Program Analysis: Shape Analysis and Slicing

Shouvick Mondal (CS16D004) Arun T (CS16S013)

Scribe week: 10^{th} April - 14^{th} April, 2017
Instructor: Dr. Rupesh Nasre.

Contents

Chapter 1

Shape Analysis (Contd.)

NOTE: Analysis is field-insensitive, path-insensitive, and is performed at the source code level. Allocation and Pointer Assignment statements are described in the previous scribe. However, we enlist all the statements of interest.

Statements of interest

Allocations

- p = malloc()

The underlying assumption is that a call to malloc(...) does not return null.

Pointer Assignments

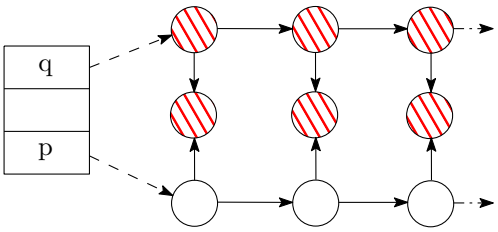
- p = q
- p = q $\rightarrow$ f
- p = &(q $\rightarrow$ f)
- p = q op k
- p = null

Structure Updates

- p $\rightarrow$ f = q
- p $\rightarrow$ f = null

1.1 Notion of Shape

It is to be noted that by the statement q.shape=tree, we are talking about the shape reachable from q and not the actual shape of the data structure. We provide an illustration as follows.



Although the structure to which p and q both point to is a DAG (Directed Acyclic Graph), the shape of p and q is still a Tree. For instance, the heap objects reachable from q are shaded in red and the shape of q is Tree. Similar explanation holds for the shape of p.

1.2 Structure Updates

These statements can drastically change the shape and connectivity of heap structures.

1.2.1 $p \rightarrow f = q$

***Question*:** As we are considering a field-insensitive analysis, is this structure update same as p = q?
***Ans*:** No. We need to change the information of p's points to set and not p.

Kill sets:

- D_Kill = { }
- I_Kill = { }

Gen sets:

- D_Gen = { D[r][s] = 1, if D[r][p] == 1 & D[q][s] == 1 }
- I_Gen = { I[r][s] = 1, if I[r][p] == 1 & I[q][s] == 1 }

Shape:

- if (D[q][p] & D[s][q])
  then:
  s.shape = cycle
- if (D[q][p] & D[s][p])
  then:
  s.shape = cycle
- if (¬D[q][p] & D[s][p] & I[s][q] & q.shape == Tree)
  then:
  s.shape = max(s.shape, DAG)
- if (¬D[q][p] & D[s][p] & q.shape != Tree)
  then:
  s.shape = max(s.shape,q.shape)

Note: The above sets were generated with respect to q (assumed to be non-null). If D[q][q] & I[q][q] is 1, then q is not null.

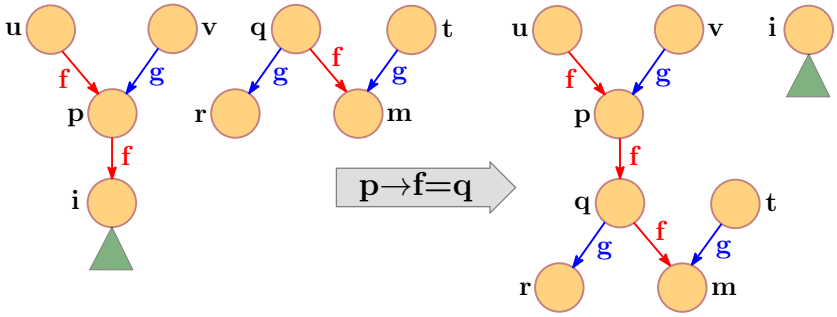


Figure 1.1: Illustration of rule: p→f=q.

***Question*:** With respect to the above Gen sets can the precision still be improved?
***Ans*:** Yes. Let us consider a scenario in which p → f and p → g, and I[r][p] = 1 because of p → g and if we set I[r][s] = 1 with respect to Gen sets, then it leads to imprecision. Making I[r][p] to D[r][p] helps us to improve the precision further.

