Geometric Encoding

Forging the High Performance Context Sensitive Points-to Analysis for Java

Xiao Xiao, Charles Zhang

The prism research group, HKUST

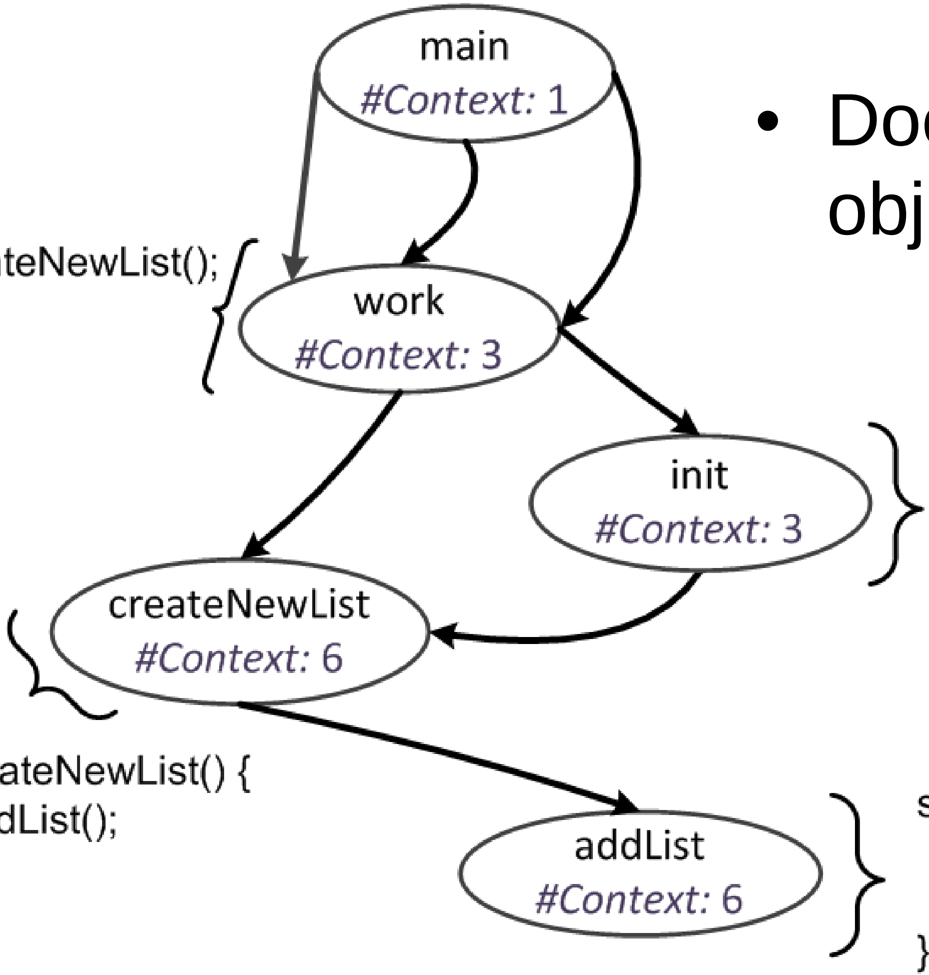
Definition

- Points-to analysis:
  - A process to determine the set of variables that are possibly pointed-to by every pointer.

Points-to Analysis

- Does *list* point to object *o* ?

```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```



```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

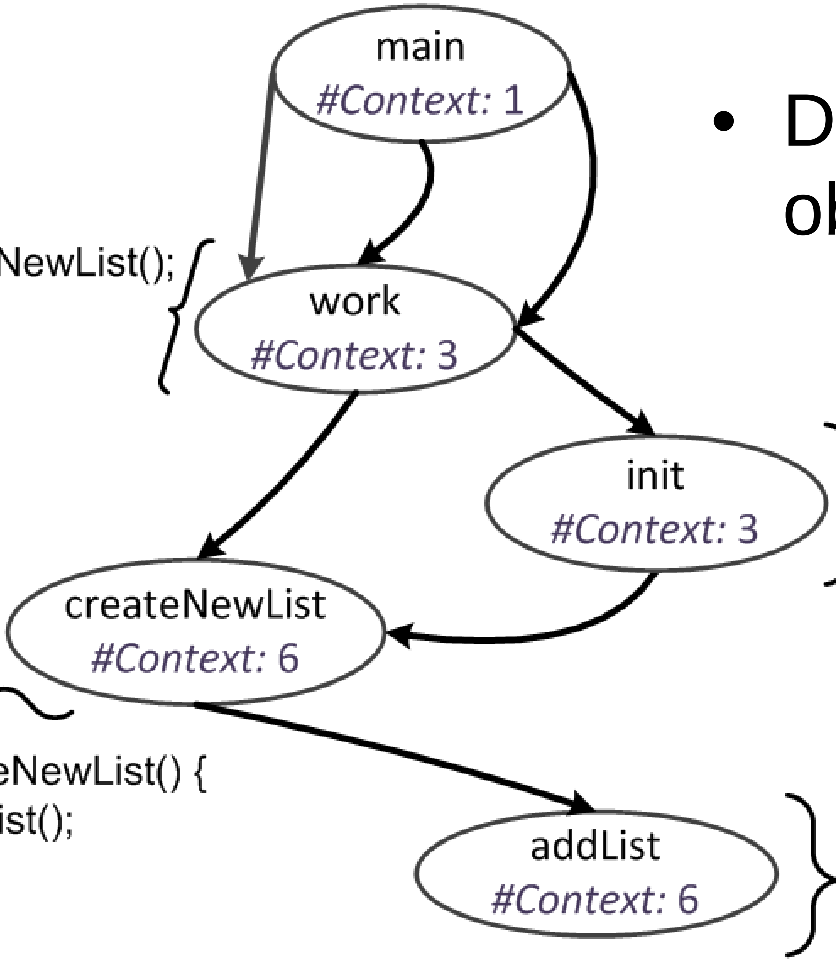
```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Points-to Analysis

- Does *list* point to object *o* ?

