails obtaining accurate results for recursive data structures.

Sagiv et al. [SRW99], [SRW02] have presented a family of abstract interpretation algorithms based on three-valued logic. They use abstraction, a method for summarizing node and to handle destructive updates they have come up with re-materialization which refers to the process of splitting summary nodes. An exponential number of shape nodes may arise because of abstract interpretation, so its not suitable for practical purposes. Sagiv and Noam [RS01] have also looked into inter procedural shape analysis for recursive programs but they work only on linked lists.

The main idea of Brian et al. [HR05] is to decompose heap abstractions and independently analyze different parts of the heap. They also decompose memory abstraction horizontally and vertically. Now for the local work to propagate globally they proposed and used a context sensitive inter procedural shape analysis algorithm.

A dynamic shape analysis technique was proposed by Jump et al. [JM09]. They compute a class field summary graph that summarizes the dynamic object graph. This summary graph also records the in-degree and out-degree of each object which are the recursive degree metrics. In their analysis they keep track of those node which are of

fixed degree and also those whose degree is in a particular range. Since running the analysis after each pointer statement is very costly they do it by piggybacking with garbage collection.

Susan et al. [HRS95] have a polynomial worst case method for inter procedural analysis provided its a demand data flow analysis. It will determine whether a single given data flow value holds at some give point. But the class of problems it can handle was limited. Alexy [GBC06] represent heap portions independently by using the notions of abstraction and separation logic. Their representation helps to easily separate the portion which is reachable and which is not from a procedure. Its limitation is that it supports only linked lists, doubly linked lists and trees.

Ghiya et al. [GH96] estimates the shapes of heap structures pointed to by pointers as a *Tree*, *Dag* or *Cycle*. They use Direction, Interference matrices and shape attributes as their data flow values which gets generated and killed after each pointer statements. As this is very closely related to our work, we have given a detailed view of it in the Appendix section.

Marron et al. [MKSH06] uses a graph based heap model with objects being vertices and pointers being the labeled edges. A node is considered as a set of cells and each node is associated with a layout saying *Singleton*, *List*, *Tree*, *Multipath* or *Cycle*. These layouts honors the order $Singleton < List < Tree < Multipath < Cycle$. This means that suppose a node is of layout *Tree* then it may have properties of *Singleton*, *List* or *Tree*. They have methods by which summarized nodes are split to concrete nodes (and edges) but its only for for the most common cases encountered, this enables them to handle strong updates. Then they have proposed a context sensitive analysis [MHKS08] for the same graph based heap models. For this they have come up with operations $project/extend$. *project* removes that part of heap which is unaffected by a called procedure and *extend* rejoins the unreachable portion back after return.

The work of Sandeep et al. [Das11], which is extended in the present paper, is explained in detail in the following section for a better understanding of the basis of this work.

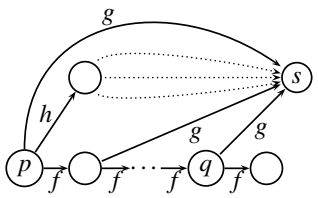
2.2 Analysis of Sandeep et. al. [Das11][1]

Most of the definitions and technical terms used in this section are borrowed from the aforementioned paper. As we will be using these details so throughout the report we have mentioned this as a separate section in our report. They have presented a shape analysis technique that uses limited field sensitivity to infer the shape. As this technique is able to handle destructive updates, so precise shape information can be obtained. They generated data flow values in the form of field sensitive matrices and boolean equations at each

[1]The contents of this section are borrowed from [Das11]

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

(a) A code fragment



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

Figure 2.1: Paths in a heap graph

program point to obtain the shape information.

2.2.1 Definitions And Notations

At a particular program point, the heap structure is viewed 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 all the relevant pointer variables are shown that can point to the heap object corresponding to that node. The edges are labeled by the name of the corresponding pointer field.

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, only the first field of a path is stored. 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, the superscript D for a direct path and I for an indirect path are used. In pictures, solid edges are used 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, first fields of paths are stored, including the count for the indirect paths, between two pointer variables in a set. To bound the size of the set, a limit k is put on number of repetitions of a particular field. If the number goes beyond k, the number of paths with that field is treated as ∞.

Example 1. Figure 2.1(a) shows a code fragment and Fig. 2.1(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 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, f^* is used for the cases when it is irrelevant to distinguish between direct or indirect path starting at the first field f. Next field sensitive matrices are defined.

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

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

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

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

The partial order is extended to set of paths S_{P_1}, S_{P_2} as[2]:

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

For pair of paths:

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

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

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

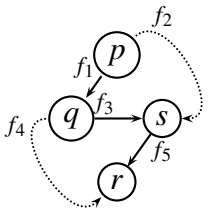
Two pointers $p,q \in \mathcal{H}$ are said to interfere if there exists $s \in \mathcal{H}$ such that both p and q have paths reaching s. Note that s could be p (or q) itself, in which case the path from p (from q) is ε.

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

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

The analysis computes over-approximations for the matrices D_F and I_F at each pro-

[2]Note that for the 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



(a) Heap graph

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

(b) Direction Matrix

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

(c) Interference Matrix

Figure 2.2: A heap graph and its field sensitive path matrices

gram 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, the above relation is used to show the computation of only one of the two entries.

Example 2. Figure 2.2 shows a heap graph and the corresponding field sensitive matrices as computed by the analysis. □

As mentioned earlier, for each variable $p \in \mathcal{H}$, the analysis uses attributes $p_{\texttt{Dag}}$ and $p_{\texttt{Cycle}}$ to store boolean functions telling whether p can reach a DAG or cycle respectively in the heap. The boolean functions consist of the values from matrices D_F, I_F, and the field connectivity information. For $f \in \mathcal{F}, p,q \in \mathcal{H}$, field connectivity is captured by boolean variables of the form f_{pq}, which is true when f field of p points directly to q. The shape of p, p.shape, can be obtained by evaluating the functions for the attributes $p_{\texttt{Cycle}}$ and $p_{\texttt{Dag}}$, and using Table 2.1.

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

Table 2.1: Determining shape from boolean attributes

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

- 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 :

$$|\varepsilon| = 1, \qquad |f^D| = 1, \qquad |f^{I\infty}| = \infty, \qquad |f^{Ij}| = j \text{ for } j \in \mathcal{N}$$
$$|S| = \sum_{\alpha \in S} |\alpha|$$

Also,

$$|(\alpha, f^{Im})| = \begin{cases} m & \text{if } \alpha \in \{f^D, \varepsilon\} \\ m*n & \text{if } \alpha = f^{In} \\ \infty & \text{if } \alpha = f^{I\infty} \end{cases}$$

$$|(f^{Im}, \beta)| = \begin{cases} m & \text{if } \beta \in \{f^D, \varepsilon\} \\ m*n & \text{if } \beta = f^{In} \\ \infty & \text{if } \beta = f^{I\infty} \end{cases}$$

$$|(\alpha, \beta)| = 1 \quad \text{if } \alpha, \beta \in \{f^D, \varepsilon\}$$
$$|(f^{I\infty}, \beta)| = \infty \quad \text{where } \beta \in \{f^D, \varepsilon, f^{I\infty}\}$$
$$|(\alpha, f^{I\infty})| = \infty \quad \text{where } \alpha \in \{f^D, \varepsilon, f^{I\infty}\}$$
$$|P| = \sum_{(\alpha,\beta) \in P} |(\alpha, \beta)|$$

- 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 2.2. 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 2.2: Path removal, intersection and union operations, where γ denotes any other path.

(a) Path removal

$\ominus$ $\{\beta\}$ / $\{\alpha\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	$\{\gamma\}$
$\{\varepsilon\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon\}$
$\{f^D\}$	$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\{f^D\}$	$\{f^D\}$
$\{f^{Ii}\}$	$\{f^{Ii}\}$	$\emptyset$	$\{f^{Im}\}$	$\emptyset$	$\{f^{Ii}\}$
$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\emptyset$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$

(b) Intersection

$\cap$ $\{\beta\}$ / $\{\alpha\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	$\{\gamma\}$
$\{\varepsilon\}$	ε	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^D\}$	$\emptyset$	$\{f^D\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{f^{Ii}\}$	$\emptyset$	$\emptyset$	$\{f^{In}\}$	$\{f^{Ii}\}$	$\emptyset$
$\{f^{I\infty}\}$	$\emptyset$	$\emptyset$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	$\emptyset$

(c) Union

$\cup$ $\{\beta\}$ / $\{\alpha\}$	$\{\varepsilon\}$	$\{f^D\}$	$\{f^{Ij}\}$	$\{f^{I\infty}\}$	$\{\gamma\}$
$\{\varepsilon\}$	$\{\varepsilon\}$	$\{\varepsilon, f^D\}$	$\{\varepsilon, f^{Ij}\}$	$\{\varepsilon, f^{I\infty}\}$	$\{\varepsilon,\gamma\}$
$\{f^D\}$	$\{f^D,\varepsilon\}$	$\{f^D\}$	$\{f^D, f^{Ij}\}$	$\{f^D, f^{I\infty}\}$	$\{f^D,\gamma\}$
$\{f^{Ii}\}$	$\{f^{Ii},\varepsilon\}$	$\{f^{Ii}, f^D\}$	$\{f^{It}\}$	$\{f^{I\infty}\}$	$\{f^{Ii},\gamma\}$
$\{f^{I\infty}\}$	$\{f^{I\infty},\varepsilon\}$	$\{f^{I\infty}, f^D\}$	$\{f^{I\infty}\}$	$\{f^{I\infty}\}$	$\{f^{I\infty},\gamma\}$

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

Table 2.3: Multiplication by a scalar

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

- Path removal, intersection and union over set of pair of paths : For singleton sets of paths $\{\alpha,\beta\}$ and $\{\gamma,\delta\}$, union($\{\alpha,\beta\} \cup \{\gamma,\delta\}$), intersection ($\{\alpha,\beta\} \cap \{\gamma,\delta\}$) and path removal ($\{\alpha,\beta\} \ominus \{\gamma,\delta\}$) operations are defined as given in Figure 2.3, 2.4 and 2.5 respectively. As before these definitions can be extended to set of pair of paths in a natural way. For example, for general sets of paths, P_1 and P_2, the definition of removal can be extended as:

$$P_1 \ominus P_2 = \bigcap_{(\gamma,\delta)\in P_2} \Big(\bigcup_{(\alpha,\beta)\in P_1} \{\alpha,\beta\} \ominus \{\gamma,\delta\} \Big)$$

- Multiplication by a scalar($\star$): Let $i, j \in \mathcal{N}, i \leq k, j \leq k$. Then, for a path α, the multiplication by a scalar i, $i \star \alpha$ is defined in Table 2.3. The operation is extended

Figure 2.3: Union operation between two singleton sets of pair of paths, where $\{\alpha, \beta\}$ denote any other pair of paths and x, y, m, n can be any positive integer or ∞

$\cup$	$\{(f^D,g^D)\}$	$\{(f^{Im},g^D)\}$	$\{(f^D,g^{In})\}$	$\{(f^{Im},g^{In})\}$	$\{(\alpha, \beta)\}$
$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(f^D,g^D),$ $(\varepsilon,g^D),(f^D,\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(f^{Im},g^D),$ $(\varepsilon,g^D),(f^{Im},\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(f^D,g^{In}),$ $(\varepsilon,g^{In}),(f^D,\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(f^{Im},g^{In}),$ $(\varepsilon,g^{In}),(f^{Im},\varepsilon)\}$	$\{(\varepsilon,\varepsilon),$ $(\alpha, \beta)\}$
$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D),(f^D,g^D)\}$	$\{(\varepsilon,g^D),(f^{Im},g^D)\}$	$\{(\varepsilon,g^D),(f^D,g^{In}),$ $(\varepsilon,g^{In}),(f^D,g^D)\}$	$\{(\varepsilon,g^D),(f^{Im},g^{In}),$ $(\varepsilon,g^{In}),(f^{Im},g^D)\}$	$\{(\varepsilon,g^D),$ $(\alpha, \beta)\}$
$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy}),(f^D,g^D)$ $(\varepsilon,g^D),(f^D,g^{Iy})\}$	$\{(\varepsilon,g^{Iy}),(f^{Im},g^D)$ $(\varepsilon,g^D),(f^{Im},g^{Iy})\}$	$\{(\varepsilon,g^{In+y}),(f^D,g^{In+y})\}$	$\{(\varepsilon,g^{In+y}),(f^{Im},g^{In+y})\}$	$\{(\varepsilon,g^{Iy}),$ $(\alpha, \beta)\}$
$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon),(f^D,g^D)\}$	$\{(f^D,\varepsilon),(f^{Im},g^D)$ $(f^{Im},\varepsilon),(f^D,g^D)\}$	$\{(f^D,\varepsilon),(f^D,g^{In})\}$	$\{(f^D,\varepsilon),(f^{Im},g^{In})$ $(f^{Im},\varepsilon).\ (f^D,g^{In})\}$	$\{(f^D,\varepsilon),$ $(\alpha, \beta)\}$
$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix},\varepsilon),(f^D,g^D)$ $(f^{Ix},g^D),(f^D,\varepsilon)\}$	$\{(f^{Ix+m},\varepsilon),(f^{Ix+m},g^D)\}$	$\{(f^{Ix},\varepsilon),(f^D,g^{In})$ $(f^{Ix},g^{In}),(f^D,\varepsilon)\}$	$\{(f^{Ix+m},\varepsilon),(f^{Ix+m},g^{In})\}$	$\{(f^{Ix},\varepsilon),$ $(\alpha, \beta)\}$
$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	$\{(f^D,g^D),$ $(f^{Im},g^D)\}$	$\{(f^D,g^D),$ $(f^D,g^{In})\}$	$\{(f^D,g^D),(f^D,g^{In}),$ $(f^{Im},g^D)\ ,\ (f^{Im},g^{In})\}$	$\{(f^D,g^D),$ $(\alpha, \beta)\}$
$\{(f^{Ix},g^D)\}$	$\{(f^{Ix},g^D),$ $(f^D,g^D)\}$	$\{(f^{Ix+m},g^D)\}$	$\{(f^{Ix},g^D)\ ,\ (f^D,g^{In}),$ $(f^{Ix},g^{In})\ ,\ (f^D,g^D)\}$	$\{(f^{Ix+m},g^D),$ $(f^{Ix+m},g^{In})\}$	$\{(f^{Ix},g^D),$ $(\alpha, \beta)\}$
$\{(f^D,g^{Iy})\}$	$\{(f^D,g^D),$ $(f^D,g^{Iy})\}$	$\{(f^D,g^{Iy})\ ,\ (f^{Im},g^D),$ $(f^{Im},g^{Iy})\ ,\ (f^D,g^D),$	$\{(f^D,g^{In+y})\}$	$\{(f^D,g^{In+y}),\ (f^D,g^{Iy}),$ $(f^{Im},g^{In+y})\}$	$\{(f^D,g^{Iy})\ ,$ $(\alpha, \beta)\}$
$\{(f^{Ix},g^{Iy})\}$	$\{(f^D,g^D),(f^{Ix},g^{Iy}),$ $(f^D,g^{Ix}),(f^{Ix},g^D)\}$	$\{(f^{Ix+m},g^D),$ $(f^{Ix+m},g^{Iy})\}$	$\{(f^D,g^{In+y}),$ $(f^{Ix},g^{In+y})\}$	$\{(f^{Im+x},g^{In+y})\}$	$\{(f^{Ix},g^{Iy}),$ $(\alpha, \beta)\}$