***Question*:** By making I[q][s] to D[q][s] in Gen sets, can the precision be still improved?
***Ans*:** No. It will lead to loss of information, hence unsound analysis. In Fig. 1.1, if we replace I[q][s] with D[q][s], we would miss the link with the node labelled t.

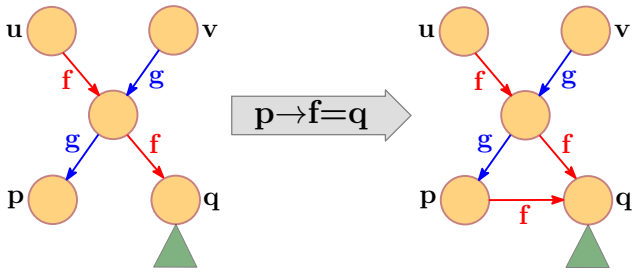


Figure 1.2: Illustration of rule: p→f=q.

The final improved Gen sets with less imprecision will be,

- D_Gen = { D[r][s] = 1, if D[r][p] == 1 & D[q][s] == 1 }
- I_Gen = { I[r][s] = 1, if **D**[r][p] == 1 & I[q][s] == 1 }

With respect to precision, Tree is considered to be the most precise shape and the precision reduces as follows.

$$\textbf{Tree} \succeq \textbf{DAG} \succeq \textbf{Cycle}$$

The maximum operation as applied to shapes are given by the following table. The result of the operation is conservative but it is still sound.

max	**Tree**	**DAG**	**Cycle**
Tree	Tree	DAG	Cycle
DAG	DAG	DAG	Cycle
Cycle	Cycle	Cycle	Cycle

max(shape1, shape2)

1.2.2 p → f = null

Same, as we discussed for the earlier structure update, p → f set is made null and not p.

Kill sets:

- D_Kill = { }
- I_Kill = { }

Gen sets:

- D_Gen = { }
- I_Gen = { }

Shape:

- Shape of p does not have any change.

1.3 Example

For the following program, find the interference, direction and shape.

```
1  void listReverse(List x) {
2    assert("x is an acyclic singly linked list");
3
4    for(y=NULL;x;) {
5      t = y;
6      y = x;
7      x = x->next;
8      y->next = t;
9    }
10   x = y;
11   t = NULL;
12   y = NULL;
13 }
```

Fig. 1.3 shows the corresponding Control Flow Graph (CFG). B_i is the label for block i. The numbering inside a block reflects the ordering of statements in that particular block. For the purposes of illustration[1], we denote statement j of block i as $B_{i,j}$.

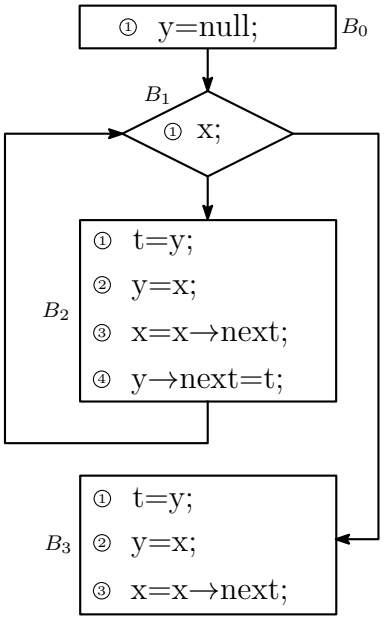
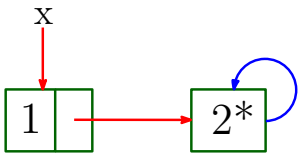


Figure 1.3: Control Flow Graph.



Assumption: Initially x (input) is not null.

Matrix Initialization

Interference			
	x	y	t
x		0	0
y			0
t			

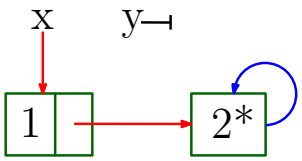
Direction			
	x	y	t
x	1	0	0
y	0	0	0
t	0	0	0

Shape	
x	tree
y	tree
t	tree

Initialization:- $B_{0,1}$: **y = null**

With respect to the rules of inference for statement **p = null**, D[y][y] is killed and y.shape is set to tree.

[1]After we process a statement, the entries of interest in the appropriate matrices appear in bold font. Matrix entries marked with * denote imprecision.