```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```



```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

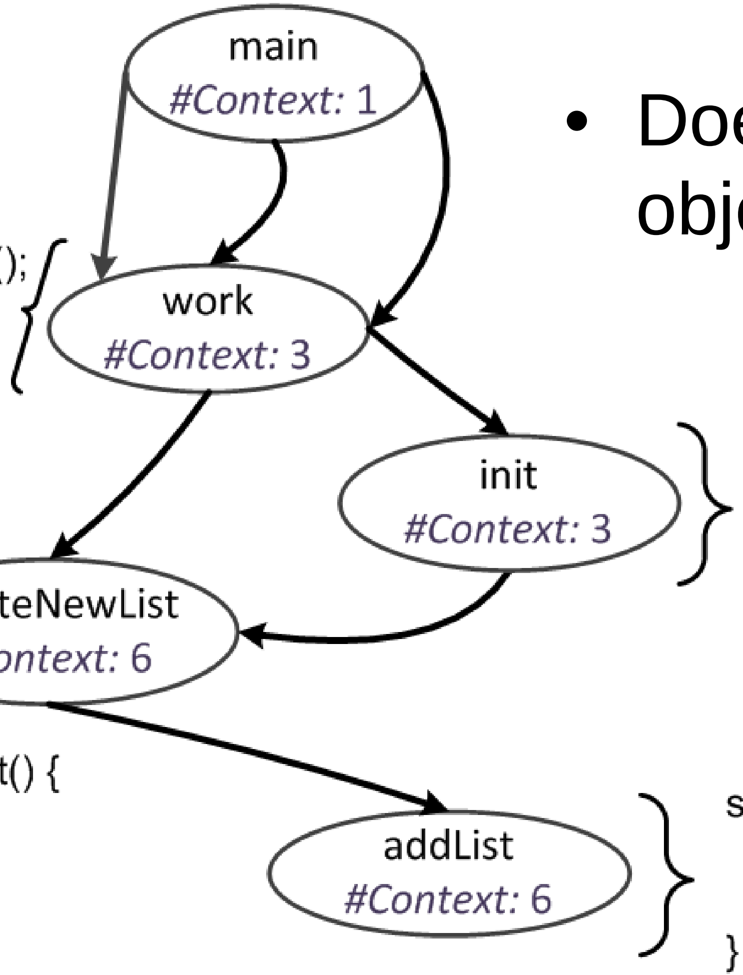
```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

香港科技大學
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Points-to Analysis

- Does *list* point to object *o* ?

```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```



```
static Node init() {
   gList = createNewList();
   return gList;
}
```

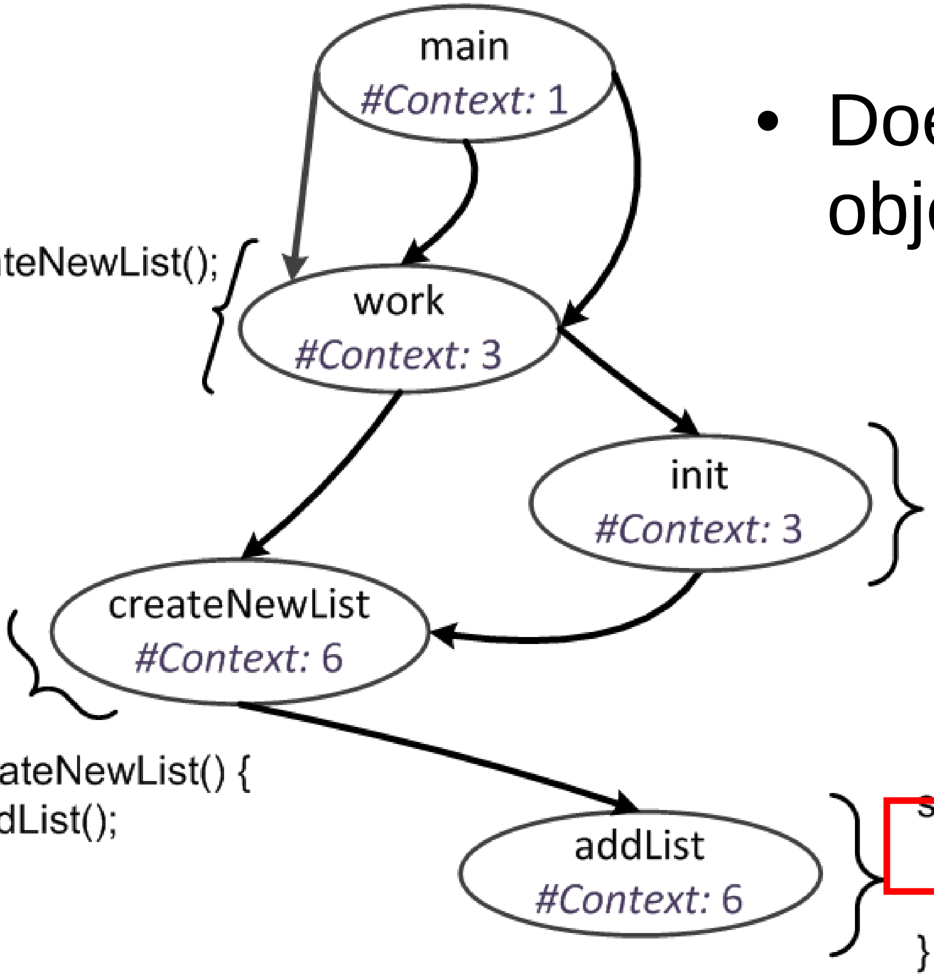
```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