$\cup$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^{In})\}$	$\{(f^D,\varepsilon)\}$	$\{(f^{Im},\varepsilon)\}$	$\{(\alpha, \beta)\}$
$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(\varepsilon,g^D)\}$	$\{(\varepsilon,\varepsilon),(\varepsilon,g^{In})\}$	$\{(\varepsilon,\varepsilon),(f^D,\varepsilon)\}$	$\{(\varepsilon,\varepsilon),(f^{Ix},\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$ $(\alpha, \beta)\}$
$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D),(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D),(\varepsilon,g^{In})\}$	$\{(\varepsilon,g^D),(f^D,\varepsilon),$ $(\varepsilon,\varepsilon),(f^D,g^D)\}$	$\{(\varepsilon,g^D),(f^{Im},\varepsilon),$ $(\varepsilon,\varepsilon),(f^{Im},g^D)\}$	$\{(\varepsilon,g^D),$ $(\alpha, \beta)\}$
$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy}),(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,g^{Iy}),(\varepsilon,g^D)\}$	$\{(\varepsilon,g^{Iy+n})\}$	$\{(\varepsilon,g^{Iy}),(f^D,\varepsilon),$ $(\varepsilon,\varepsilon),(f^D,g^{Iy})\}$	$\{(\varepsilon,g^{Iy}),(f^{Im},\varepsilon),$ $(\varepsilon,\varepsilon),(f^{Im},g^{Iy})\}$	$\{(\varepsilon,g^{Iy}),$ $(\alpha, \beta)\}$
$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,g^D),$ $(f^D,g^D),(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,g^{In}),$ $(f^D,g^{In}),(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon),(f^{Ix},\varepsilon)\}$	$\{(f^D,\varepsilon),$ $(\alpha, \beta)\}$
$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix},\varepsilon),(\varepsilon,\varepsilon)\}$	$\{(f^{Ix},\varepsilon),(\varepsilon,g^D),$ $(f^{Ix},g^D),(\varepsilon,\varepsilon)\}$	$\{(f^{Ix},\varepsilon),(\varepsilon,g^{In}),$ $(f^{Ix},g^{In}),(\varepsilon,\varepsilon)\}$	$\{(f^{Ix},\varepsilon),(f^D,\varepsilon)\}$	$\{(f^{Ix+m},\varepsilon)\}$	$\{(f^{Ix},\varepsilon),$ $(\alpha, \beta)\}$
$\{(f^D,g^D)\}$	$\{(f^D,g^D),(\varepsilon,\varepsilon),$ $(f^D,\varepsilon),(\varepsilon,g^D)\}$	$\{(f^D,g^D),(\varepsilon,g^D)\}$	$\{(f^D,g^D),(\varepsilon,g^{In}),$ $(f^D,g^{In}),(\varepsilon,g^D)\}$	$\{(f^D,g^D),(f^D,\varepsilon)\}$	$\{(f^D,g^D),(f^{Ix},\varepsilon),$ $(f^D,\varepsilon),(f^{Im},g^D)\}$	$\{(f^D,g^D),$ $(\alpha, \beta)\}$
$\{(f^{Ix},g^D)\}$	$\{(f^{Ix},g^D),(\varepsilon,\varepsilon),$ $(f^{Ix},\varepsilon),(\varepsilon,g^D)\}$	$\{(f^{Ix},g^D),(\varepsilon,g^D)\}$	$\{(f^{Ix},g^D),(\varepsilon,g^{In}),$ $(f^{Ix},g^{In}),(\varepsilon,g^D)\}$	$\{(f^{Ix},g^D),(f^D,\varepsilon),$ $(f^{Ix},\varepsilon),(f^D,g^D)\}$	$\{(f^{Ix+m},g^D),(f^{Ix+m},\varepsilon)\}$	$\{(f^{Ix},g^D),$ $(\alpha, \beta)\}$
$\{(f^D,g^{Iy})\}$	$\{(f^D,g^{Iy}),(\varepsilon,\varepsilon),$ $(f^D,\varepsilon),(\varepsilon,g^{Iy})\}$	$\{(f^D,g^{Iy}),(\varepsilon,g^D),$ $(f^D,g^D),(\varepsilon,g^{Iy})\}$	$\{(f^D,g^{Iy+n}),(\varepsilon,g^{Iy+n})\}$	$\{(f^D,g^{Iy}),(f^D,\varepsilon)\}$	$\{(f^D,g^{Iy}),(f^{Im},\varepsilon),$ $(f^D,\varepsilon),(f^{Im},g^{Iy})\}$	$\{(f^D,g^{Iy}),$ $(\alpha, \beta)\}$
$\{(f^{Ix},g^{Iy})\}$	$\{(f^{Ix},g^{Iy}),(\varepsilon,\varepsilon),$ $(f^{Ix},\varepsilon),(\varepsilon,g^{Iy})\}$	$\{(f^{Ix},g^{Iy}),(\varepsilon,g^D),$ $(f^{Ix},g^D),(\varepsilon,g^{Iy})\}$	$\{(f^{Ix},g^{Iy+n}),(\varepsilon,g^{Iy+n})\}$	$\{(f^{Ix},g^{Iy}),(f^D,\varepsilon),$ $(f^{Ix},\varepsilon),(f^D,g^{Iy})\}$	$\{(f^{Ix+m},g^{Iy}),(f^{Ix+m},\varepsilon)\}$	$\{(f^{Ix},g^{Iy}),$ $(\alpha, \beta)\}$

Figure 2.4: Intersection operation between two singleton sets of pair of paths, where $\{\alpha, \beta\}$ denote any other pair of paths and $a \star b$ = min(a,b).

$\bigcap$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^{In})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,g^D)\}$	$\{(f^D,g^{In})\}$	$\{(f^D,g^{I\infty})\}$	$\{(\alpha, \beta)\}$
$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^D)\}$	ϕ	$\{(\varepsilon,g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^{Iy})\}$	ϕ	ϕ	$\{(\varepsilon,g^{Iy\star n})\}$	$\{(\varepsilon,g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^{I\infty})\}$	ϕ	ϕ	$\{(\varepsilon,g^{In})\}$	$\{(\varepsilon,g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^D,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^D,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ
$\{(f^D,g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^D)\}$	ϕ	ϕ	ϕ
$\{(f^D,g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^{Iy\star n})\}$	$\{(f^D,g^{Iy})\}$	ϕ
$\{(f^D,g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^{In})\}$	$\{(f^D,g^{I\infty})\}$	ϕ
$\{(f^{Ix},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{Ix},g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{Ix},g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{Ix},g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{I\infty},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{I\infty},g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{I\infty},g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{I\infty},g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ

$\bigcap$	$\{(f^{Im},\varepsilon)\}$	$\{(f^{Im},g^D)\}$	$\{(f^{Im},g^{In})\}$	$\{(f^{Im},g^{I\infty})\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^{In})\}$	$\{(f^{I\infty},g^{I\infty})\}$	$\{(\alpha, \beta)\}$
$\{(\varepsilon,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(\varepsilon,g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^D,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^D,g^D)\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^D,g^{Iy})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^D,g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ
$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix\star m},\varepsilon)\}$	ϕ	ϕ	ϕ	$\{(f^{Ix},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ
$\{(f^{Ix},g^D)\}$	ϕ	$\{(f^{Ix\star m},g^D)\}$	ϕ	ϕ	ϕ	$\{(f^{Ix},g^D)\}$	ϕ	ϕ	ϕ
$\{(f^{Ix},g^{Iy})\}$	ϕ	ϕ	$\{(f^{Ix\star m},g^{Iy\star n})\}$	$\{(f^{Ix\star m},g^{Iy})\}$	ϕ	ϕ	$\{(f^{Ix},g^{Iy\star n})\}$	$\{(f^{Ix},g^{Iy})\}$	ϕ
$\{(f^{Ix},g^{I\infty})\}$	ϕ	ϕ	$\{(f^{Ix\star m},g^{In})\}$	$\{(f^{Ix\star m},g^{I\infty})\}$	ϕ	ϕ	$\{(f^{Ix},g^{In})\}$	$\{(f^{Ix},g^{In})\}$	ϕ
$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{Im},\varepsilon)\}$	ϕ	ϕ	ϕ	$\{(f^{I\infty},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ
$\{(f^{I\infty},g^D)\}$	ϕ	$\{(f^{Im},g^D)\}$	ϕ	ϕ	ϕ	$\{(f^{I\infty},g^D)\}$	ϕ	ϕ	ϕ
$\{(f^{I\infty},g^{Iy})\}$	ϕ	ϕ	$\{(f^{Im},g^{Iy\star n})\}$	$\{(f^{Im},g^{Iy})\}$	ϕ	ϕ	$\{(f^{I\infty},g^{Iy\star n})\}$	$\{(f^{I\infty},g^{Iy})\}$	ϕ
$\{(f^{I\infty},g^{I\infty})\}$	ϕ	ϕ	$\{(f^{Im},g^{In})\}$	$\{(f^{Im},g^{I\infty})\}$	ϕ	ϕ	$\{(f^{I\infty},g^{In})\}$	$\{(f^{I\infty},g^{I\infty})\}$	ϕ

Figure 2.5: Removal operation between two singleton sets of pair of paths, where $\{\alpha, \beta\}$ denote any other pair of paths and $a\#b$ = max(a-b,0).

$\ominus$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^{In})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,g^D)\}$	$\{(f^D,g^{In})\}$	$\{(f^D,g^{I\infty})\}$	$\{(\alpha, \beta)\}$
$\{(\varepsilon,\varepsilon)\}$	ϕ	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$
$\{(\varepsilon,g^D)\}$	ϕ	ϕ	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$
$\{(\varepsilon,g^{Iy})\}$	ϕ	ϕ	$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy})\}$	ϕ	$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy\#n})\}$	$\{(\varepsilon,g^{Iy})\}$
$\{(\varepsilon,g^{I\infty})\}$	ϕ	ϕ	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	ϕ	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$
$\{(f^D,\varepsilon)\}$	ϕ	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^D,\varepsilon)\}$
$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	ϕ	$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^D)\}$
$\{(f^D,g^{Iy})\}$	$\{(f^D,g^{Iy})\}$	ϕ	$\{(f^D,g^{Iy\#n})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^{Iy})\}$
$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	ϕ	$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^D,g^{I\infty})\}$
$\{(f^{Ix},\varepsilon)\}$	ϕ	$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},\varepsilon)\}$
$\{(f^{Ix},g^D)\}$	$\{(f^{Ix},g^D)\}$	ϕ	$\{(f^{Ix},g^D)\}$	$\{(f^{Ix},g^D)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^D)\}$
$\{(f^{Ix},g^{Iy})\}$	$\{(f^{Ix},g^{Iy})\}$	ϕ	$\{(f^{Ix},g^{Iy\#n})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^{Iy})\}$
$\{(f^{Ix},g^{I\infty})\}$	$\{(f^{Ix},g^{I\infty})\}$	ϕ	$\{(f^{Ix},g^{I\infty})\}$	$\{(f^{Ix},g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^{I\infty})\}$
$\{(f^{I\infty},\varepsilon)\}$	ϕ	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{I\infty},\varepsilon)\}$
$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	ϕ	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{I\infty},g^D)\}$
$\{(f^{I\infty},g^{Iy})\}$	$\{(f^{I\infty},g^{Iy})\}$	ϕ	$\{(f^{I\infty},g^{Iy\#n})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^{I\infty},g^{Iy})\}$
$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	ϕ	$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{I\infty},g^{I\infty})\}$

$\ominus$	$\{(f^{Im},\varepsilon)\}$	$\{(f^{Im},g^D)\}$	$\{(f^{Im},g^{In})\}$	$\{(f^{Im},g^{I\infty})\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^{In})\}$	$\{(f^{I\infty},g^{I\infty})\}$	$\{(\alpha,\beta)\}$
$\{(\varepsilon,\varepsilon)\}$	ϕ	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$
$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	ϕ	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	ϕ	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$	$\{(\varepsilon,g^D)\}$
$\{(\varepsilon,g^{Iy})\}$	$\{(\varepsilon,g^{Iy})\}$	ϕ	$\{(\varepsilon,g^{Iy\#n})\}$	ϕ	$\{(\varepsilon,g^{Iy})\}$	ϕ	$\{(\varepsilon,g^{Iy})\}$	ϕ	$\{(\varepsilon,g^{Iy})\}$
$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	ϕ	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	ϕ	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$	$\{(\varepsilon,g^{I\infty})\}$
$\{(f^D,\varepsilon)\}$	ϕ	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	ϕ	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$
$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	ϕ	$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	ϕ	$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$	$\{(f^D,g^D)\}$
$\{(f^D,g^{Iy})\}$	$\{(f^D,g^{Iy})\}$	ϕ	$\{(f^D,g^{Iy\#n})\}$	ϕ	$\{(f^D,g^{Iy})\}$	ϕ	$\{(f^D,g^{Iy\#n})\}$	ϕ	$\{(f^D,g^{Iy})\}$
$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	ϕ	$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	ϕ	$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$	$\{(f^D,g^{I\infty})\}$
$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix},\varepsilon)\}$	$\{(f^{Ix\#m},\varepsilon)\}$	$\{(f^{Ix\#m},\varepsilon)\}$	$\{(f^{Ix\#m},\varepsilon)\}$	$\{(f^{Ix},\varepsilon)\}$	ϕ	ϕ	ϕ	$\{(f^{Ix},\varepsilon)\}$
$\{(f^{Ix},g^D)\}$	$\{(f^{Ix\#m},g^D)\}$	ϕ	$\{(f^{Ix\#m},g^D)\}$	$\{(f^{Ix\#m},g^D)\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^D)\}$
$\{(f^{Ix},g^{Iy})\}$	$\{(f^{Ix\#m},g^{Iy})\}$	ϕ	$\{(f^{Ix\#m},g^{Iy\#n})\}$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^{Iy})\}$
$\{(f^{Ix},g^{I\infty})\}$	$\{(f^{Ix\#m},g^{I\infty})\}$	ϕ	$\{(f^{Ix\#m},g^{I\infty})\}$	$\{(f^{Ix\#m},g^{I\infty})\}$	ϕ	ϕ	ϕ	ϕ	$\{(f^{Ix},g^{I\infty})\}$
$\{(f^{I\infty},\varepsilon)\}$	ϕ	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	ϕ	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$	$\{(f^{I\infty},\varepsilon)\}$
$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	ϕ	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	ϕ	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$	$\{(f^{I\infty},g^D)\}$
$\{(f^{I\infty},g^{Iy})\}$	$\{(f^{I\infty},g^{Iy})\}$	ϕ	$\{(f^{I\infty},g^{Iy\#n})\}$	$\{(f^{I\infty},g^{Iy})\}$	$\{(f^{I\infty},g^{Iy})\}$	ϕ	$\{(f^{I\infty},g^{Iy\#n})\}$	$\{(f^{I\infty},g^{Iy})\}$	$\{(f^{I\infty},g^{Iy})\}$
$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	ϕ	$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	ϕ	$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$	$\{(f^{I\infty},g^{I\infty})\}$

to set of paths as:

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

2.2.2 Analysis

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

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

They intend to determine, at each program point, the field sensitive matrices D_F and I_F, and the boolean variables capturing field connectivity. The problem is formulated 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 sake the basic blocks are constructed with single statements each The definition of the confluence operator (merge) for various data flow values as used by the analysis is given below. The superscripts x and y are used to denote the values coming along two paths,

$$\begin{aligned}
\text{merge}(f^x_{pq}, f^y_{pq}) &= f^x_{pq} \vee f^y_{pq}, f \in \mathcal{F}, p,q \in \mathcal{H} \\
\text{merge}(p^x_{\texttt{Cycle}}, p^y_{\texttt{Cycle}}) &= p^x_{\texttt{Cycle}} \vee p^y_{\texttt{Cycle}}, p \in \mathcal{H} \\
\text{merge}(p^x_{\texttt{Dag}}, p^y_{\texttt{Dag}}) &= p^x_{\texttt{Dag}} \vee p^y_{\texttt{Dag}}, p \in \mathcal{H} \\
\text{merge}(D^x_F, D^y_F) &= D_F \text{ where } D_F[p,q] = D^x_F[p,q] \cup D^y_F[p,q], \forall p,q \in \mathcal{H} \\
\text{merge}(I^x_F, I^y_F) &= I_F \text{ where } I_F[p,q] = I^x_F[p,q] \cup I^y_F[p,q], \forall p,q \in \mathcal{H}
\end{aligned}$$

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

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

Field connectivity information is updated directly by the statement.