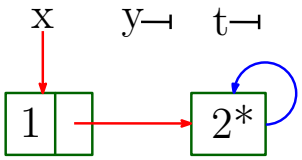


Interference			
	x	y	t
x		0	0
y			0
t			

Direction			
	x	y	t
x	1	0	0
y	0	**0**	0
t	0	0	0

Shape	
x	tree
y	**tree**
t	tree

Iteration 1:- $B_{2,1}$: $\mathbf{t = y}$



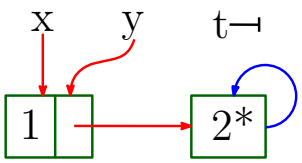
With respect to the rules of inference for the statement $\mathbf{p = q}$, D and I matrix are not set because y is pointing to null. The shape of t is same as that of y.

Interference			
	x	y	t
x		0	0
y			**0**
t			

Direction			
	x	y	t
x	1	0	0
y	0	0	**0**
t	0	**0**	**0**

Shape	
x	tree
y	tree
t	**tree**

Iteration 1:- $B_{2,2}$: $\mathbf{y = x}$



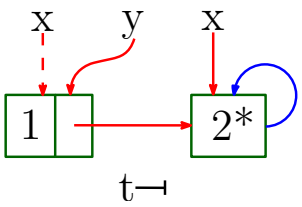
With respect to the rules of inference for statement $\mathbf{p = q}$, x is not null and I[x][y], D[x][y], D[y][x] and D[y][y] are set to 1. The shape of y is same as that of x, which is tree.

Interference			
	x	y	t
x		**1**	0
y			0
t			

Direction			
	x	y	t
x	1	**1**	0
y	**1**	**1**	0
t	0	0	0

Shape	
x	tree
y	**tree**
t	tree

Iteration 1:- $B_{2,3}$: $\mathbf{x = x \rightarrow next}$



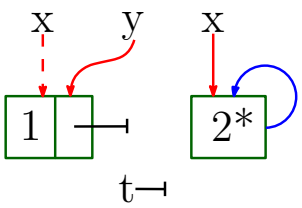
With respect to the rules of inference for statement $\mathbf{p = q \rightarrow f}$, D[x][y] is killed and the shape of x remains same, which is tree. As our analysis is field-insensitive, D[x][y] is set to 1 and it leads to imprecision.

Interference			
	x	y	t
x		1	0
y			0
t			

Direction			
	x	y	t
x	**1**	**1***	0
y	1	1	0
t	0	0	0

Shape	
x	**tree**
y	tree
t	tree

Iteration 1:- $B_{2,4}$: $\mathbf{y \rightarrow next = t}$



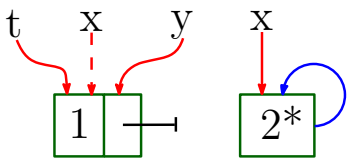
With respect to the rules of inference for statement $\mathbf{p \rightarrow f = q}$, as t is null the points-to set of y is made null and the shape of y remains the same.

Interference			
	x	y	t
x		0	0
y			**0**
t			

Direction			
	x	y	t
x	1	1	0
y	1	1	**0**
t	0	0	0

Shape	
x	tree
y	**tree**
t	tree

Iteration 2:- $B_{2,1}$: $\mathbf{t = y}$



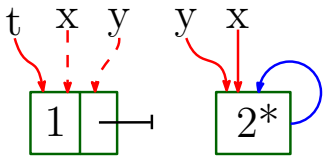
With respect to the rules of inference for statement $\mathbf{p = q}$, D[t][y],D[y][t] and I[y][t] is set to 1. Due to earlier imprecision by $\mathbf{x = x \rightarrow next}$, more imprecisions D[x][t],D[t][x] and I[x][t] are added.

Interference			
	x	y	t
x		1	$\mathbf{1}^*$
y			**1**
t			

Direction			
	x	y	t
x	1	1	$\mathbf{1}^*$
y	1	1	**1**
t	$\mathbf{1}^*$	**1**	**1**

Shape	
x	tree
y	tree
t	**tree**

Iteration 2:- $B_{2,2}$: $\mathbf{y = x}$



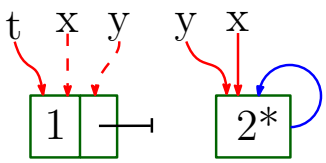
Applying the same rule of $\mathbf{p = q}$, the affected values appear in bold font in the appropriate matrices.