Points-to Analysis

- Does *list* point to object *o* ?

```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```



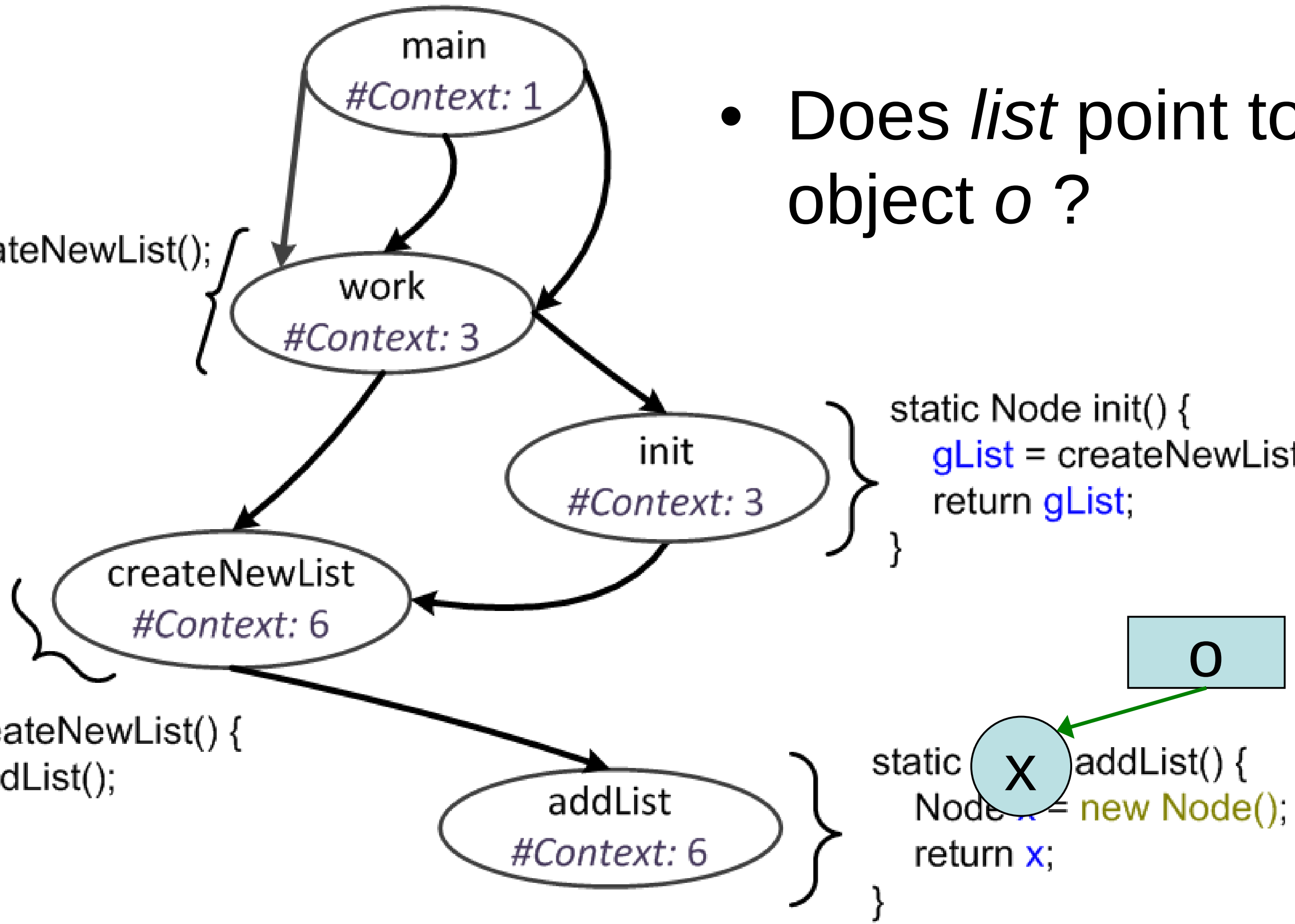
```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Points-to Analysis