Few details about each of this basic statements are given below. The data flow values for each of these statements are shown in Figure 2.6.

- `p = malloc`: After this statement all the existing relationships of p get killed and it will point to a newly allocated object.It is considered that p can have an empty path to itself and it can interfere with itself using empty paths (or ε paths).

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

- `p=q`, `p=&(q→f)`, `p=q op n`: All these three pointer assignment statements are considered 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.

- `p → f = 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 relations.

- `p → f = q`: This statement first breaks the existing link f and then re-links the the heap object pointed to by p to the heap object pointed to by q. The kill effects are exactly same as described in the case of `p→f = null`. Only the generated relationships are described in Figure 2.6.

- `p = q → f`: The relations killed by the statement are same as that in case of `p = NULL`. The relations created by this statement are heavily approximated as not much information is available about the heap node pointed by `q → f` before the statement. After this statement p points to the heap object which is accessible from pointer q through f link.

`p = malloc()`	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$ $p^{gen}_{\texttt{Cycle}} = \textbf{False}$ $\quad p^{gen}_{\texttt{Dag}} = \textbf{False}$ $\forall s \in \mathcal{H},\ s \neq p,$ $D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $\quad D^{kill}_F[p,p] = D^{in}_F[p,p]$ $D^{gen}_F[p,s] = \emptyset$ $\quad D^{gen}_F[s,p] = \emptyset$ $\quad D^{gen}_F[p,p] = \{\varepsilon\}$ $I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{kill}_F[p,p] = I^{in}_F[p,p]$ $I^{gen}_F[p,s] = \emptyset$ $\quad I^{gen}_F[p,p] = \{\varepsilon,\varepsilon\}$
`p = NULL`	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$ $p^{gen}_{\texttt{Cycle}} = \textbf{False}$ $\quad p^{gen}_{\texttt{Dag}} = \textbf{False}$ $\forall s \in \mathcal{H},$ $D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $D^{gen}_F[p,s] = \emptyset$ $\quad D^{gen}_F[s,p] = \emptyset$ $I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{gen}_F[p,s] = \emptyset$

`p = q`
`p = &(q→f)`
`p = q op n`
	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$
	$p^{gen}_{\texttt{Cycle}} = q^{in}_{\texttt{Cycle}}[\text{q/p}]$ $\quad p^{gen}_{\texttt{Dag}} = q^{in}_{\texttt{Dag}}[\text{q/p}]$
	where $X[q/p]$ creates a copy of X with all occurrences of q replaced by p.
	$\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F},$
	$f_{ps} = f_{qs}$ $\quad f_{sp} = f_{sq}$
	$D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $\quad D^{kill}_F[p,p] = D^{in}_F[p,p]$
	$D^{gen}_F[p,s] = D^{in}_F[q,s]$ $\quad D^{gen}_F[s,p] = D^{in}_F[s,q]$ $\quad D^{gen}_F[p,p] = D^{in}_F[q,q]$
	$I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{gen}_F[p,s] = I^{in}_F[q,s]$
	$I^{kill}_F[p,p] = I^{in}_F[p,p]$ $\quad I^{gen}_F[p,p] = I^{in}_F[q,q]$

`p→f = null`	$p^{kill}_{\texttt{Cycle}} = \textbf{False},$ $\quad p^{kill}_{\texttt{Dag}} = \textbf{False}$ $p^{gen}_{\texttt{Cycle}} = \textbf{False},$ $\quad p^{gen}_{\texttt{Dag}} = \textbf{False}$ $\forall q,s \in \mathcal{H}, s \neq p,$ $f_{pq} = \textbf{False}$ $D^{kill}_F[p,q] = D^{in}_F[p,q] \triangleright f$ $\quad D^{kill}_F[s,q] = \emptyset$ $I^{kill}_F[p,s] = \{(\alpha,\beta) \mid (\alpha,\beta) \in I^{in}_F[p,q], \alpha \equiv f^*\}$ $I^{kill}_F[q,s] = \emptyset$ if $q \neq p$ $\quad I^{kill}_F[p,p] = \emptyset$

`p→f = q`	The KILL relations are same as that of `p→f = null` $p^{gen}_{\texttt{Cycle}} = (f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1))$ $\quad p^{gen}_{\texttt{Dag}} = f_{pq} \wedge (\lvert I_F[p,q] \rvert > 1)$ $q^{gen}_{\texttt{Cycle}} = f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1)$ $\quad q^{gen}_{\texttt{Dag}} = \textbf{False}$ $f_{pq} = \textbf{True}$ $s^{gen}_{\texttt{Cycle}} = ((\lvert D_F[s,p] \rvert \geq 1) \wedge f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee ((\lvert D_F[s,p] \rvert \geq 1) \wedge f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1))$ $\vee ((\lvert D_F[s,q] \rvert \geq 1) \wedge f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1)), \quad \forall s \in \mathcal{H}, s \neq p, s \neq q$ $s^{gen}_{\texttt{Dag}} = (\lvert D_F[s,p] \rvert \geq 1) \wedge f_{pq} \wedge (\lvert I_F[s,q] \rvert > 1), \ \forall s \in \mathcal{H}, s \neq p, s \neq q$ $D^{gen}_F[r,s] = \lvert D^{in}_F[q,s] \rvert \star D^{in}_F[r,p], \ s \neq p, \ r \notin \{p,q\}$ $D^{gen}_F[r,p] = \lvert D^{in}_F[q,p] \rvert \star D^{in}_F[r,p], \ r \neq p$ $D^{gen}_F[p,r] = \lvert D^{in}_F[q,r] \rvert \star (D^{in}_F[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}), \quad r \neq q$ $D^{gen}_F[p,q] = \{f^D\} \cup (\lvert D^{in}_F[q,q] - \{\varepsilon\} \rvert \star \{f^{I1}\}) \cup (\lvert D^{in}_F[q,q] \rvert \star (D^{in}_F[p,p] \ominus \{\varepsilon\}))$ $D^{gen}_F[q,q] = 1 \star D^{in}_F[q,p]$ $D^{gen}_F[q,r] = \lvert D^{in}_F[q,r] \rvert \star D^{in}_F[q,p], \ r \notin \{p,q\}$ $I^{gen}_F[p,q] = \{(f^D, \varepsilon)\} \cup ((1 \star (D^{in}_F[p,p] \ominus \{\varepsilon\})) \times \{\varepsilon\})$ $I^{gen}_F[p,r] = (1 \star (D^{in}_F[p,p] \ominus \{\varepsilon\})) \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r]\} \cup \{f^D\} \times \{\beta \mid (\varepsilon,\beta) \in I^{in}_F[q,r]\}$ $\cup \{f^{I1}\} \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r], \alpha \neq \varepsilon\}, \qquad r \notin \{p,q\}$ $I^{gen}_F[s,q] = (1 \star D^{in}_F[s,p]) \times \{\varepsilon\}, \ s \notin \{p,q\}$ $I^{gen}_F[s,r] = (1 \star D^{in}_F[s,p]) \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r]\}, \qquad s \notin \{p,q\}, \ r \notin \{p,q\}, \ s \neq r$
`p = q→f`	The KILL relations are same as that of `p = NULL` $p^{gen}_{\texttt{Cycle}} = q^{in}_{\texttt{Cycle}} \qquad p^{gen}_{\texttt{Dag}} = q^{in}_{\texttt{Dag}}$ $f_{qp} = \textbf{True} \quad h_{pr} = \lvert D^{in}_F[q,r] \triangleright f \rvert \geq 1 \quad \forall h \in \mathcal{F}, \forall r \in \mathcal{H}$ $\mathcal{U} \quad = \quad \{\varepsilon\} \cup \bigcup_{f \in \mathcal{F}} \{f^D, f^{I\infty}\}$ $D_1[p,s] = \mathcal{U} \quad \forall s \in \mathcal{H}, s \neq p \wedge D^{in}_F[q,s] \triangleright f \neq \emptyset$ $D_1[p,p] = \begin{cases} \mathcal{U} & q.\text{shape evaluates to } \texttt{Cycle} \\ \{\varepsilon\} & \text{Otherwise} \end{cases}. \qquad I_1[p,p] = \mathcal{U} \times \mathcal{U}$ $D_2[s,p] = \infty \star D^{in}_F[s,q] \quad \forall s \in \mathcal{H}, s \neq q$ $D_2[q,p] = \{f^D\} \cup (\infty \star (D^{in}_F[q,q] \ominus \{\varepsilon\})) \cup \mathcal{U}$ $I_2[s,p] = D_2[s,p] \times \{\varepsilon\} \quad \forall s \in \mathcal{H}$ $D_3[s,p] = \{\alpha \mid (f^D, \alpha) \in I^{in}_F[q,s]\}$ $I_3[s,p] = \{\alpha \mid (f^*, \alpha) \in I^{in}_F[q,s]\} \times \mathcal{U}$ Finally I_F and D_F relations are: $D^{gen}_F[r,s] = D_1[r,s] \cup D_2[r,s] \cup D_3[r,s] \quad \forall r,s \in \mathcal{H}$ $I^{gen}_F[r,s] = I_1[r,s] \cup I_2[r,s] \cup I_3[r,s] \quad \forall r,s \in \mathcal{H}$

Figure 2.6: Data flow values corresponding to each statement.

Chapter 3

Enhancements to Field Sensitive Analysis

3.1 Enhancements

In this section we will discuss about all the necessary enhancements and amendments done to the data flow values (refer Fig 2.6). Those mainly involve corrections, tackling unhandled cases and augmentations to make the analysis more precise.

Correctness: Lets discuss a scenario which involve correctness issue.

```
S1.  l→f = m;
S2.  m→f = l;
S3.  l = null;
```

Example 3. *Consider the sample code given above. After statement* `S1` *there is a path from* `l` *to* `m` *via field* `f` *and after* `S2` *there is a path from* `m` *to* `l` *also via field* `f` *hence creating a cycle. When* `S3` *is done there is still a cycle at* `m`*. The relevant Direction matrix entries after* `S2` *are:*

$$\begin{aligned} \textit{After } \texttt{S1} : D_F[l,m] &= f^D, f_{l,m} = 1 \\ \textit{After } \texttt{S2} : D_F[l,m] &= f^D, D_F[m,l] = f^D, f_{l,m} = 1, f_{m,l} = 1 \end{aligned}$$

The boolean equation of $l_{\texttt{Cycle}}$ *after* `S2` *is*

$$\{f_{l,m} \wedge (|D_F[m,l]| >= 1)\} \vee \{(f_{m,l} \wedge (|D_F[l,m]| >= 1))\}$$

which evaluates to true assuring that there is a cycle at `l` *after* `S2`*. After statement* `S3` *all the Direction and Interference information corresponding to* `l` *are killed, i.e.* $D_F[l,m] = \emptyset, D_F[m,l] = \emptyset, f_{l,m} = 0, f_{m,l} = 0$.

As a result $\texttt{l}_{\texttt{Cycle}}$ *gets evaluated to 0 inferring it is not a CYCLE, which is not true.*

Solution: The problem is, after statement `S3` we are killing all the information related to pointer variable `l` even though the graph structure still contains the corresponding heap object. We need to somehow preserve the information about the node which is labeled with name `l` before `S3` and unlabeled afterwards.

For this we create a new dummy pointer variable (say δ) pointing to the same node pointed by `l` before `S3`. This is done by adding a statement $\delta = l$ before `S3` such that this particular statement will not be having any kill information. The GEN information will be same as the pointer statement `p = q`. Also we replace any information about the term l by δ, i.e. in the D_F, I_F matrices and in all the boolean equations corresponding to all the heap pointers, we replace the occurrences of l by δ. So the solution can be generalized as:

Whenever any statement of type *Allocations* or *Pointer Assignments* are encountered, we add a new statement $\delta = p$ before it where δ is a dummy variable of same type as p. For this new statement we have the following gen and kill relations.

$$\delta^{gen}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}[p/\delta] \quad \delta^{gen}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}[p/\delta]$$

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

$$\begin{array}{cc} s^{kill}_{\texttt{Cycle}} = s^{in}_{\texttt{Cycle}} & s^{kill}_{\texttt{Dag}} = s^{in}_{\texttt{Dag}} \\ s^{gen}_{\texttt{Cycle}} = s^{in}_{\texttt{Cycle}}[p/\delta] & s^{gen}_{\texttt{Dag}} = s^{in}_{\texttt{Dag}}[p/\delta] \end{array}$$

where $X[q/p]$ creates a copy of X with all occurrences of q replaced by p.
$\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F}$

$$\begin{array}{cc} f_{\delta s} = f_{ps} & f_{s\delta} = f_{sp} \\ D^{gen}_F[\delta, s] = D^{in}_F[p, s] & D^{gen}_F[s, \delta] = D^{in}_F[s, p] \\ D^{gen}_F[\delta, \delta] = D^{in}_F[p, p] & I^{gen}_F[\delta, s] = I^{in}_F[p, s] \\ I^{gen}_F[\delta, \delta] = I^{in}_F[p, p] & \end{array}$$

We have two options in adding the new statement δ `= p`: Use the same dummy δ variable every time or use different dummy variables for each new statement added. Now if we use the same variable then at some point different unlabeled nodes will be using the same name δ causing less precise results. We can also use new variable every time for more precise results, but this will increase memory consumption. So usage of a single variable or multiple variables is a matter of trade off between accuracy and memory consumption. As this statement doesn't have any kill information, it may happen at some point that this pointer is referring to a summarized node which indeed contains distinct nodes. This can have an effect on preciseness, details about its effect are given in the Results Chapter.

Unhandled Dataflow Information: Before we go into this we need to understand what does $\varepsilon \in D^{in}_F[p, q]$ convey. It means that we can reach q from p through the path Epsilon, which simply says p and q are pointing to the same heap object, i.e. they both are aliases.

Some of the data flow information present in Fig: 2.6 (shaded with blue) didn't consider the effect on aliases, and so we considered those as well.