Interference			
	x	y	t
x		1	**1**
y			**1**
t			

Direction			
	x	y	t
x	1	**1**	**1**
y	1	1	**1**
t	**1**	**1**	**1**

Shape	
x	tree
y	**tree**
t	tree

Iteration 2:- $B_{2,3}$: $\mathbf{x = x \rightarrow next}$



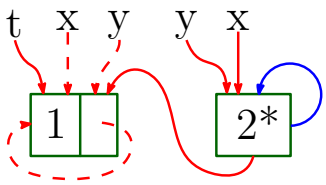
Applying the same rule of $\mathbf{p = q \rightarrow f}$, the affected values appear in bold font in the appropriate matrices.

Interference	x	y	t
x		**1**	**1**
y			1
t			

Direction	x	y	t
x	1	**1**	**1**
y	**1**	1	1
t	**1**	1	1

Shape	
x	**tree**
y	tree
t	tree

Iteration 2:- $B_{2,4}$: $\mathbf{y \rightarrow next = t}$



With respect to the rules of inference for statement $\mathbf{p \rightarrow f = q}$, the affected values appear in bold font in the appropriate matrices. Due to the imprecision, the shape of x,y, and t are changed to cycle.

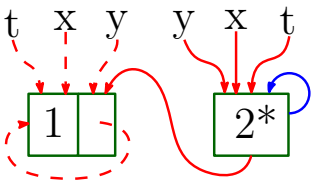
Interference	x	y	t
x		1	1
y			1
t			

Direction	x	y	t
x	1	1	1
y	1	1	**1**
t	1	1	1

Shape	
x	**cycle**
y	**cycle**
t	**cycle**

NOTE: The following is the fixed-point iteration for the for loop.

Iteration 3:- $B_{2,1}$: $\mathbf{t = y}$

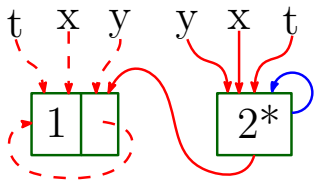


Interference	x	y	t
x		1	1
y			1
t			

Direction	x	y	t
x	1	1	1
y	1	1	1
t	**1**	**1**	**1**

Shape	
x	cycle
y	cycle
t	cycle

Iteration 3:- $B_{2,2}$: $\mathbf{y = x}$

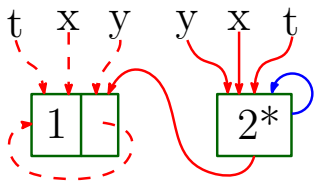


Interference			
	x	y	t
x		1	1
y			1
t			

Direction			
	x	y	t
x	1	$\mathbf{1}^*$	1
y	$\mathbf{1}^*$	1	$\mathbf{1}^*$
t	1	$\mathbf{1}^*$	1