- Does *list* point to object *o* ?



```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```

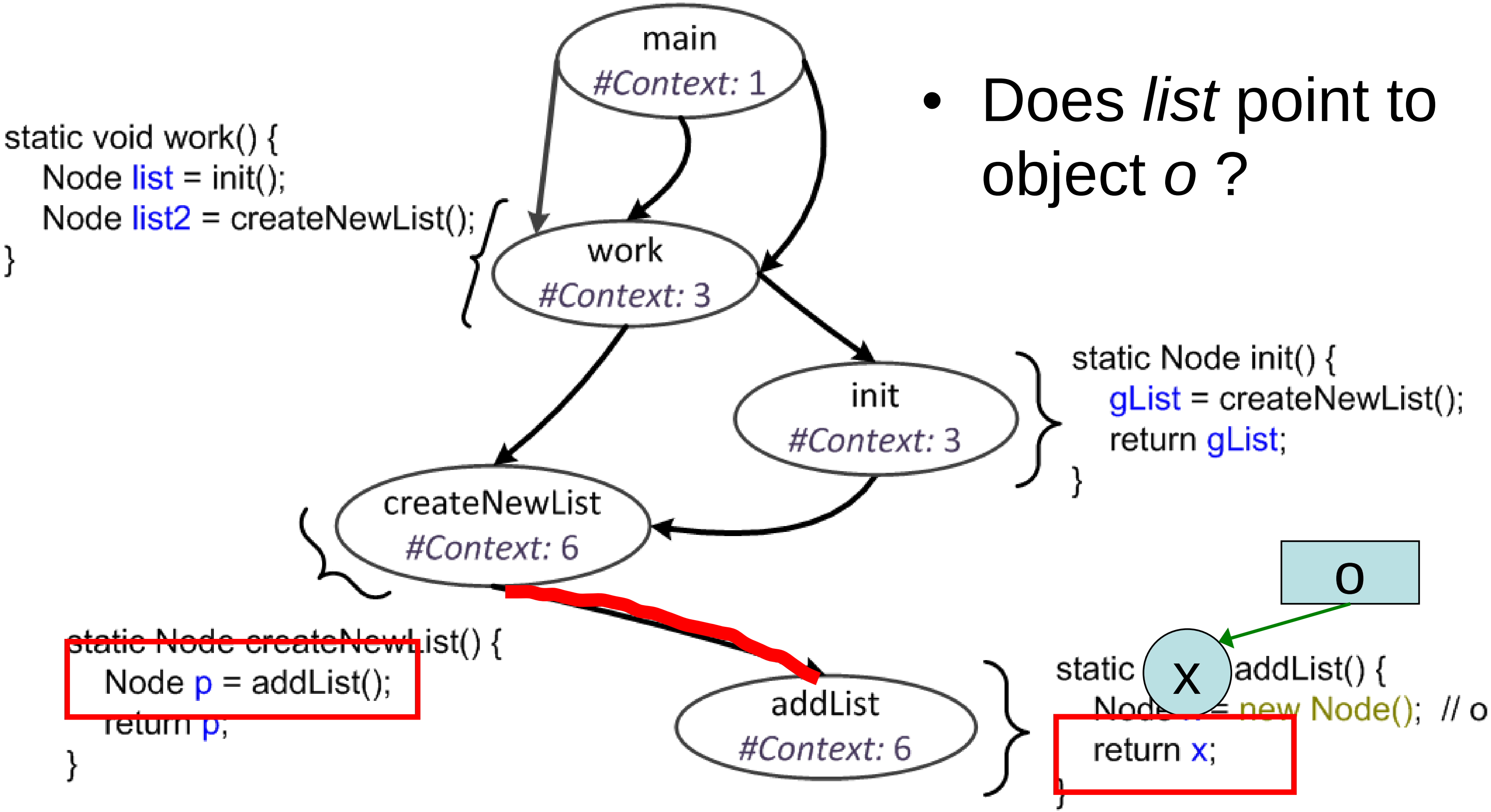
```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

Points-to Analysis

- Does *list* point to object *o* ?



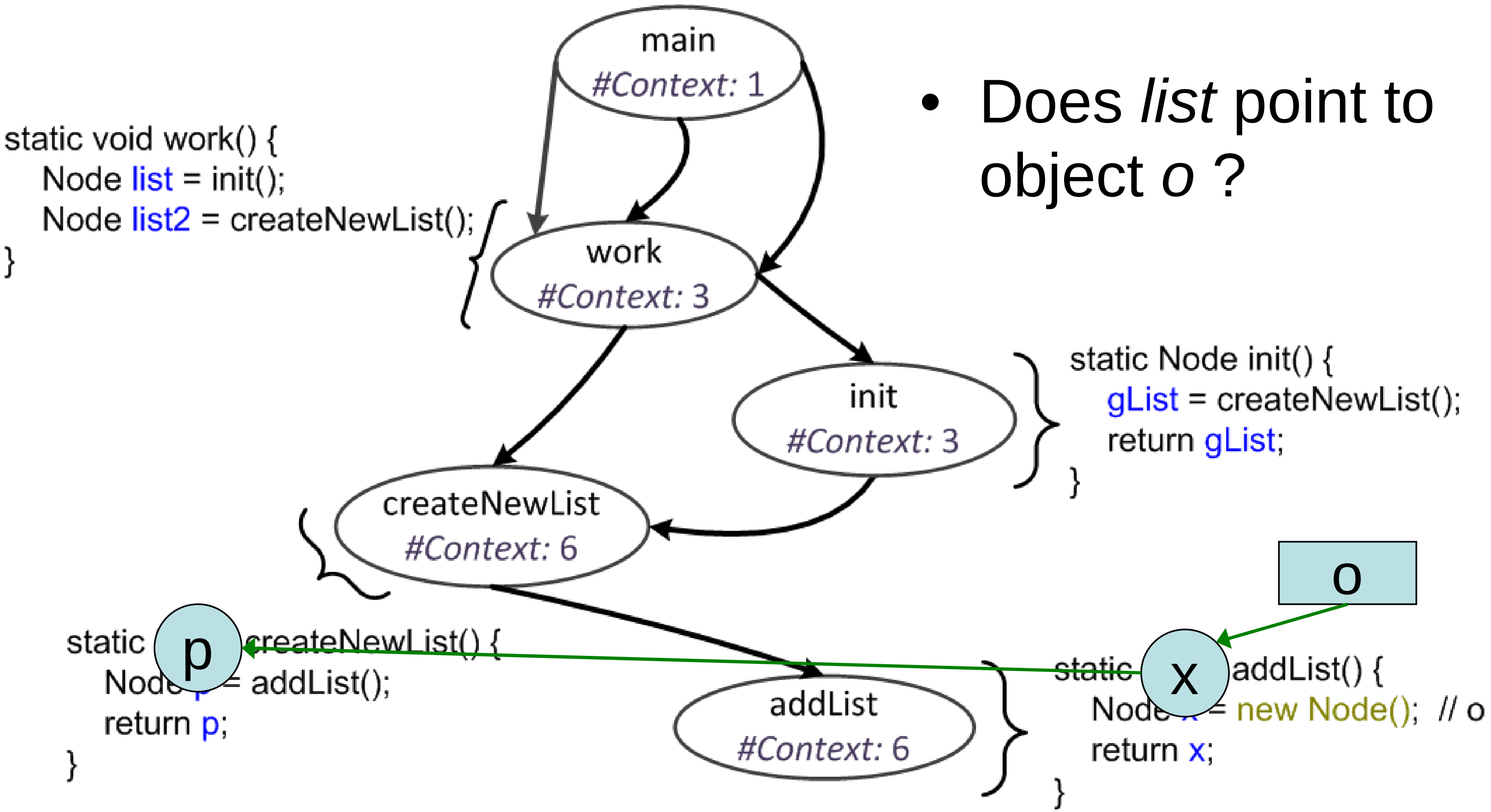
```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```