- `p → f = NULL`:

  We can see that the equation $D_F^{kill}[s,q] = \emptyset$ doesn't consider the case when s and p are aliases, in that case $D_F^{kill}[s,q]$ should be equal to $D_F^{in}[s,q] \triangleright f$, which is similar what $D_F^{kill}[p,q]$ is assigned. A similar modification is also required for the equation $I_F^{kill}[q,s] = \emptyset$ when p and q are aliases. In that case we have $I_F^{kill}[q,s] = \{(\alpha,\beta) \mid (\alpha,\beta) \in I_F^{in}[q,s], \alpha \equiv f^* \}$, otherwise the original equation holds good. We fine tune all the data flow values so as to incorporate the such effects of aliases in our analysis (refer Fig: 3.4). Now consider a scenario that there is a self loop at p so that the entry $I_F^{kill}[p,p]$ will contain something like $\{f^D, \varepsilon\}$ which indeed should be killed once this statement is encountered. So we write $I_F^{kill}[p,p] = \{(\alpha,\beta) \mid (\alpha,\beta) \in I_F^{in}[p,p], \alpha \equiv f^* \}$.

- `p → f = q`:

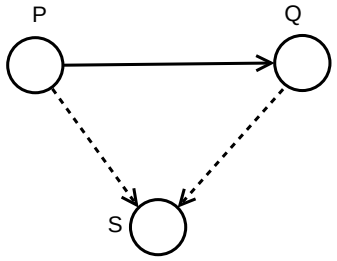


Figure 3.1: Heap graphs

  Consider the heap graph in Fig: 3.1, you can notice that the heap nodes pointed by p and q interfere at q as well as at s. The latter part is not captured by $I_F^{gen}[p,q]$ i.e

  $I_F^{gen}[p,q] = \{(f^D,\varepsilon)\} \cup ((1 \star (D_F^{in}[p,p] \ominus \{D_F^{in}[p,p] \triangleright f \cup \{\varepsilon\}\})) \times \{\varepsilon\})$

  This equation is not considering the effect when both p and q reach some node pointed by s at which they interfere. To handle this case we look at the set of all heap pointer's s which is not equal to p or q but has paths to it from p and q. Going by the definition of Interference matrix we find the fields through which these two pointers interfere, which is nothing but $D_F^{in}[p,s] \times D_F^{in}[q,s]$. The Union of all such elements for each s constitute our final information. Thus

  $\{(f^D,\varepsilon)\} \cup ((1 \star (D_F^{in}[p,p] \ominus \{D_F^{in}[p,p] \triangleright f \cup \{\varepsilon\}\})) \times \{\varepsilon\}) \cup I$
  where $I = \bigcup_{s \in \mathcal{H}, s \neq p,q} \{D_F^{in}[p,s] \times D_F^{in}[q,s] \mid |D_F^{in}[p,s]| > 1, |D_F^{in}[q,s]| > 1\}$

- `p = q → f`:

  Let up consider the sample code given below. After statement `S2` both p and q will be pointing to the same heap structure.

```
S1.  y→f = x;
S2.  y = y→f;
```

At the end of statement S1, $D[y,x]$ would contain the entry f^D. Hence at statement S2, $D_1[y,x]$ is assigned $\mathcal{U}$, but if we could somehow find that x and y are aliases after this statement we could escape this approximation and assign ε to $D_1[y,x]$. After any statement $p = q \rightarrow f$, p will be pointing to that node which is directly reachable from q through field f. If there is any pointer s which is reachable from q directly or indirectly through field f that node would be reachable from p also. But as we don't know through which field p can reach s, $\mathcal{U}$ is assigned to $D_1[p,s]$.

$$D_1[p,s] = \mathcal{U} \quad \forall s \in \mathcal{H}, s \neq p \wedge D_F^{in}[q,s] \triangleright f \neq \emptyset$$

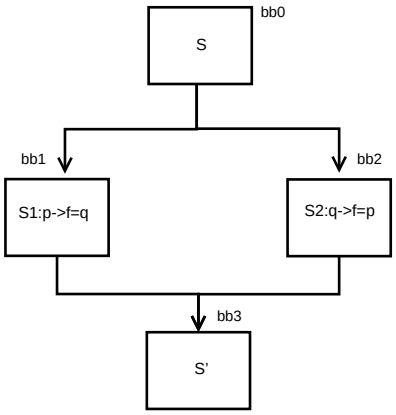
But if we consider only those s which can be directly reachable from q, it would simply be an alias for p and in that case we could just assign ε to $D_1[p,s]$. For this we need to check whether $D_F^{in}[q,s] \triangleright f = f^D$, if true then s and p are aliases after the statement. This change for $D_1[p,s]$is reflected below

$$\forall s \in \mathcal{H}, s \neq p$$
$$D_1[p,s] = \begin{cases} \mathcal{U} & \{D_F^{in}[q,s] \triangleright f - f^D\} \neq \emptyset \\ \{\varepsilon\} & D_F^{in}[q,s] \triangleright f = f^D \end{cases}$$

After p = q → f there will be a path from q to p through field f. This is reflected in the equation $D_2[q,p] = \{f^D\} \cup (\infty \star (D_F^{in}[q,q] \ominus \{\varepsilon\})) \cup \mathcal{U}$. We can also see $\mathcal{U}$ appended at the end, this was added because we were not sure if there was another path by which $q \rightarrow f$ can be reached from q . One observation here is that, if there was a path other than through f that the heap node pointed by $q \rightarrow f$ is reachable from heap node pointed by q then shape at q would have been a DAG or a CYCLE. So when the shape at q is a TREE initially there would be just one path from node pointed by q to node pointed by $q \rightarrow f$ and that is through f only. Writing the same thing according to our data flow values

$$if q \neq p$$
$$D_2[q,p] = \begin{cases} f^D & q_{in}^{dag} = \textbf{False}, q_{in}^{cycle} = \textbf{False} \\ \{f^D\} \cup (\infty \star (D_F^{in}[q,q] \ominus \{\varepsilon\})) \cup \mathcal{U} & \text{Otherwise} \end{cases}$$

Information Passed to successors: Here we will discuss about the less precise results that we are obtaining while passing the boolean equations to its successorss according to [DK12]. Also we will discuss ways to fix this. Let us look at the Fig: 3.2(a), it contains a single statement in the if and else blocks. The Direction matrices at the OUT of bb1 and bb2 are shown in Fig: 3.2(b).



(a)Control flow graph

Basic block	D_F			I_F		
$OUT(bb1)$		p	q		p	q
	p	ε	$\{f^D\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon)\}$
	q	$\emptyset$	ε	q	$\{(\varepsilon,f^D))\}$	$\{(\varepsilon,\varepsilon)\}$
$OUT(bb2)$		p	q		p	q
	p	ε	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon),f^D)\}$
	q	$\{f^D\}$	ε	q	$\{(f^D,\varepsilon)\}$	$\{(\varepsilon,\varepsilon)\}$
$IN(bb3)$		p	q		p	q
	p	ε	$\{f^D\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$
	q	$\{f^D\}$	ε	q	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$

(b)Direction and Interference Matrices

Basic Block	Boolean Equations
OUT(bb1)	$(f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1))$
OUT(bb2)	$(f_{qp} \wedge (\lvert D_F[q,p] \rvert \geq 1))$
IN(bb3)	$((f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (\lvert D_F[q,p] \rvert \geq 1)) \cup (f_{qp} \wedge (\lvert D_F[q,p] \rvert \geq 1)))$

(c)Boolean equation of p_{cycle}

Figure 3.2: DataFlow values and CFG

f_{pq} and f_{qp} are TRUE from basic blocks bb1 and bb2 respectively. Now if we look at the equation of p_{cycle} at IN(bb3) from Fig: 3.2(c) and substitute the data flow values at IN(bb3) from Fig: 3.2(b) we can see that it evaluates to TRUE, inferring that the shape of p as CYCLE even though its a TREE in reality. The problem here is we are not taking into consideration that only one of the basic blocks among bb1, bb2 will be executed. Merging of these boolean equations does not capture that effect.

Solution: For this problem we propose a simple and effective solution which even reduces the memory consumption by a good amount. What we propose is, at any statement when we evaluate a boolean equation and pass it to its successor only if that equation evaluates to 1 otherwise we do not pass it to the successor at all. This solution works because at those basic statements which can change the heap shape, such as $p \rightarrow f = q$, whatever boolean equations are generated they alone are sufficient to determine the shape

of each pointer at that program point.

$$\begin{aligned} S1 : q \rightarrow g &= p \\ S2 : p \rightarrow f &= q \end{aligned}$$

Lets take a simple example of just two statements. As equations at `S1` evaluates to false and hence no boolean equation is passed to `S2`. It means the value of $p^{in}_{\texttt{Cycle}}$ for `S2` is FALSE and $p^{out}_{\texttt{Cycle}}$ of `S2` is same as $p^{gen}_{\texttt{Cycle}}$ of `S2`. The equation for $p^{gen}_{\texttt{Cycle}}$, which is the same given in Fig: 3.2(c) first row evaluates to TRUE hence detecting the shape correctly as a CYCLE.

There is one case where boolean equation is to be changed to get correct results which is for $p^{gen}_{\texttt{Dag}}$ of statement $p \rightarrow f = q$.

$$p^{gen}_{\texttt{Dag}} = (f_{pq} \wedge (|I_F[p,q]| > 1))$$

Consider the set of statements

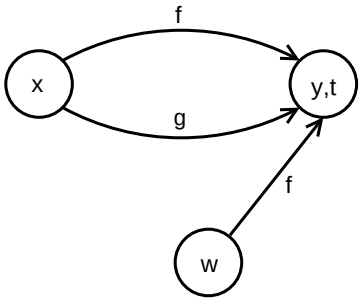
$$\begin{aligned} S1 : x \rightarrow f &= y \\ S2 : x \rightarrow g &= y \\ S3 : w \rightarrow f &= x \end{aligned}$$

After the first two statements a DAG is formed at x. After the third statement a link is created from w to x via field f, that means w also points to a DAG. As the shape of w before this statement was a TREE, $w^{in}_{\texttt{Dag}}$ is empty according to the above mentioned change. We know that $w^{kill}_{\texttt{Dag}}$ is also **False** for this statement, hence $w^{out}_{\texttt{Dag}}$ is nothing but $w^{gen}_{\texttt{Dag}}$. Now consider the equation of $w^{gen}_{\texttt{Dag}}$, it is equal to $(f_{wx} \wedge (|I_F[w,x]| > 1))$. The value of boolean variable f_{wx} is **True** but value of $(|I_F[w,x]| > 1))$ is **False** as the entry $I_F[w,x]$ would just contain $\{f^D, \varepsilon\}$. Finally evaluating the value of $w^{out}_{\texttt{Dag}}$ to **False**, which is incorrect.

If at all we have used the old approach of passing information to successors then $w^{in}_{\texttt{Dag}}$ would not have been empty, and would have successfully identified the shape at w as a DAG. Now to overcome this problem we have to change to the equation of $p^{gen}_{\texttt{Dag}}$. Whenever a statement $p \rightarrow f = q$ is encountered and shape of q before this statement is a DAG, then we can simply say that p also points to a DAG. This is formalized as

$$p^{gen}_{\texttt{Dag}} = (f_{pq} \wedge q^{in}_{\texttt{Dag}}) \vee (f_{pq} \wedge (|I_F[p,q]| > 1))$$

Limitation: This change in $p^{gen}_{\texttt{Dag}}$ will cause loss of accuracy in one scenario (also exhibited by Ghiya et al. [GH96]), but as a whole it has a lot of advantages in accuracy and memory consumption, so we have adopted this change. Lets look at that particular scenario.



(a)Heap Graph

S1: $x \rightarrow f = y$
S2: $x \rightarrow g = y$
S3: $t = x \rightarrow g$
S4: $w \rightarrow f = t$

(b)Set of statements

Figure 3.3: Example with Heap graph and Set of statements

Refer to Fig: 3.3, the effect of the set of statements are shown in the heap graph. After the second statement a DAG is formed at x and when the third statement is encountered the equation of x is copied to the equation of t. And at statement S4 the shape at w is reported as a DAG. When this change in $p^{gen}_{\texttt{Dag}}$ was not incorporated, the shape was reported as a TREE. The details about how this change is making the shape vary is shown below.

(According to the original equation of $p^{gen}_{\texttt{Dag}}$)

After S1:

$x^{out}_{\texttt{Dag}} = False$

After S2:

$x^{out}_{\texttt{Dag}} = (f_{xy} \wedge (|I_F[x,y]| > 1))$ (which evaluates to TRUE)

After S3:

$t^{out}_{\texttt{Dag}} = (f_{xy} \wedge (|I_F[x,y]| > 1))$ (which evaluates to TRUE)

After S4:

$w^{out}_{\texttt{Dag}} = (f_{wt} \wedge (|I_F[w,t]| > 1))$ (which evaluates to FALSE)

(According to the new equation of $p^{gen}_{\texttt{Dag}}$)

After S4:

$w^{out}_{\texttt{Dag}} = (f_{wt} \wedge t^{in}_{\texttt{Dag}}) \vee (f_{wt} \wedge (|I_F[w,t]| > 1))$ (which evaluates to TRUE)

The new version of equation of $w^{out}_{\texttt{Dag}}$ evaluates to TRUE because of the term $t^{in}_{\texttt{Dag}}$ which is also TRUE. As this part was not present in the previous version of $p^{gen}_{\texttt{Dag}}$ it correctly infers the shape of w as Tree.

3.2 Final Data Flow Equations

All the modifications that were proposed are shown in Fig: 3.4. It contains the final set of data flow values for each of the statements.

Let us first introduce some notations which will be used frequently in the following analysis.