Shape	
x	cycle
y	cycle
t	**cycle**

Iteration 3:- $B_{2,3}$: $\mathbf{x = x \rightarrow next}$

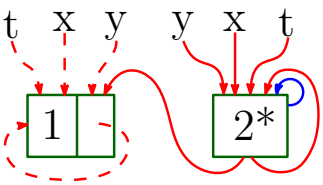


Interference			
	x	y	t
x		1	1
y			1
t			

Direction			
	x	y	t
x	1	**1**	1
y	1	1	1
t	1	1	1

Shape	
x	**cycle**
y	cycle
t	cycle

Iteration 3:- $B_{2,4}$: $\mathbf{y \rightarrow next = t}$

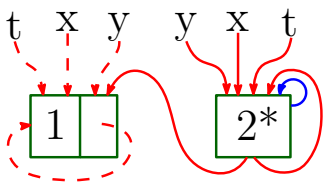


Interference			
	x	y	t
x		1	1
y			1
t			

Direction			
	x	y	t
x	1	1	1
y	1	1	**1**
t	1	1	1

Shape	
x	cycle
y	**cycle**
t	**cycle**

Termination block:- $B_{3,1}$: **x = y**

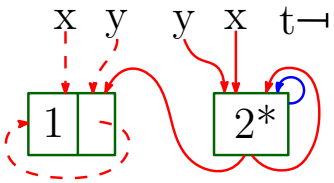


Interference			
	x	y	t
x		1	1
y			1
t			

Direction			
	x	y	t
x	1	1	1
y	1	1	1
t	1	1	1

Shape	
x	**cycle**
y	cycle
t	cycle

Termination block:- $B_{3,2}$: **t = null**



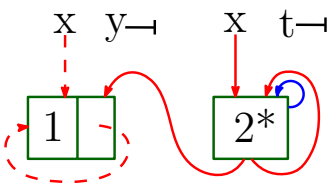
With respect to the rules of inference for statement **p = null**, the killed values are highlighted in the matrix set and t.shape is set to tree.

Interference			
	x	y	t
x		1	**0**
y			**0**
t			

Direction			
	x	y	t
x	1	1	**0**
y	1	1	**0**
t	**0**	**0**	**0**

Shape	
x	cycle
y	cycle
t	**tree**

Termination block:- $B_{3,3}$: **y = null**



With respect to the rules of inference for statement **p = null**, the killed values are highlighted in the matrix set and y.shape is set to tree.

Interference			
	x	y	t
x		**0**	0
y			0
t			

Direction			
	x	y	t
x	1	**0**	0
y	**0**	**0**	0
t	0	0	0

Shape	
x	cycle
y	**tree**
t	**tree**

Result: Initially the shape of x is "Tree", but at the end of the analysis, we found it to be "Cycle".

Chapter 2

Slicing

Introduction

Program slicing is the computation of the set of programs statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.

A slice is a subset of program statements (possibly) relevant to an event of interest. Those statements need not be consecutive. Slicing is a may analysis. These set of statements may or may not have impact on each other. Slicing is illustrated now with a toy example.

Example:

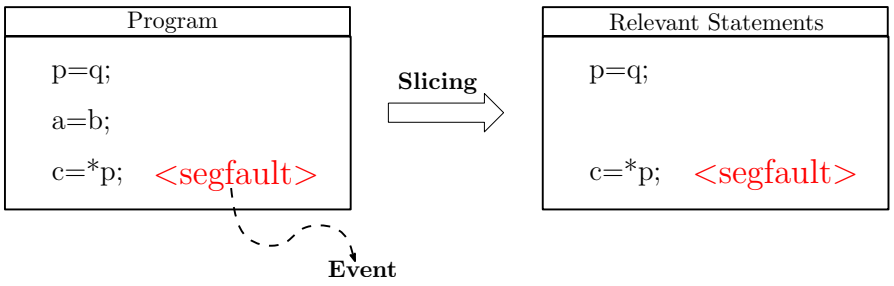


Figure 2.1: Slicing: event is possibility of segmentation fault at line 3.

Fig. 2.1 shows a piece of code containing three statements. The event that we are interested in is the possibility of segmentation fault at **c = *p** (line 3). The segfault at line 3 might occur if the pointer p is null or uninitialized. So, line 3 has a use of p and we only need to track the def of p that may affect this use of p. We

see that the def of p at line 1 may affect line 3, therefore we include it as a part of the slice. Note that the slicing criteria (program point) is also a part of the slice.

Now the code is modified by introducing a new statement **b = fun(d)** as shown in Fig. 2.2. Now the slicing criteria changes. The program slice for this criteria is shown in Fig. 2.2 (right).

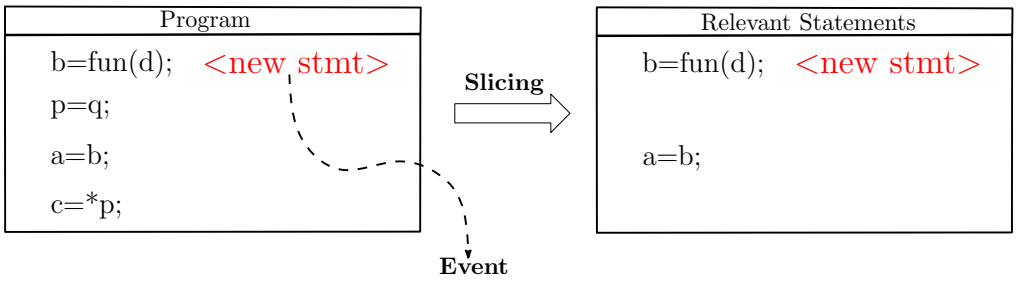


Figure 2.2: Slicing: event is introduction of a new statement **b = fun(d)** in Fig. 2.1 (left).