```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

Points-to Analysis



```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

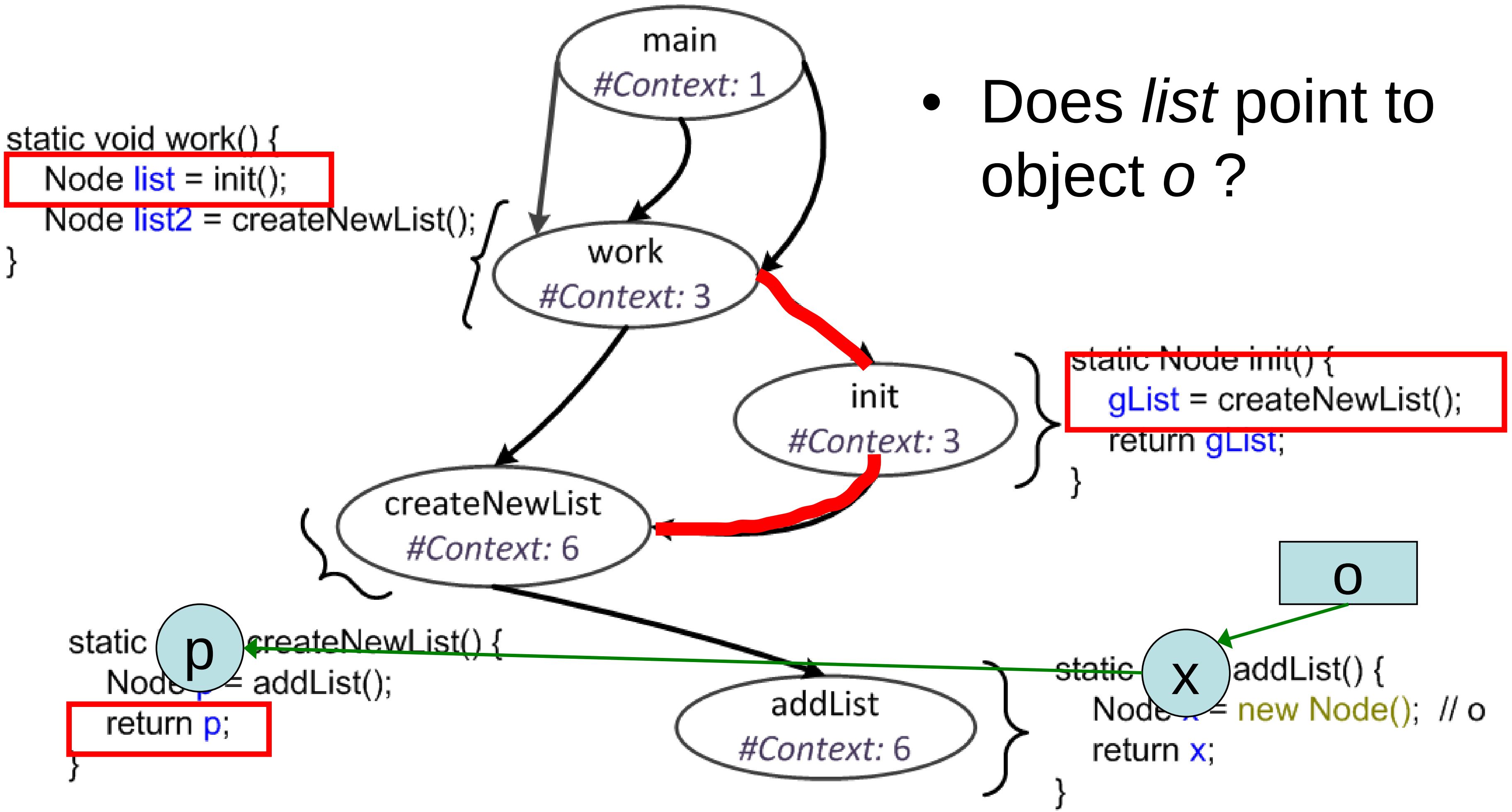
```
static      createNewList() {
    Node p = addList();
    return p;
}
```