$\forall p \in \mathcal{H}$, we introduce two notations $P^{\dagger}$ and $p^{\dagger} as$

$$\begin{aligned} P^{\dagger} &= \{r | r = p \vee \varepsilon \in D_F^{in}[r,p] \vee \varepsilon \in D_F^{in}[p,r]\} \text{ and} \\ p^{\dagger} &\in P^{\dagger} \end{aligned}$$

`p = malloc()`	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$ $p^{gen}_{\texttt{Cycle}} = \textbf{False}$ $\quad p^{gen}_{\texttt{Dag}} = \textbf{False}$ $\forall s \in \mathcal{H},\ s \neq p,$ $D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $\quad D^{kill}_F[p,p] = D^{in}_F[p,p]$ $D^{gen}_F[p,s] = \emptyset$ $\quad D^{gen}_F[s,p] = \emptyset$ $\quad D^{gen}_F[p,p] = \{\varepsilon\}$ $I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{kill}_F[p,p] = I^{in}_F[p,p]$ $I^{gen}_F[p,s] = \emptyset$ $\quad I^{gen}_F[p,p] = \{\varepsilon,\varepsilon\}$
`p = NULL`	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$ $p^{gen}_{\texttt{Cycle}} = \textbf{False}$ $\quad p^{gen}_{\texttt{Dag}} = \textbf{False}$ $\forall s \in \mathcal{H},$ $D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $D^{gen}_F[p,s] = \emptyset$ $\quad D^{gen}_F[s,p] = \emptyset$ $I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{gen}_F[p,s] = \emptyset$

`p = q`
`p =&(q→f)`
`p = q op n`
	$p^{kill}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}$ $\quad p^{kill}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}$
	$p^{gen}_{\texttt{Cycle}} = q^{in}_{\texttt{Cycle}}[\text{q/p}]$ $\quad p^{gen}_{\texttt{Dag}} = q^{in}_{\texttt{Dag}}[\text{q/p}]$
	where $X[q/p]$ creates a copy of X with all occurrences of q replaced by p.
	$\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F},$
	$f_{ps} = f_{qs}$ $\quad f_{sp} = f_{sq}$
	$D^{kill}_F[p,s] = D^{in}_F[p,s]$ $\quad D^{kill}_F[s,p] = D^{in}_F[s,p]$ $\quad D^{kill}_F[p,p] = D^{in}_F[p,p]$
	$D^{gen}_F[p,s] = D^{in}_F[q,s]$ $\quad D^{gen}_F[s,p] = D^{in}_F[s,q]$ $\quad D^{gen}_F[p,p] = D^{in}_F[q,q]$
	$I^{kill}_F[p,s] = I^{in}_F[p,s]$ $\quad I^{gen}_F[p,s] = I^{in}_F[q,s]$
	$I^{kill}_F[p,p] = I^{in}_F[p,p]$ $\quad I^{gen}_F[p,p] = I^{in}_F[q,q]$

`δ = p`	$\delta^{gen}_{\texttt{Cycle}} = p^{in}_{\texttt{Cycle}}[p/\delta]$ $\quad \delta^{gen}_{\texttt{Dag}} = p^{in}_{\texttt{Dag}}[p/\delta]$ $\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F}$ $s^{kill}_{\texttt{Cycle}} = s^{in}_{\texttt{Cycle}}$ $\quad s^{kill}_{\texttt{Cycle}} = s^{in}_{\texttt{Dag}}$ $s^{gen}_{\texttt{Cycle}} = s^{in}_{\texttt{Cycle}}[p/\delta]$ $\quad s^{gen}_{\texttt{Dag}} = s^{in}_{\texttt{Dag}}[p/\delta]$ where $X[q/p]$ creates a copy of X with all occurrences of q replaced by p. $\forall s \in \mathcal{H}, s \neq p, \forall f \in \mathcal{F}$ $f_{\delta s} = f_{ps}$ $\quad f_{s\delta} = f_{sp}$ $D^{gen}_F[\delta,s] = D^{in}_F[p,s]$ $\quad D^{gen}_F[s,\delta] = D^{in}_F[s,p]$ $\quad D^{gen}_F[\delta,\delta] = D^{in}_F[p,p]$ $I^{gen}_F[\delta,s] = I^{in}_F[p,s]$ $\quad I^{gen}_F[\delta,\delta] = I^{in}_F[p,p]$

<table>
<tr><td>p→f=null</td><td>
$p^{kill}_{\texttt{Cycle}}$ = False, $p^{kill}_{\texttt{Dag}}$ = False
$p^{gen}_{\texttt{Cycle}}$ = False, $p^{gen}_{\texttt{Dag}}$ = False
$\forall q,s \in \mathcal{H}, s \notin P^{\dagger}$
$f_{p^{\dagger}q}$ = False
$D^{kill}_F[p^{\dagger},q] = D^{in}_F[p,q] \triangleright f \qquad D^{kill}_F[s,q] = \emptyset$
$I^{kill}_F[p^{\dagger},s] = \{(\alpha,\beta) \mid (\alpha,\beta) \in I^{in}_F[p,q], \alpha \equiv f^*\}$
$I^{kill}_F[q,s] = \emptyset$ if $q \notin P^{\dagger} \qquad I^{kill}_F[p^{\dagger},p^{\dagger}] = \{(\alpha,\beta) \mid (\alpha,\beta) \in I^{in}_F[p,p], \alpha \equiv f^*\}$
</td></tr>
<tr><td>p→f = q</td><td>
The KILL relations are same as that of p→f = null
$p^{gen}_{\texttt{Cycle}} = (f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee (f_{pq} \wedge (|D_F[q,p]| \geq 1)) \quad p^{gen}_{\texttt{Dag}} = (f_{pq} \wedge q^{in}_{\texttt{Dag}}) \vee (f_{pq} \wedge (|I_F[p,q]| > 1))$
$q^{gen}_{\texttt{Cycle}} = f_{pq} \wedge (|D_F[q,p]| \geq 1) \qquad q^{gen}_{\texttt{Dag}}$ = False
$f_{p^{\dagger}q^{\dagger}}$ = True
$s^{gen}_{\texttt{Cycle}} = ((|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge q^{in}_{\texttt{Cycle}}) \vee ((|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|D_F[q,p]| \geq 1))$
$\vee ((|D_F[s,q]| \geq 1) \wedge f_{pq} \wedge (|D_F[q,p]| \geq 1)), \ \forall s \in \mathcal{H}, s \neq p,q$
$s^{gen}_{\texttt{Dag}} = (|D_F[s,p]| \geq 1) \wedge f_{pq} \wedge (|I_F[s,q]| > 1), \ \forall s \in \mathcal{H}, s \neq p,q$
$D^{gen}_F[r,s] = |D^{in}_F[q,s]| \star D^{in}_F[r,p], \ s \notin P^{\dagger}, \ r \notin P^{\dagger}, \ r \notin Q^{\dagger}$
$D^{gen}_F[r,p^{\dagger}] = |D^{in}_F[q,p]| \star D^{in}_F[r,p], \ r \notin P^{\dagger}$
$D^{gen}_F[p^{\dagger},r] = |D^{in}_F[q,r]| \star (D^{in}_F[p,p] \ominus \{\varepsilon\} \cup \{f^{I1}\}), \ \ r \notin Q^{\dagger}$
$D^{gen}_F[p^{\dagger},q^{\dagger}] = \{f^D\} \cup (|D^{in}_F[q,q] - \{\varepsilon\}| \star \{f^{I1}\}) \cup (|D^{in}_F[q,q]| \star (D^{in}_F[p,p] \ominus \{D^{in}_F[p,p] \triangleright f \cup \{\varepsilon\}\}))$
$D^{gen}_F[q^{\dagger},q^{\dagger}] = 1 \star D^{in}_F[q,p]$
$D^{gen}_F[q^{\dagger},r] = |D^{in}_F[q,r]| \star D^{in}_F[q,p], \ r \notin P^{\dagger}, \ r \notin Q^{\dagger}$
$I^{gen}_F[p^{\dagger},q^{\dagger}] = \{(f^D,\varepsilon)\} \cup ((1 \star (D^{in}_F[p,p] \ominus \{D^{in}_F[p,p] \triangleright f \cup \{\varepsilon\}\})) \times \{\varepsilon\}) \cup I$
where $I = \bigcup_{x \in \mathcal{H}, x \notin P^{\dagger}, Q^{\dagger}} \{D^{in}_F[p,x] \times D^{in}_F[q,x] \mid \ |D^{in}_F[p,x]| > 1, |D^{in}_F[q,x]| > 1\}$
$I^{gen}_F[p^{\dagger},r] = (1 \star (D^{in}_F[p,p] \ominus \{\varepsilon\})) \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r]\} \cup \{f^D\} \times \{\beta \mid (\varepsilon,\beta) \in I^{in}_F[q,r]\}$
$\cup \{f^{I1}\} \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r], \alpha \neq \varepsilon\}, \qquad r \notin P^{\dagger}, Q^{\dagger}$
$I^{gen}_F[s,q^{\dagger}] = (1 \star D^{in}_F[s,p]) \times \{\varepsilon\}, \ s \notin P^{\dagger}, \ Q^{\dagger}$
$I^{gen}_F[s,r] = (1 \star D^{in}_F[s,p]) \times \{\beta \mid (\alpha,\beta) \in I^{in}_F[q,r]\}, \qquad s \notin P^{\dagger}, Q^{\dagger}, \ r \notin P^{\dagger}, Q^{\dagger}, \ s \neq r$
</td></tr>
</table>

`p = q→f`	The KILL relations are same as that of `p = NULL` $p^{gen}_{\texttt{Cycle}} = q^{in}_{\texttt{Cycle}} \qquad p^{gen}_{\texttt{Dag}} = q^{in}_{\texttt{Dag}}$ $f_{q^\dagger p} = \textbf{True} \quad h_{pr} = \lvert D^{in}_F[q,r] \triangleright f \rvert \geq 1 \quad \forall h \in \mathcal{F}, \forall r \in \mathcal{H}$ $\mathcal{U} \quad = \quad \{\varepsilon\} \cup \bigcup_{f \in \mathcal{F}} \{f^D, f^{I\infty}\}$ $\forall s \in \mathcal{H}, s \neq p$ $D_1[p,s] = \begin{cases} \mathcal{U} & \{D^{in}_F[q,s] \triangleright f - f^D\} \neq \emptyset \\ \{\varepsilon\} & D^{in}_F[q,s] \triangleright f = f^D \end{cases} \qquad D_1[p,p] = \begin{cases} \mathcal{U} & q.\text{shape evaluates to } \texttt{Cycle} \\ \{\varepsilon\} & \text{Otherwise} \end{cases}.$ $I_1[p,p] = \mathcal{U} \times \mathcal{U}$ $D_2[q^\dagger, p] = \begin{cases} f^D & q^{tree}_{in} = \textbf{True} \\ \{f^D\} \cup (\infty \star (D^{in}_F[q,q] \ominus \{\varepsilon\})) \cup \mathcal{U} & \text{Otherwise} \end{cases} \quad if q \neq p$ $D_2[s,p] = \infty \star D^{in}_F[s,q] \quad \forall s \in \mathcal{H}, s \notin Q^\dagger,\ s \neq p$ $I_2[s,p] = D_2[s,p] \times \{\varepsilon\} \quad \forall s \in \mathcal{H}$ $D_3[s,p] = \{\alpha \mid (f^D, \alpha) \in I^{in}_F[q,s]\}$ $I_3[s,p] = \{\alpha \mid (f^*, \alpha) \in I^{in}_F[q,s]\} \times \mathcal{U}$ Final I_F and D_F relations are: $D^{gen}_F[r,s] = D_1[r,s] \cup D_2[r,s] \cup D_3[r,s] \quad \forall r,s \in \mathcal{H}$ $I^{gen}_F[r,s] = I_1[r,s] \cup I_2[r,s] \cup I_3[r,s] \quad \forall r,s \in \mathcal{H}$

Figure 3.4: Modified Data Flow Equations

Chapter 4

Subset Based Analysis

Sometimes data structures include auxiliary fields that are useful for traversing the data structure for debugging or diagnostic purposes. The presence of such fields, however can result in the more conservative shape. To illustrate this we present the following example .

4.1 Motivating Example

Example 4. *Consider the code segment in Program. 4.1 which has functions for searching data in a binary tree and inserting node into the same. The fields of structure* `Node`*, which is used to realize the functions, are also shown. The structure* `Node` *has three field pointers* `Left`, `Right` *and* `Parent`*. In the* `insert` *function we can see a cycle getting created due to the lines* `S5` *and* `S6`*. Take a look at Fig: 4.1 to see how the field sensitive analysis also gets the shape of p and s as a cycle at that point.*

Now in the `search` *function, the* `Parent` *pointer is not at all used, so ideally the shape that is actually traversed inside the* `search` *function is a Tree. But as the function is called with the root of the tree, whose shape gets evaluated to Cycle in the* `insert` *function, we infer that the shape of the root as Cycle in the* `search` *function as well even though the parent link is never been traversed.*

For the above case, unlike the Field Sensitive Analysis, Subset Based Analysis can identify the shape as Tree by considering the fields used by a function and using only those fields to infer the shape. Subset based analysis is a way in which only a subset of field pointers accessed in a function are used to get more precise shape information. We now present the details of the analysis.

4.2 Analysis

The Field Sensitive Analysis needs to be modified slightly for the subset based analysis to occur. During pre-processing of code, we parse each function separately and create a

Program 4.1: Motivating Example

```
Struct Node {
  Struct Node *Left,*Right,*Parent;
  int key;
};
typedef Struct Node Node;

bool search(Node *root,int key){
if(root)
  return (key==root->key)||search(root->Left,key)||search(root->Right,key);
return 0;
}


void insert(Node *root,int key){
    Node *s=root; //This pointer is used for traversing the Tree
    ..
    //New node is inserted as a child of s(s can be any node of the tree)
    Node *p;
    S1. p=(Node *)malloc(sizeof(Node));
    S2. p->Left=NULL;
    S3. p->Right=NULL;
    S4. p->key=key;
    S5. s->Left=p;
    S6. p->Parent = s;
    ..
}
```

set of fields associated with each of them. This set will contain all the field pointers that are used at least once in the function. We use S_F to denote the subset of fields used by function F. For the functions present in Program. 4.1

$$S_{insert} = \{ \text{Left , Right , Parent } \}$$
$$S_{search} = \{ \text{Left , Right } \}$$

During the evaluation of the boolean equations for function F, we restrict our analysis to the fields that are present in S_F. In this modified analysis when we reach a statement in F we have to modify the evaluation of equations according to the following rules.

Any boolean equation would generally contain the terms like f_{pq}, $D_F[p,q]$, $I_F[p,q]$. We replace each of the these terms by $f^{\#}_{pq}, D^{\#}[p,q], I^{\#}[p,q]$ respectively. Now lets understand what each of these terms mean.

- $f^{\#}_{p,q}$: This term will have a value as False if f is not accessed at any point in function F, i.e $f \notin S_F$ If its present then the value is same as that of $f_{p,q}$

$$f^{\#}_{p,q} = \begin{cases} f_{p,q} & f \in S_F \\ \textbf{False} & \text{Otherwise} \end{cases}$$

After	$Left_{p,t}$	$Parent_{t,p}$
S1	false	false
S2	false	false
S3	false	false
S5	true	false
S6	true	true

(a) Boolean Variables

After Stmt		D_F			I_F	
		p	s		p	s
S1	p	ε	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
		p	s		p	s
S2	p	ε	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
		p	s		p	s
S3	p	ε	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\emptyset$
	s	$\emptyset$	$\emptyset$	s	$\emptyset$	$\emptyset$
		p	s		p	s
S5	p	ε	$\emptyset$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,Left^D)\}$
	s	$\{Left^D\}$	$\emptyset$	s	$\{(Left^D,\varepsilon)\}$	$\emptyset$
		p	s		p	s
S6	p	ε	$\{Parent^D\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,Left^D),(Parent^D,\varepsilon)\}$
	s	$\{Left^D\}$	$\emptyset$	s	$\{(Left^D,\varepsilon),(\varepsilon,Parent^D)\}$	$\emptyset$

(b) Direction (D_F) and Interference (I_F) matrices

Heap Pointer	Boolean Equations
p_{cycle}	$\{\ (Parent_{p,s} \wedge False) \vee (Parent_{p,s} \wedge (\lvert D[s,p]\rvert >= 1))\ \} \vee \{\ Left_{s,p} \vee (\lvert D[p,s]\rvert >= 1)\ \}$
p_{dag}	$\{\ (Parent_{p,s} \wedge (\lvert I[p,s]\rvert > 1))\ \}$
s_{cycle}	$\{\ Parent_{p,s} \wedge (\lvert D[s,p]\rvert >= 1)\ \} \vee \{\ (Left_{s,p} \wedge False) \vee (Left_{s,p} \wedge (\lvert D[p,s]\rvert >= 1))\ \}$
s_{dag}	$\{\ (Left_{s,p} \wedge (\lvert I[s,p]\rvert > 1))\ \}$

(c) Boolean Equations after S6

Figure 4.1: Data Flow Values at each statement for Program. 4.1

- $D^{\#}[p,q]$: It contains the first field of all the paths that start from p and end at q except those whose first fields are not present in S_F. In other way this is nothing but the difference of the contents of $D[p,q]$ and the set of all first fields for the paths from p to q whose first fields are not present in S_F.

$$D^{\#}[p,q] \;=\; D[p,q] - \bigcup_{f \in \mathcal{F}, f \notin S_F} \{D[p,q] \triangleright f\}$$

- $I^{\#}[p,q]$: Very similar to the way how $D^{\#}[p,q]$ was defined this term is also defined.

$$I^{\#}[p,q] \;=\; I[p,q] - \bigcup_{f \in \mathcal{F}, f \notin S_F} \{I[p,q] \triangleright f\}$$

After making all these replacements we evaluate the equation to get the shape. Now that we have all the details about the analysis, lets look at how it works for the above mentioned motivating example.