***Question*:** How slicing differs from def-use chains?
***Ans*:** Def-use chain takes care of data flow facts only, whereas slicing considers both data and control flow facts.

***Question*:** Should the order of statements be taken care of (preserved) after slicing the program code?
***Ans*:** Yes. Control flow is maintained.

2.1 Applications of Slicing

- **Program understanding / debugging** - Program comprehension or program understanding or source code comprehension is a domain of computer science concerned with the ways software engineers maintain existing source code. The cognitive and other processes involved are identified and studied. The results are used to develop tools and training. Software maintenance tasks have five categories: adaptive maintenance, corrective maintenance, perfective maintenance, code reuse, and code leverage.

- **Program re-structuring -** Initially, a program may contain a single module which implements multiple functionalities. Slicing helps in re-structuring the program into separate modules, in which each module code takes care of certain functionality.

- **Program differencing -** To see the slicing difference between two programs.

- **Test Coverage -** Helps us to include all test cases in every scenario to test the program functionality.
- **Model checking -** Use of formal methods for program verification and correctness.

2.2 Slicing in Programs: More Examples

Code 1 implements two functionalities: (1) *Line counting*, and (2) *Character counting*, in a file provided as input to the program.

Code 1: Line and Character counting

```
1 extern void scanLine(FILE *f, bool *eof, int *nread);
2 void lineCharCount(FILE *f) {
3   int nlines = 0;
4   int nchars = 0;
5   int nread;
6   bool eof = false;
7
8   do {
9     scanLine(f, &eof, &nread);
10    ++nlines;
11    nchars += nread;
12  } while (!eof);
13
14  printf("nlines=%d\n", nlines);
15  printf("nchar=%d\n", nchar);
16 }
```

Code 2 and 3 are slices of Code 1, which implement appropriate sub-functionalities. The statements commented in Code 2 and Code 3 do not belong to their respective slices.

Code 2: Line counting

```
1 extern void scanLine(FILE *f, bool *eof, int *nread);
2 void lineCharCount(FILE *f) {
3   int nlines = 0;
4   //int nchars = 0;
5   int nread;
6   bool eof = false;
7
8   do {
```

```
 9      scanLine(f, &eof, &nread);
10      ++nlines;
11      //nchars += nread;
12    } while (!eof);
13
14    printf("nlines=%d\n", nlines);
15    //printf("nchar=%d\n", nchar);
16 }
```

Code 3: Character counting

```
 1 extern void scanLine(FILE *f, bool *eof, int *nread);
 2 void lineCharCount(FILE *f) {
 3   //int nlines = 0;
 4   int nchars = 0;
 5   int nread;
 6   bool eof = false;
 7
 8   do {
 9     scanLine(f, &eof, &nread);
10     //++nlines;
11     nchars += nread;
12   } while (!eof);
13
14   //printf("nlines=%d\n", nlines);
15   printf("nchar=%d\n", nchar);
16 }
```

Now if we look at Code 2 (which has some statements commented out from Code 1), there is a need for a change in the scanLine() function in the form of discarding the third formal parameter, **int *nread**. Code 4 is a modification of Code 2, where scanLine2() replaces scanLine().

Code 4: Line counting (Code 2 modified)

```
1 extern void scanLine2(FILE *f, bool *eof/*, int
    *nread*/);
2 void lineCharCount(FILE *f) {
3   int nlines = 0;
4   //int nchars = 0;
5   //int nread;
6   bool eof = false;
7
```

```
 8   do {
 9     scanLine(f, &eof/*, &nread*/);
10     ++nlines;
11     //nchars += nread;
12   } while (!eof);
13
14   printf("nlines=%d\n", nlines);
15   //printf("nchar=%d\n", nchar);
16 }
```

2.3 Types of Slicing

In the following examples, the codes are commented at appropriate points for the slicing criteria and a very brief explanation. The slicing criteria is denoted as <*line*, *var*>, where *line* is the line number in the program and *var* is the variable in that line.