```
static      addList() {
    Node x = new Node();  // o
    return x;
}
```

- Does *list* point to object *o* ?

Points-to Analysis

- Does *list* point to object *o* ?



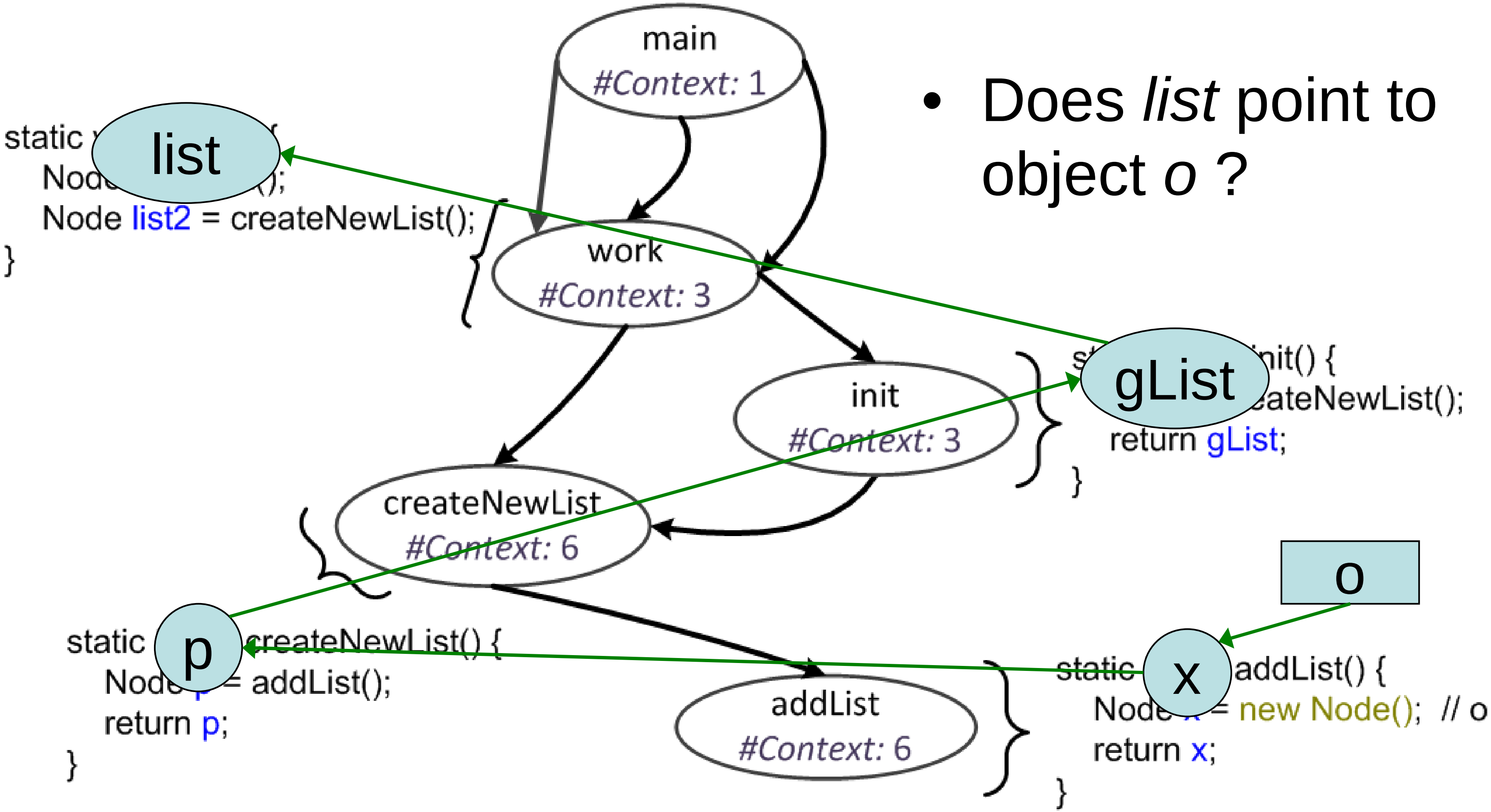
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

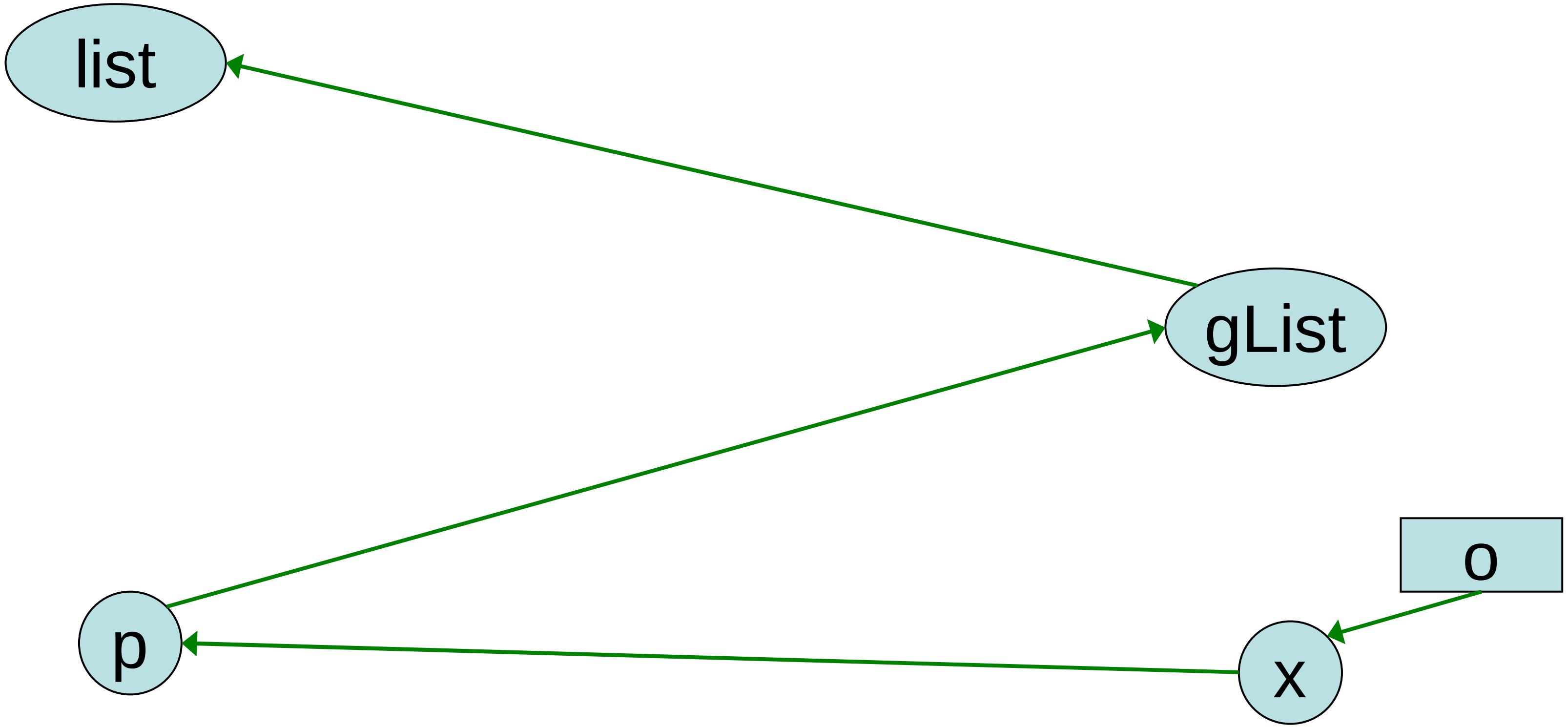
```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Points-to Analysis