Example 5. *The function* `search` *would be called with the root of the Tree as the parameter. In Program. 4.1 that the variable* `root` *is assigned to* `s` *in the insert function. So at the end of the insert function whatever boolean equation* `s` *would have,* `root` *also would have the same except replacing every* `s` *by* `root` *in the equation. Actually the insert function may be recursive or iterative, if we consider the equation of* `s` *at the end of complete insert function it would be very large and explaining would be a lot difficult. So we have considered the equation only at the end of statement* `S6`*. From Fig: 4.1(c). the the boolean equation of* `root` *is derived as*

$$\begin{aligned} \texttt{root}_{\texttt{cycle}} &= \{Parent_{p,root} \wedge (|D[root,p]| >= 1)\} \vee \{(Left_{root,p} \wedge False) \vee \\ &\quad (Left_{root,p} \wedge (|D[p,root]| >= 1))\} \\ \texttt{root}_{\texttt{dag}} &= \{(Left_{root,p} \wedge (|I[root,p]| > 1))\} \end{aligned}$$

Even with the dataflow values concerning the Boolean variables, Direction and Interference matrices, `root` *would have the same values corresponding to that of* `s` *at the end of statement* `S6`*. Hence at the start of* `search` *function* $Left_{root,p} = \textbf{\textit{True}}, Parent_{p,root} = \textbf{\textit{True}}$ *while the matrices are as given below.*

D	p	$root$	I	p	$root$
p	ε	$\{Parent^D\}$	p	$\{(\varepsilon,\varepsilon)\}$	$\{(\varepsilon,Left^D),(Parent^D,\varepsilon)\}$
$root$	$\{Left^D\}$	$\emptyset$	$root$	$\{(Left^D,\varepsilon),(\varepsilon,Parent^D)\}$	$\emptyset$

Before going into the search function we need to know the small change this function undergoes when represented in intermediate form(GIMPLE) on which the analysis takes place. The function would transform to

```
bool search(Node *root,int key){
      Node *temp1,*temp2;
      int tempInt;
St1: temp1=root->left;
St2: temp2=root->right;
      tempInt=root->key
      if(root)
        return (key==tempInt)||search(tmep1,key)||search(temp2,key);
      return 0;
}
```

At the statement `St1` *first time the boolean equation will be evaluated for this function. We are not considering the evaluation after* `St2` *because it would be done in the same way as that of* `St1`*. As* $temp1^{\text{gen}}_{\texttt{Cycle}} = root^{\text{in}}_{\texttt{Cycle}}$ *and* $temp1^{\text{kill}}_{\texttt{Cycle}} = temp1^{\text{in}}_{\texttt{Cycle}}$*,* $temp1^{\text{out}}_{\texttt{Cycle}}$ *is same as* $root^{\text{in}}_{\texttt{Cycle}}$*. Now lets look at each term in* $root^{\text{in}}_{\texttt{Cycle}}$ *and find what it evaluates to in the*

subset based analysis.

$$\begin{aligned}
Parent^{\#}_{p,root} &= \textbf{\textit{False}} \quad (Parent \notin S_{Search}) \\
Left^{\#}_{root,p} &= Left_{root,p} \ (Left \in S_{Search}) \\
D^{\#}[root,p] &= D[root,p] - D[root,p] \triangleright Parent \\
&= D[root,p] - \phi \\
&= \{Left^{D}\} \\
D^{\#}[p,root] &= D[p,root] - D[p,root] \triangleright Parent \\
&= D[p,root] - \{Parent^{D}\} \\
&= \phi
\end{aligned}$$

Substituting these above values in the equation of $root^{\text{in}}_{\texttt{Cycle}}$ *would result in* ***False****. In the similar way when we try to evaluate* $root^{\text{in}}_{\texttt{Dag}}$ *it would give us* ***False****, thus rightly detecting the shape of data structure traversed as Tree.*

Note that the subset based analysis increases the precision at the cost of extra computation required to compute $D^{\#}, I^{\#}$ and $f^{\#}$. Due to the lack of sufficient benchmarks its not very clear to us if and when this overhead can be threat by the gain in precision. More work is required to evaluate this trade off.

Chapter 5

Inter Procedural Analysis

Every programmer knows the importance of procedures and how vastly they are used in programs, so for any analysis to be effective handling procedures is very important. Just going by intra procedural analysis causes lot of information loss because we have to make worst case assumptions at the time of function calls. In Inter procedural analysis we have to take care of call-return, parameter passing local variables, recursion etc. apart from the work done in intra procedural analysis.

There are two variants of Interprocedural analysis, Context Sensitive and Context Insensitive. In Context Sensitive only interprocedural valid paths are considered during the data flow unlike Context insensitive in which some invalid paths also may be considered. Lets look at a small example.

Example 6. *Consider a program in which* `main` *function has 2 call statements for the function* `p1`*. Fig: 5.1(a) is a global control flow graph containing control flow graph of both procedures and considering each call statement as a goto from that statement to the start of the called procedure, similarly treating each return statement as goto from that statement to the instruction following the call statement by which this function was called. Just for the sake of clarity we have introduced return blocks after each call statement.*

Let the data flow values available just before the call statement `c1` *is* `df1` *and that before* `c2` *be* `df2`*. So* `df1` *and* `df2` *would be entering the function* `p1` *when called at* `c1`*,* `c2` *respectively. Similarly let the corresponding data flow values released out of function be* `df1'` *and* `df2'` *respectively. If we notice the edges between the functions* `main` *and* `p1`*, there is a path* `c1`*,* `Enter p1`*,* `Exit p1`*,* `return p1` *(i.e. the block below callblock c2); which is not followed in any execution sequence and hence it is not an inter procedurally valid path. When such paths are considered during data flow imprecise data is obtained leading to imprecise results. Such an analysis called Context Insensitive, and the analysis becomes Context Sensitive when we don't consider such invalid paths.*

Fig: 5.1 is a global control graph of small program, but in real life applications the size of this graph may become very large, so scalability and efficiency get more importance in Inter Procedural Analysis. Hence for some problems a compromise may be required

between precision and efficiency. We also discuss how our analysis goes about these compromises. In the next few sections we discuss in detail about one of the context sensitive approach, *Callstring method* and a newly proposed method called *shape sensitive approach*.

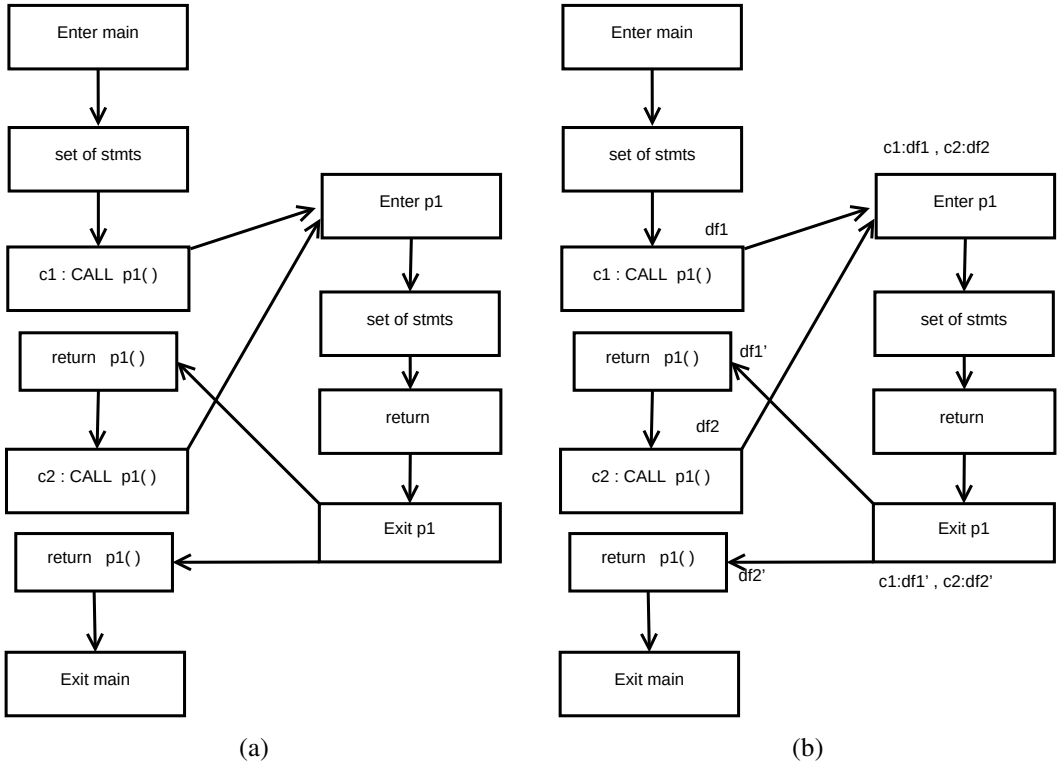


Figure 5.1: Global control flow graph

5.1 Callstrings Based Approach

Callstring Method is one of variants of Context Sensitive Interprocedural Analysis in which along with the data flow values a callstring is also propagated. This method of analysis was proposed by Sharir et .al [MS76]. Callstring at any program point means the sequence of unfinished procedure calls reaching that point starting form the main functions entry. This new tagged information will make the inter procedural analysis explicit, so at return statements information can be validly propagated. The data flow information at any program point looks like

$$< \text{Call String cs: Data Flow value d} >$$

We can notice this representation in Fig. 5.1(b). Since we know to which call site `df1` and `df2` belongs to, during the exit from function `p1`, we can be sure of sending a data flow value to its correct call site. As seen in the figure, `df1'`, `df2'` are sent to `c1`,`c2` respectively. In this approach only inter procedurally valid paths are used for data flow

transfer hence giving more precise results. But the problem with this approach is that we have to keep the dataflow values for each callstring in the memory, leading to increased memory consumption.

Lets discuss an overview of the implementation of call string based interprocedural Field Sensitive analysis. First we process the Control flow graph. We create separate basic blocks for call statements and also add a new basic block just below call block, i.e. the return block. Then we initialize the global and local worklist's; global worklist contains functions and separate local worklist's are present for each function which holds basic blocks. For handling termination of call string construction we use the method proposed in [KMR11].

[KMR11] use value based termination of call strings. In order to adopt that approach we used a call string map, that is present at the start block of each function and maps those data flow tuples whose call string's are different but data flow values are same . In a way what we are doing is discarding redundant call strings at start of a function. Consider two call strings σ_1 and σ_2 with same data flow values at the start of some function, both of them need not be propagated inside because both of them will anyway undergo the same transitions and generate the same output data flow values at the end of function. So we pass only, say the data flow value associated with σ_1, it undergoes some transitions inside the function and some dataflow is output at the end of the function. This dataflow value is directly copied into the dataflow value associated with σ_2 at the end of function and this saves us from processing the same dataflow value associated with σ_2 again.

As mentioned above the memory consumption is more because we have to maintain separate set of data flow values for each callstring. This problem with memory is quite an issue in the field sensitive analysis because when we ran the analysis even on a program like merging of linked list recursively [Pav10] the analysis couldn't complete because of large memory consumption. Hence we have moved from Context sensitive to Context insensitive. Though Context insensitive approach takes less memory the accuracy will decrease which is a trade off we have to make. Further study has to be made about how to design a memory efficient context sensitive analysis for this shape analysis technique.

5.2 Shape Sensitive Approach

In Context Insensitive analysis we have to compromise on accuracy and in Context Sensitive analysis we have to compromise on memory consumption, so if we could find some sort of a middle way approach of both of these, that would be a good gain where both accuracy and memory are optimized. Keeping that in mind we proposed the Shape Sensitive approach.

Lets consider the following scenario in Context Insensitive Analysis. Let we have a function with parameter as a heap pointer. The data flow values present at the start of the function when it was called the first time be DF1. At that point shape of that heap

pointer is a cycle. After a few statements again this function is called and the incoming dataflow values is DF2, let the shape of the same heap pointer now be a tree. Since its a context insensitive approach the data entering the function would be DF1 ∪ DF2. This also contains DF1 which was responsible for cycle being detected at the first time the function was called. So there is a good probability that now also the shape may be inferred as a cycle even though it's not.

So one natural thought that emerges is keep a separate set of data flow values for each shape at the start of functions. This is what we call shape sensitive method of context merging. Here what we do is, based on the shape of those heap pointer arguments at that point of function call, merging of call contexts happen. Lets look at this concept by using a small piece of code and later we will show the comparison of this approach with context insensitive approach.

Example 7. *Consider a function with one parameter which is a heap pointer, the shape of the node pointed to by this can be a TREE, DAG or a CYCLE. For such a function we associate an array of data flow values of size three. Let that array be denoted by IN[3]. IN[0] denotes that data flow values incoming when the shape at that heap pointer is a TREE, IN[1] when shape is a DAG and IN[2] when shape is CYCLE. So in the similar way if the number of heap pointer parameters are* `n` *then size of its corresponding IN would be* 3^n *. This size can be adjusted accordingly depending on compromise between precision and memory.*

We also maintain an OUT of the same size as IN which has the data flow values at the end of that function for its corresponding IN. Now if we look at the sample code given below, the number of parameters is one so size of IN is three. At `c1` *the shape of the parameter p is a cycle so the incoming values into that function are fed to IN[2] and when the function is returned OUT[2] is updated. Now at* `c2` *the shape of p is a TREE so this time IN[0] and OUT[0] are updated.*

```
struct Node
{
  struct Node *f,*g;
};

void foo(struct Node *s)
{
  struct Node *t;
  S: t=s;
  ....
}
```

```
int main()
{
  struct Node *p,*q;
  p=(struct Node *)malloc(sizeof(struct Node));
  q=(struct Node *)malloc(sizeof(struct Node));

  S1: p->f=q;
  S2: q->f=p;
  c1: foo(p);
  S3: q->f=NULL;
  c2: foo(p);
}
```

Comparison with Context Insensitive: In Context Insensitive analysis, we maintain an INmap and OUTmap foreach function, and if the incoming data flow values to the

D_F	p	q	s
p	$\{\varepsilon\}$	$\{f^D\}$	$\{\varepsilon\}$
q	$\{f^D\}$	$\{\varepsilon\}$	$\{f^D\}$
s	$\{\varepsilon\}$	$\{f^D\}$	$\{\varepsilon\}$

I_F	p	q	s
p	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$
q	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$
s	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$

(a) C1:IN[2]

D_F	p	q	s
p	$\{\varepsilon\}$	$\{f^D\}$	$\{\varepsilon\}$
q	$\{f^D\}$	$\{\varepsilon\}$	$\{f^D\}$
s	$\{\varepsilon\}$	$\{f^D\}$	$\{\varepsilon\}$

I_F	p	q	s
p	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$
q	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$
s	$\{(\varepsilon,\varepsilon)\}$	$\{(f^D,\varepsilon),(\varepsilon,f^D)\}$	$\{(\varepsilon,\varepsilon)\}$

(b) C1:INmap

D_F	p	q	s
p	$\{\varepsilon\}$		$\{\varepsilon\}$
q		$\{\varepsilon\}$	
s	$\{\varepsilon\}$		$\{\varepsilon\}$

I_F	p	q	s
p	$\{(\varepsilon,\varepsilon)\}$		$\{(\varepsilon,\varepsilon)\}$
q		$\{(\varepsilon,\varepsilon)\}$	
s	$\{(\varepsilon,\varepsilon)\}$		$\{(\varepsilon,\varepsilon)\}$

(c) C2:IN[0]

Figure 5.2: Data Flow Values at Function Calls

function is a subset of present INmap we just pass the OUTmap without processing the function. Otherwise we update the INmap by merging incoming dataflow values with previous INmap and process the function. We will now compare the INmap and IN array of both context insensitive and context sensitive approaches and see how the shape is effected at statement `S`.