2.3.1 Forward Slicing

The forward slice at the program point p is the program subset that may be affected by p. We can say forward slicing as a measure to check, which all statements are affected by this selection (slicing) criteria.

Example: For the following program, apply ***forward slicing*** with slicing criteria <***2, sum***>.

```
1 void main() {
2     int sum = 0; //slicing criteria
3     int i = 0;
4
5     while(i < 11) {
6         sum = sum + i;
7         ++i;
8     }
9     printf("sum=%d\n", sum);
10    printf("i=%d\n", i);
11 }
```

***Ans*:** Statements at line no: 2, 6 and 9 may be affected and will be sliced out as a result of forward slicing.

Reasons:

- line no: 2 is included because it is the slicing criteria.
- line no: 6 uses the value of sum defined at line no: 2, hence added to sliced code.
- line no: 9 uses the value of sum defined at line no: 6, as line no: 6 has been included in sliced code, line no: 9 is also added.

Let us try inserting a new statement to the above program and re-evaluate the sliced code.

```
1 void main() {
2     int sum = 0; //slicing criteria
3     sum = 1;  //newly inserted statement
4     int i = 0;
5
6     while(i < 11) {
7         sum = sum + i;
8         ++i;
9     }
10    printf("sum=%d\n", sum);
11    printf("i=%d\n", i);
12 }
```

***Ans*:** Only statement at line no: 2 is now the entire sliced code.

Reasons:

- line no: 2 is included as it is the slicing criteria.
- line no: 3 *re-defines* sum and the definition at line 2 does not propagate further. This makes line no: 7, use the definition of line no: 3 and not the definition of line no: 2. Hence, it is not added to the sliced code. No other statement is affected by this slicing criteria.

2.3.2 Backward Slicing

The backward slice at the program point p is the program subset that may affect p. Backward slicing is mostly used for debugging purposes.

Example: For the following program, apply ***backward slicing*** with slicing criteria <***9, sum***>.

```
1 void main() {
2     int sum = 0;
3     int i = 0;
4
5     while(i < 11) {
6         sum = sum + i;
7         ++i;
8     }
9     printf("sum=%d\n", sum); //slicing criteria
10    printf("i=%d\n", i);
11 }
```

***Ans*:** Statements at line no: 2, 3, 5, 6, 7 and 9 may affect the slicing criteria and will be sliced out as a result of backward slicing.

Reasons:

- line no: 9 is included because it is the slicing criteria.
- line no: 6 & 2, sum value is getting changed and it affects the slicing criteria, hence added to sliced code. Note that as we are doing a static analysis, the while loop may or may not get executed at run time. Therefore, conservatively, we include both line no: 6 & 2.
- line no: 7, 5 & 3, i's value may get added up with sum at line no: 6, which affects the slicing criteria. As a result, i and its involved (affected or affecting) statements got added to the sliced code.

Let us try inserting a new statement to the above program and re-evaluate the sliced code, slicing criteria being <***10, sum***>.

```
1 void main() {
2     int sum = 0;
3     int i = 0;
4
5     while(i < 11) {
6         sum = sum + i;
7         ++i;
8     }
9     sum = 1;  //newly inserted statement
10    printf("sum=%d\n", sum); //slicing criteria
11    printf("i=%d\n", i);
```

```
12 }
```

***Ans*:** Statements at line no: 9 and 10 form the sliced code.

Reasons:

- line no: 10 is included because it is the slicing criteria.
- line no: 9 is included because it is the corresponding def the sum's use at line no: 10. The *re-definition* of sum at line 9 kills all other definitions reaching line no: 9. Hence, no other line i, $(i < 9)$ is included in the sliced code.

2.3.3 Chop Slicing

The chop between program points p and q is the program subset that may be affected by p ***and*** that may affect q. Chopping applies both the techniques of forward and backward slicing and takes the ***intersection*** of the set of statements between them to get the sliced code. One use of this type of slicing is in regression testing of software.