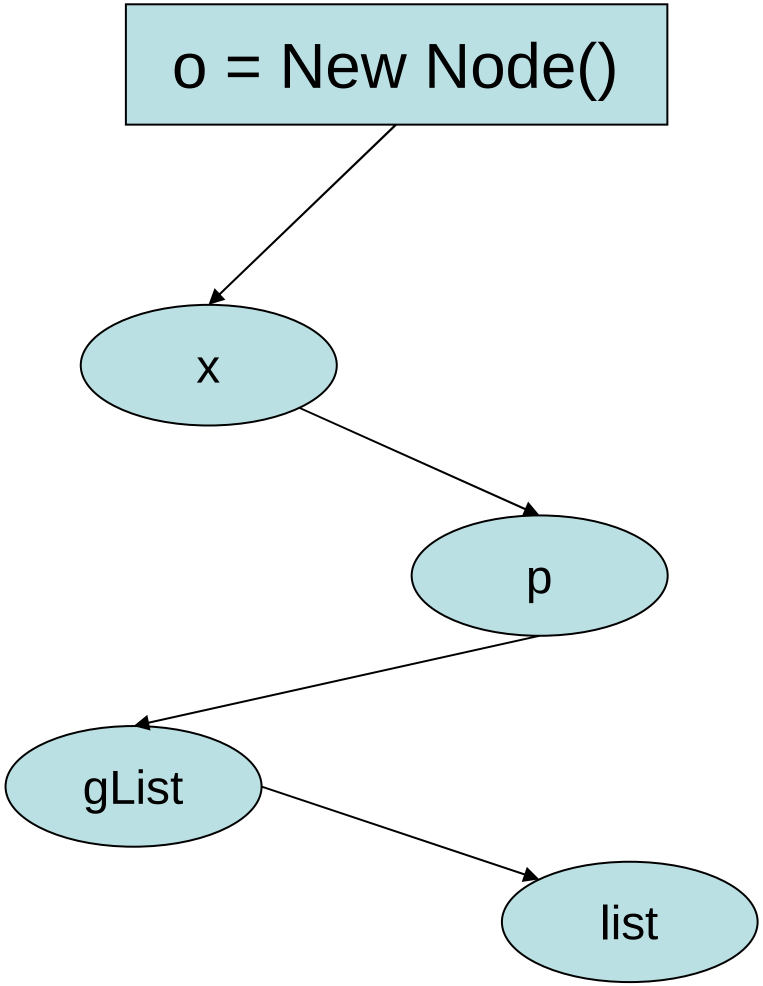


- Does *list* point to object *o* ?