At `c1` the data flow values at the start of function foo are those in Fig. 5.2(a), since shape of p at that call statement is a cycle, so it is assigned to IN[2]. Even if we go by context insensitive it would be the same as shape sensitive given in Fig: 5.2(b) denoted by INmap. At statement `S3` cycle gets killed, hence at `c2`, during shape sensitive analysis, IN[2] will remain the same as that for `c1`, but now IN[0] is newly created. Just by seeing Fig. 5.2(c) we can see that it correctly shows the shape at statement `S` as a TREE. Now if we come to context insensitive, the new INmap during processing of `c2` is the union of previous INmap i.e what's present in Fig. 5.2(b) and current incoming data flow values, which same as that present in Fig. 5.2(c). So even after merging the INmap would same as earlier. Now just compare Fig. 5.2(b) and Fig. 5.2(c). According to Fig. 5.2(b) there is a path from p to q via field f and also from q to$p$ via f but not according to Fig. 5.2(c) This clearly shows that inside the function foo after `c2` shape of p,q and s are identified as cycle for context insensitive approach, but shape sensitive conveys the correct shape i.e TREE.

It was told earlier that size of the IN array for each function should be 3^n when the number of heap parameters are n for a function, but that is not a compulsion. We can vary the size depending on the preciseness and memory handoff. For example even for a function with 3 parameters (all of them are heap pointers) we can keep the size of the

array as 3, where IN[0] is used when any of the parameter is cycle, IN[1] when none of the parameter is a cycle and at least one is a dag and IN[2] when all of them points to a tree. If we go by 3^n then the size of the array would be 9, but now its 3. Though the memory required now is less, information would be accurate in the former case than latter. So the way we choose this depends on the constraints we have in terms of memory and preciseness.

Chapter 6

Implementation And Results

The Interprocedural Field Sensitive Shape Analysis is implemented as a plugin which adds this analysis as one of the passes in GCC. In the first section a few details about GCC internals are given. Then in the last three sections testing strategy, optimizations and results are discussed.

6.1 GCC Internals

PLUGINS: Plugin's make the developer add new features to the compiler without modifying the compiler itself. It is a way of adding, removing and maintaining modules independently. This feature is available from gcc 4.5 and later versions only. Before we discuss plugins, we present some basic information about GCC architecture.

The GCC architecture has many passes in it each being either a GIMPLE, IPA (Interprocedural Analysis) or RTL(Register Transfer Language) pass. So whenever we want to add a pass in GCC we need to talk to the pass manager which is located in three files 'passes.c', 'tree-optimize.c' and 'tree-pass.h' and in some way these files need to be modified. And once modified we need to build the entire GCC so as to get the pass included in GCC.

But as GCC code base is a very large so it would take lot of time to build each time we change our pass source code. At this point plugins make our life easy. Using plugins we will be able to write a shared object (.so) file that can be loaded into GCC and attached to various stages of compilation without touching the GCC source code, hence no need of compiling gcc source every time.

TREE: Tree is the central data structure used by GCC in its internal representation. It can point to a lot of types and to know the particular type we need to refer TREE_CODE macro. Each Tree usually have two fields named TREE_CHAIN and TREE_TYPE. While TREE_CHAIN contains the pointer to next tree (where all the Tree's are arranged in a singly linked list fashion), TREE_TYPE has information about Type or declaration.

Program 6.1: Code to identify pointer statement p = NULL

```
if(is_gimple_assign(stmt))
{
   tree lhsop=gimple_tree_lhs(stmt)
   tree rhsop1=gimple_tree_rhs1(stmt);
   tree rhsop1 = gimple_assign_rhs1 (stmt);

   int lhsCode=TREE_CODE(lhsop);
   int rhsCode=TREE_CODE(rhsop1);

   if((lhsCode==VAR_DECL || lhsCode==PARM_DECL) && rhsCode1==INTEGER_CST)
     {
        if(POINTER_TYPE_P(TREE_TYPE(lhsop)) &&
             (TREE_CODE(TREE_TYPE(TREE_TYPE(lhsop))) == RECORD_TYPE))
          return TRUE;

        return FALSE;
     }
}
```

GIMPLE: Our analysis is performed on the GIMPLE statements which are generated by gcc in its compilation process. Whenever GCC receives a source file say C source code, the GCC frontend invokes the gimplifier for each function which converts the source code to GIMPLE, which is understood by language independent parts of the compiler. Its actually a 3-address representation with at max one load/store per statement, with memory loads only in RHS and store in LHS of assignment statements.

All the GIMPLE statements that are present in a basic block are in the form of a doubly linked list. Any manipulation to be done on them require iterators provided by GCC. Our analysis is written as an Inter procedural gcc plugin which operates on the callgraph. In our analysis we need to identify whether a particular statement is one among the basic pointer statements. The example code in Program 6.1 gives us the details about identifying heap manipulation statement `p = NULL;`. Similarly the other types of pointer statements are also identified.

6.2 Implementation

Lets have an overview of the plugin that inserts this analysis as a pass in GCC. First it finds out what are the heap pointers present in the input program and then accumulates information about its properties like its type, to which struct or union its pointing, fields present in that data type etc. Along with heap pointers, field pointers are also identified whose properties are stored. Field pointers are pointers to some struct or union but is also a member variable of some struct or union. After this we parse all the GIMPLE statements one by one and check if it is one of the basic statements. If the identified statement is a pointer assignment statement a new dummy statement (as discussed in

Chapter 3) is inserted before it. Also we modify the Control flow graph by inserting callblocks and return blocks if necessary. Since the data flow values may change for any of those statements, whenever any of those is encountered GEN and KILL are evaluated, followed by calculation of OUT from IN, GEN and KILL in the usual way.i.e

$$OUT = GEN \cup (IN - KILL)$$

All the equations for GEN and KILL of each statement are mentioned in Fig. 3.4. As we have modified the Control flow graph initially before returning we restore the CFG to its original form. The below pseudo code gives the flow of the implementation. In the

```
begin
  gatherHeapandFieldPointers();
  preprocess_CFG();
  shapeAnalysis();
  restore_CFG();
end
```

function shapeAnalysis the actual identification of shape is done. This is implemented as a worklist based interprocedural analysis, so whenever the worklist goes empty the analysis is stopped.

Now we will discuss the data structures used to contain the data flow values. The Direction Matrix and Interference Matrix are represented as an adjacency matrix with each cell being a pointer to nested structures. These were designed in such a way to handle all the possible values that can be present in each cell. The boolean equations are represented as character strings. This representation leads to huge memory consumption as we need to store them at each program point. Also the size of equation grows with the program, thus taking more time to evaluate. We have tried to resolve these problems to some extent by performing some memory and time optimizations.Next we discuss some memory and time optimizations performed.

6.2.1 Storing of boolean equations

Consider the control flow graph in Fig: 6.1(a) which represents a program containing if-else statements.

$$\begin{aligned} IN_{eq}(S1) &= OUT_{eq}(bb0) \\ IN_{eq}(S2) &= OUT_{eq}(bb0) \\ OUT_{eq}(S1) &= GEN_{eq}(S1) \cup (IN_{eq}(S1) - KILL_{eq}(S1)) \\ OUT_{eq}(S2) &= GEN_{eq}(S2) \cup (IN_{eq}(S2) - KILL_{eq}(S2)) \\ OUT_{eq}(bb1) &= OUT_{eq}(S1) \\ OUT_{eq}(bb2) &= OUT_{eq}(S2) \\ IN_{eq}(bb3) &= OUT_{eq}(bb1) \cup OUT_{eq}(bb2) \end{aligned}$$

Both $OUT_{eq}(bb1)$ and $OUT_{eq}(bb2)$ have a copy of $OUT_{eq}(bb0)$ in them, so $IN_{eq}(bb3)$ also has two copies of it. This may seem very little amount of redundancy but actually for a large program this would become a very big problem.

Instead of storing two copies of the same equation we can save memory by storing the equation at some place and just store pointers to that equation. So now both $OUT_{eq}(S1)$ and $OUT_{eq}(S2)$ will not have the copies of $OUT_{eq}(bb0)$ in it but just the pointers to them. For this optimization to occur we should keep all the data flow boolean equations at each statement. But just doing this change isn't enough. Lets look at Fig: 6.1(b) for the issue with this. This figure shows us a small part of CFG of a program containing while loop.

First time we process `S1`, `S2`,as discussed just above, we will be keeping their corresponding boolean equations. There is a very good chance that boolean equation at `S2` has a pointer to the boolean equation at `S1`. Now this information is passed to bb1 which gets passed to S1 itself. As we are storing the equation again we will be overwriting the already present boolean equations with the current incoming equations, causing loss of information. The similar case can occur in recursive programs too. In order to avoid this we have to keep versions of this equations, and keep track of the version number when pointing to boolean equations. This change significantly reduces the amount of memory consumption.

6.2.2 Time optimization

Initially when we just stored the boolean equation as it is and for that the time taken for the analysis of merge_recur to complete was **4 hrs**. But later we realized that, during the evaluation of boolean equation we are converting it to postfix, it is effective to store the postfix equation itself instead of infix. After this change the time for the analysis of merge_recur reduced to **2 hrs**.

We have just seen in the above subsection that there may be repetitions of terms in the

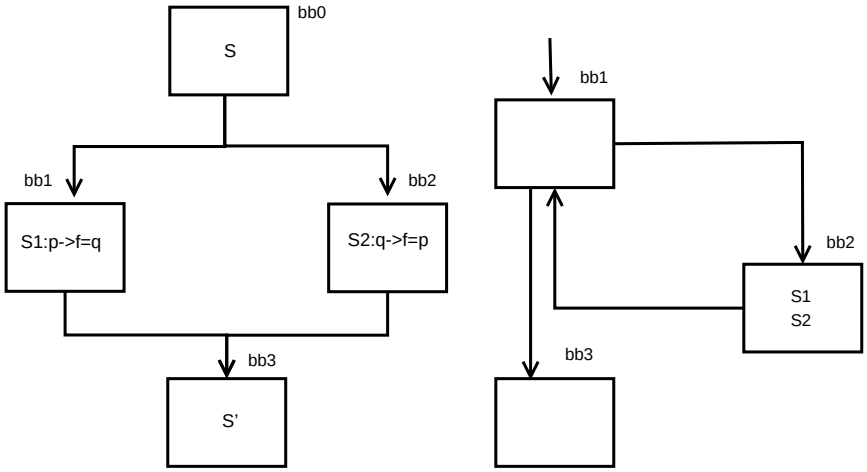


Figure 6.1: CFG example

boolean equations when conditional, looping or recursion is present in programs. Careful analysis of the equation for some sample programs made us notice that the time taken to evaluate these repeated terms can be cut down if we store their results in some hash table and reuse whenever repetition occurs. This has reduced the time taken drastically, and the analysis of merge_recur comes down to **35 seconds**.

6.3 Testing Strategy

We all know how important testing is in order to determine whether we are meeting our required results. In our analysis with the source code spanning close to 12000 LOC and a vast range of possible testcases, an efficient testing strategy is a must.

We have written a script that could generate us many unit testcases depending on the number of statements that we want to have in our test case. A sample template of how our testcase looks is presented in Program 6.2

It is at those dotted lines that our pointer statements will be inserted. You could see in the template that the number of heap pointers are 2 and number of field pointer are 2. With these properties the number of possibilities for each statement is 42, some of the possible statements are $p \rightarrow f = q$,$p \rightarrow g = q \rightarrow f$, $p \rightarrow f = malloc()$, $p = NULL$ and so on. If we increase the number of heap pointers or field pointers the possibilities for each statement would further increase. The number of test cases generated for different number of statements is given in the Table. 6.3 Once we have generated the testcases we run GHIYA's analysis and our analysis and compare the results for each. By comparison we mean comparing the shape of each pointer after each GIMPLE statement. Based on the comparison we put the test case in one of the three categories: PASS, SAFE, FAIL/ACCURATE.

Program 6.2: Unit Testcase Template

```
#include <stdlib.h>
int main()
{
        typedef struct _node node;
        struct _node {
               node* f;
               node* g;
       };

        node* p = (node*)malloc(sizeof(node));
        node* q = (node*)malloc(sizeof(node));

       /* HEAP MANIPULATION STARTS */
       .......
       .......
       /* HEAP MANIPULATION ENDS */

}
```

No Of Stmts	TestCases Generated
1	42
2	1764
3	74088

Table 6.1: No Of TestCases

- PASS : GHIYA's analysis and our analysis gives the shape information at all statements
- SAFE : GHIYA's analysis is more accurate than us, .i.e for example if our analysis infers the shape as CYCLE, then Ghiya will infer more precise shapes like DAG or Tree.
- FAIL/ACCURATE : Our analysis gives less conservative shapes than GHIYA. If this happens then there could be two scenarios: either we are giving accurate results or we are giving incorrect results. All the testcases present in this case are checked manually we ensure that we are not getting any fail cases.

6.4 Results

This section contains details about how would the Field sensitive analysis performs in terms of accuracy and performance when run on different benchmarks. We also give results when this analysis is run on unit test cases. Every time we compare the results with Ghiya method [GH96]. Ghiya's shape analysis is also implemented as a GCC plugin

	TestCases	Pass	Safe	Accurate
Before	ml-1	42	0	0
	ml-2	1572	24	168
	ml-2-if	1764	0	0
	ml-2-while	1108	452	204
	ml-3	51854	6864	15370
After	ml-1	42	0	0
	ml-2	1612	0	152
	ml-2-if	1764	0	0
	ml-2-while	1452	0	312
	ml-3	58444	0	16202

Table 6.2: Unit test cases results

which does context insensitive shape analysis. First we will look at how the analysis performs on the unit test cases (about which we explained in earlier section) and later we will show its performance on List benchmarks followed by one of the olden benchmark. The configuration of the machine used for the generation of results are 2 GB RAM, Pentium Dual Core, 2.10Ghz.