Example: For the following program apply ***chop slicing*** with slicing criteria ***(<2, sum>, <10, i>)***.

```
1  void main() {
2      int sum = 0; //chopping criteria start
3      int i = 0;
4
5      while(i < 11) {
6          sum = sum + i;
7          ++i;
8      }
9      printf("sum=%d\n", sum);
10     printf("i=%d\n", i);  //chopping criteria end
11 }
```

***Ans*:** No statements belong to the sliced code, except line no: 2 and 10.

Reasons:

- line no: 2, 6 and 9 form the forward sliced code, $S_f = \{2, 6, 9\}$.
- line no: 10, 7, 5 and 3 form the backward sliced code, $S_b = \{3, 5, 7, 10\}$.
- the intersection of these two sets, $S_f \cap S_b = \emptyset$.

***Question*:** Why we are going for Chop slicing? Can it be replaced by forward or backward slicing?
***Ans*:** Chopping does slicing on a part of program code, whereas forward/backward slicing does it on the whole program code. Chopping helps in regression testing by identifying statements that *may affect* and *may be affected* by introducing a new statement in the code. Also, chopping helps increasing the precision of the analysis by performing set intersection and removing extraneous statements. Let us look at the following code and perform chop slicing.

```
1 int *q = (int *)malloc(sizeof(int));
2 int *p = NULL;
3 q = NULL; //newly added statement to the program code
4 p = q;
5 r = p;
6 printf("%d",*p); //do chop slicing to find which all
    statements are required if segmentation fault occurs
    at this point
```

Forward slicing contains sliced code of line no: 3, 4, 5 & 6 ($\{3, 4, 5, 6\}$). Backward slicing contains sliced code of line no: 3, 4 & 6 ($\{3, 4, 6\}$). Intersection of these two sets results in sliced code of line no: 3, 4 & 6.

No, it cannot be replaced by forward or backward slicing.

2.3.4 Static and Dynamic Slicing

Slicing is said to be static, if it is performed at compile time without any knowledge of the program input. Slicing is dynamic if it is done at execution time or compile time, relying on specific test cases.

Static Slicing

- There could be multiple valid slices for a given criteria, depending upon analysis precision.
- There exists at least one slice for a criteria (the program itself).
- The analysis implicitly assumes that the program halts.
- Finding a statement-minimal slice is impossible (halting problem). If we can do this, we will be able to find out statically whether a loop terminates (halts) or not.

NOTE 1: If we formulate *Slicing* as a *Data-Flow Analysis* (DFA), then consider the definition of a variable q in some basic block. Now, if we are doing forward slicing, we need to keep track of paths along which this definition of q might reach

some other basic block. If at least one such path exists in the associated control-flow graph, we conservatively include this definition of p in the program slice, even if other paths kill this definition (q is re-defined along other paths).

NOTE 2: Language features such as functions, unstructured control flow, composite data types, pointers, and concurrency require specific extensions to the slicing algorithms.

Chapter 3

A Note on Parallelization

Parallelization is discussed in detail in the following scribes.

- Scribe week: 13^{th} March, 2017. Authors: Mani Raja & Puneet Saraf.
- Scribe week: 20^{th} March, 2017. Authors: Indrani Roy & Sayantan Ray.

Here we mention one possible prototype and definition of the DCAS (Double Compare-And-Swap) operation only.

Code for Double Compare-And-Swap (DCAS)

```
1 int DCAS(int *addr1, int *addr2, int old1, int old2,
     int new1, int new2)
2 {
3   <begin atomic>
4   if((*addr1 == old1) && (*addr2 == old2))
5   {
6     *addr1 = new1;
7     *addr2 = new2;
8     return(TRUE);
9   }
10  else
11  {
12    return(FALSE);
13  }
14  <end atomic>
15 }
```

Bibliography

[1] Rakesh Ghiya , Laurie J. Hendren, *Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C*, Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.1-15, January 21-24, 1996, St. Petersburg Beach, Florida, United States.

[2] M. Weiser. *Program slicing.* In Proceedings of the 5th international conference on Software engineering, ICSE 81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[3] Dr. Rupesh Nasre's slides for the course *CS6843: Program Analysis*, Jan–May, 2017, at IIT Madras, India.

[4] http://gregorio.stanford.edu/papers/non-blocking-osdi/slides/sld005.htm

[5] https://en.wikipedia.org/wiki/Program_comprehension