Unit Test cases: The details about these Unit Test cases are already given in the previous section. All these test cases are compared with Ghiya's analysis. Here a comparison is done on how the results have varied before and after all the enhancements mentioned in Chapter 3 were included. They are given in the Table 6.2.

We have already mentioned about the script used to generate these unit test cases, these test cases can have any number of heap manipulation statements as we desire. All those test cases with one heap manipulation are said to be ml-1, those with two as ml-2 etc. Referring to template Program 6.2, in the area where the heap manipulation statements are to be inserted, for ml-while and ml-if type test cases, while and if conditional statements are put. The template for these two are given below. The statement p `= null` was added just to find the shape after these conditionals are executed.

```
statement_1
while (condition){
  statement_2
}
p = null;
```
(a)ml-2-while

```
if (condition)
  statement_1
else
  statement_2
p = null;
```
(b)ml-2-if

The meaning of terms `Safe`,`Pass`,`Accurate` are already explained in the previous section,but still just to reiterate.

- `Safe` : The shape inferred by Ghiya's analysis is more precise than that inferred by Sandeep's

- `Accurate`: The shape inferred by Sandeep's analysis is more precise than that inferred by Ghiya's
- `Pass` : The shape inferred is same by both the analysis.

When we look at the ml-2 results the number of `Accurate` cases were more initially than there were after the enhancements. Though this was the case here we were able to reduce the `Safe` cases to 0 sacrificing some of the `Accurate` cases. One such case is already mentioned in the Chapter 3's last part about Information passed to successors. In ml-2-while there is a significant improvement in how the results turned out, the same can be noticed in ml-3.

Now with all these changes we can say that in all these cases Sandeep's analysis is better than Ghiya's as there is not even a single safe case among all of them. You can clearly understand this by seeing the entries of column `Safe` which contains 0 throughout.

List benchmarks: We have also ran the analysis on Linked List benchmarks, source of which is [Pav10]. These benchmarks contains all the important operations that could be performed on linked list. We find the shape of each heap pointer at each basic statement, then we sum the number of times Tree's are detected, similarly number of times Dag's and Cycle's are detected. This triple (Tree,Dag,Cycle) is used for comparison of results.

The results are present in Table 6.3. The last column tells whether the field sensitive performs better than Ghiya's analysis or not in the particular benchmark. A blank entry means that both gave the same results. Also the meaning of $ and # are explained in the table.

Olden Benchmarks [Car95]: We ran the analysis on the benchmark TreeAdd, there are several other benchmarks which belong to this set, but due to the problem of large memory consumption we were not able to run those benchmarks. Results on this benchmark are given in Table 6.4.

As you can see there are eight of List benchmarks where the results are better than Ghiya's analysis. Also there are seven List benchmarks and the TreeAdd benchmark for which the results are not as good as Ghiya. The main reason for the reduction in the preciseness is the dummy statement. As this statement doesn't kill any information, summarization of nodes takes place. It means that a pointer will be having having data flow values about more than one node. Some of the recursive benchmarks like create_recur and remove_recur when ran on context sensitive interprocedural analysis, gave results same as that of Ghiya, but we were not able to run it on all of the benchmarks because of excessive memory consumption. If we could come up with a memory efficient context sensitive analysis the results would surely improve.

Benchmark	Ghiya's Analysis		Field Sensitive		Result
	Shape	Time(Sec)	Shape	Time(Sec)	
100_create_iter.cpp	(30,0,0)	0	(30,0,0)	0.179	
100_create_recur.cpp	(36,0,0)	0	(22,0,14)	0.147	#
200_delall_iter_create_fixed.cpp	(168,0,0)	0	(168,0,0)	1.385	
200_delall_iter_create_iter.cpp	(63,0,0)	0	(63,0,0)	0.644	
200_delall_recur_create_fixed.cpp	(12,0,0)	0	(12,0,0)	0.093	
200_delall_recur_create_iter.cpp	(56,0,0)	0	(56,0,0)	0.502	
300_insert_iter_create_fixed.cpp	(341,19,0)	0.002	(343,17,0)	8.945	$
300_insert_iter_create_iter.cpp	(161,19,0)	0.002	(163,17,0)	5.018	$
300_insert_recur_create_fixed.cpp	(225,0,27)	0.001	(225,0,27)	3.147	
300_insert_recur_create_iter.cpp	(113,0,27)	0.001	(113,0,27)	2.266	
400_remove_iter_create_fixed.cpp	(314,0,26)	0.001	(318,22,0)	3.787	$
400_remove_iter_create_iter.cpp	(154,0,26)	0.001	(177,3,0)	3.363	$
400_remove_recur_create_fixed.cpp	(243,0,18)	0.001	(234,0,27)	2.932	#
400_remove_recur_create_iter.cpp	(108,0,18)	0.001	(99,0,27)	1.766	#
500_search_iter_create_fixed.cpp	(168,0,0)	0	(168,0,0)	1.225	
500_search_iter_create_iter.cpp	(63,0,0)	0	(63,0,0)	0.596	
500_search_recur_create_fixed.cpp	(208,0,0)	0	(208,0,0)	2.123	
500_search_recur_create_iter.cpp	(88,0,0)	0	(88,0,0)	1.229	
600_append_iter_create_fixed.cpp	(358,6,0)	0.002	(349,15,0)	4.462	#
600_append_iter_create_iter.cpp	(163,6,0)	0.001	(154,15,0)	3.239	#
600_append_recur_create_fixed.cpp	(354,0,38)	0.002	(342,0,50)	5.641	#
600_append_recur_create_iter.cpp	(144,0,38)	0.002	(132,0,50)	3.886	#
700_merge_iter_create_fixed.cpp	(311,0,109)	0.005	(311,0,109)	670.033	
700_merge_iter_create_iter.cpp	(242,0,142)	0.007	(242,0,142)	318.464	
700_merge_recur_create_fixed.cpp	(498,0,114)	0.006	(498,0,114)	33.446	
700_merge_recur_create_iter.cpp	(228,0,114)	0.006	(228,0,114)	22.446	
800_reverse_iter_create_fixed.cpp	(233,0,47)	0.001	(241,0,39)	7.179	$
800_reverse_iter_create_iter.cpp	(83,0,47)	0.001	(91,0,39)	3.707	$
800_reverse_recur_create_fixed.cpp	(499,0,62)	0.004	(489,0,72)	13.408	$
800_reverse_recur_create_iter.cpp	(244,0,62)	0.003	(234,0,72)	8.285	$

$-Field sensitive analysis is more precise #- Ghiya's analysis is more precise

Table 6.3: Comparison On List Benchmark

Benchmark	Ghiya's Analysis		Field Sensitive		Result
	Shape	Time(Sec)	Shape	Time(Sec)	
TreeAdd	(63,0,0)	0.001	(30,0,24)	1.2	#

Table 6.4: Olden: TreeAdd benchmark

Coming to the time taken, we evaluate the boolean equation of each heap pointer and each basic statement; and as the size of the boolean equations also can be large, the time it takes is much more compared to other analysis. Still effort is needed to reduce this by finding any optimizations possible or change the way the boolean equations are represented.

Chapter 7

Conclusion and Future Work

In this report we have suggested several enhancements to Sandeep's work on Field Sensitive analysis which ensure the correctness and increase the accuracy of the analysis. Now after these changes the analysis is better than Ghiya's work in all those unit test cases described. We have also introduced a new analysis called subset based analysis which infers shape based on the subset of fields actually accessed inside a function. This helps us inferring information like a function is traversing/accessing a tree substructure of a cyclic data structure. We also proposed a shape sensitive inter procedural analysis which is in mid way of context sensitive and context insensitive analysis and could possibly balance the memory consumption and preciseness at the same time. We have also performed various optimizations with the aim of decreasing the memory consumption and time for completion. The testing strategy used is exhaustive and has helped a lot in identifying the cases of safe and incorrect results.

There are some benchmarks where the results are not as good as Ghiya's, these are due to the summarization of heap nodes due to the dummy statement. In the future we plan to work on this issue. The analysis has concerns over the amount of memory it takes even after the optimizations performed. We want to address this concern by representing boolean equations in much efficient way.

Appendix A

Analysis of Ghiya [GH96][1]

A large part of the definitions and terms used in this chapter are borrowed from aforementioned paper.At each program point they compute three abstractions that work together to find shape information.For each heap pointer they approximate the attribute shape and direction,interference relationships are approximated for each pair of heap directed pointers. Those three abstractions are given below

Definition 3. *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.*

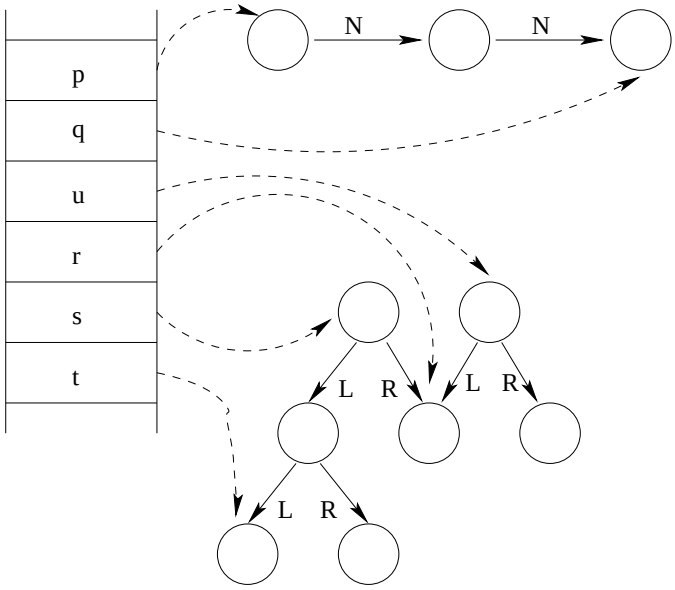
Definition 4. *Given two heap directed pointers p and q, the direction matrix D captures the following relationships between them:*

- $D[p,q] = 1$: *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 5. *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.*

[1]The contents of this section are borrowed from [GH96]



(a) Heap Structure

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`→`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.

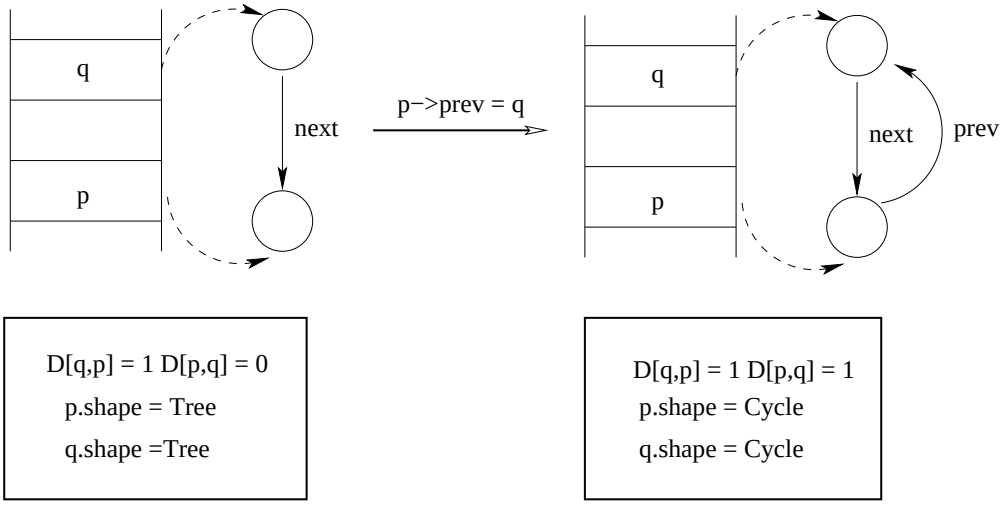


Figure A.2: Example Demonstrating Shape Estimation

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.

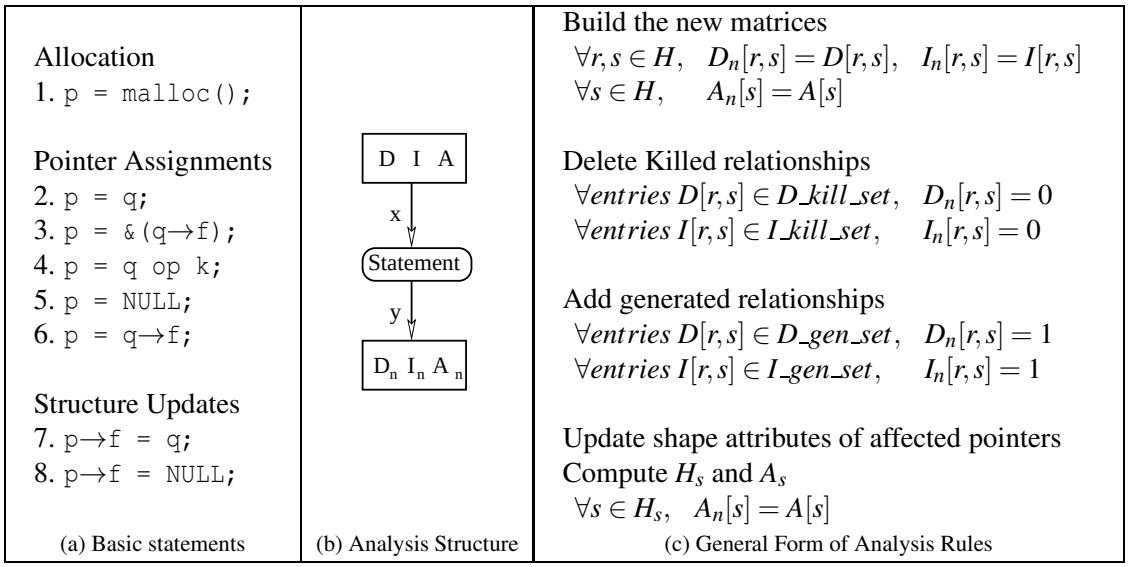


Figure A.3: The Overall Struture of the Analysis

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

Figure A.4: Analysis Rules

Bibliography

[Car95] Martin C. Carlisle. Olden benchmarks, 1995. http://www.martincarlisle.com/olden_benchmarks.tar.Z.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. PLDI '90, pages 296–310, 1990.

[Das11] Sandeep Dasgupta. Precise shape analysis using field sensitivity. Technical report, Master's thesis, IIT Kanpur, 2011. http://www.cse.iitk.ac.in/~karkare/MTP/2010-11/sandeep2010precise.pdf.

[DK12] Sandeep Dasgupta and Amey Karkare. Precise shape analysis using field sensitivity. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1300–1307, New York, NY, USA, 2012. ACM.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer, 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 *POPL '96*, 1996.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. POPL '05, pages 310–323, 2005.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 104–115, New York, NY, USA, 1995. ACM.

[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.

[KMR11] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Lazy pointer analysis. *CoRR*, abs/1112.5000, 2011.

[MHKS08] Mark Marron, Manuel V. Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, pages 245–259, 2008.

[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.

[MS76] Amir Pnuleli Micha Sharir. Two approaches to inter procedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234, 1976.

[Pav10] Viktor Pavlu. Basic operations on linked lists (c++), 2010. `http://www.complang.tuwien.ac.at/vpavlu/2010/list-benchmark.tgz`.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[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.