Points-to Analysis

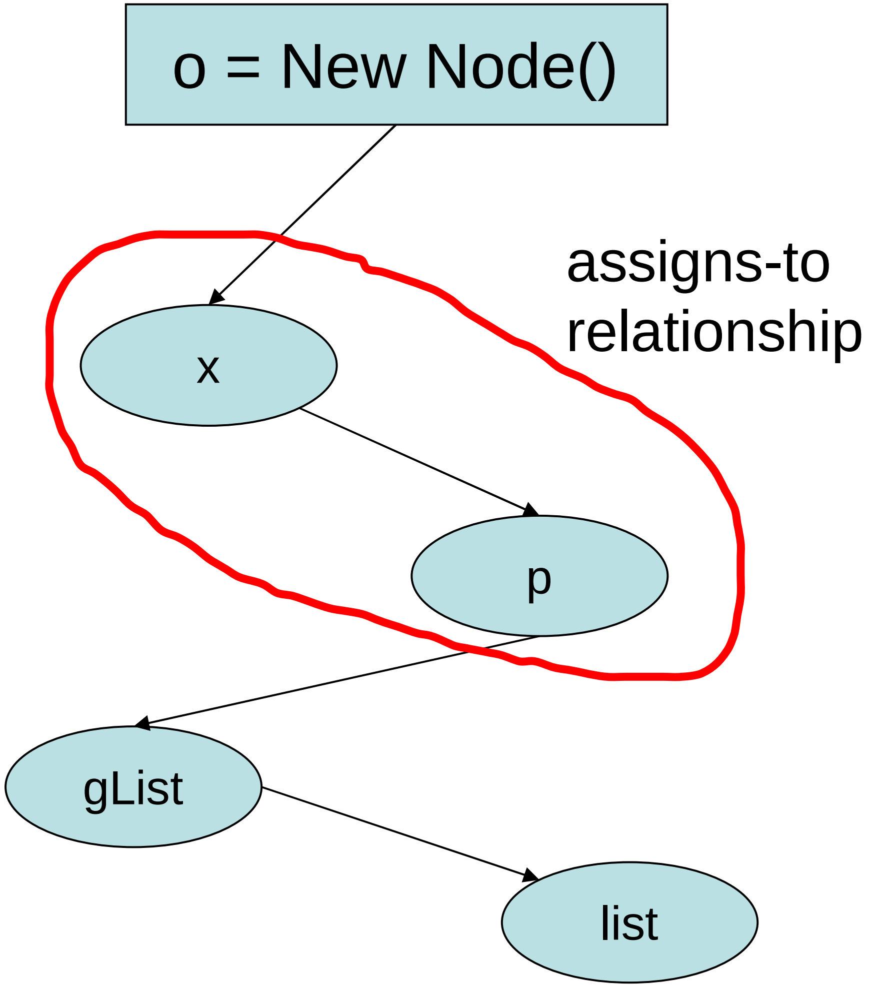


Points-to Analysis



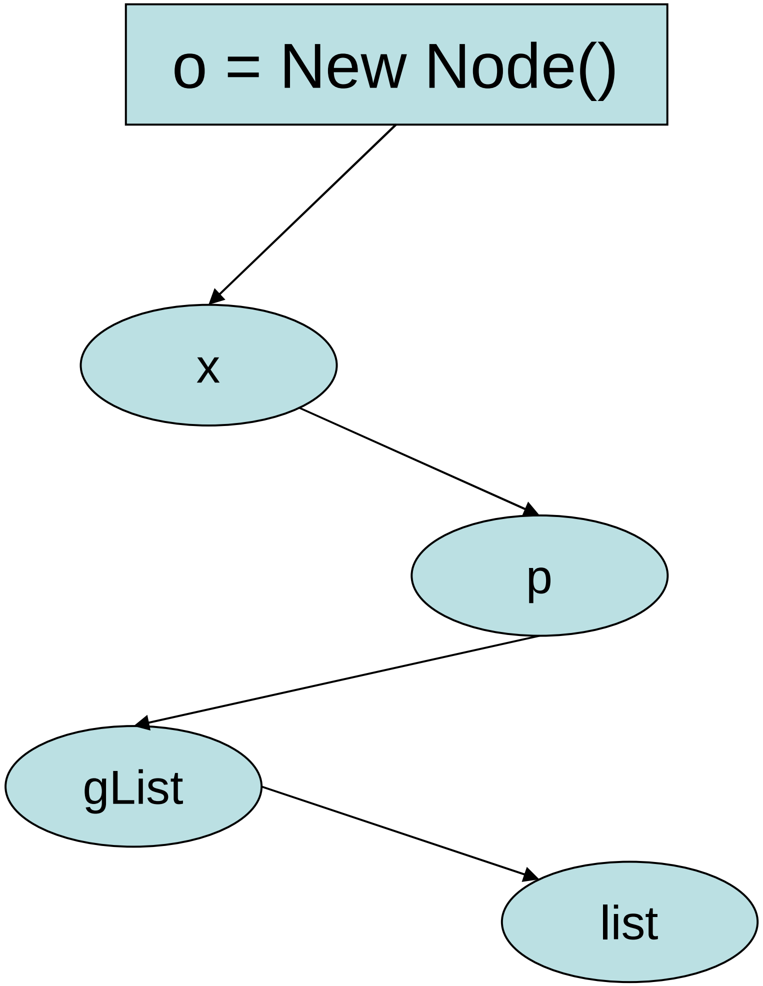
- Flow graph;

Points-to Analysis



- Flow graph;

Points-to Analysis

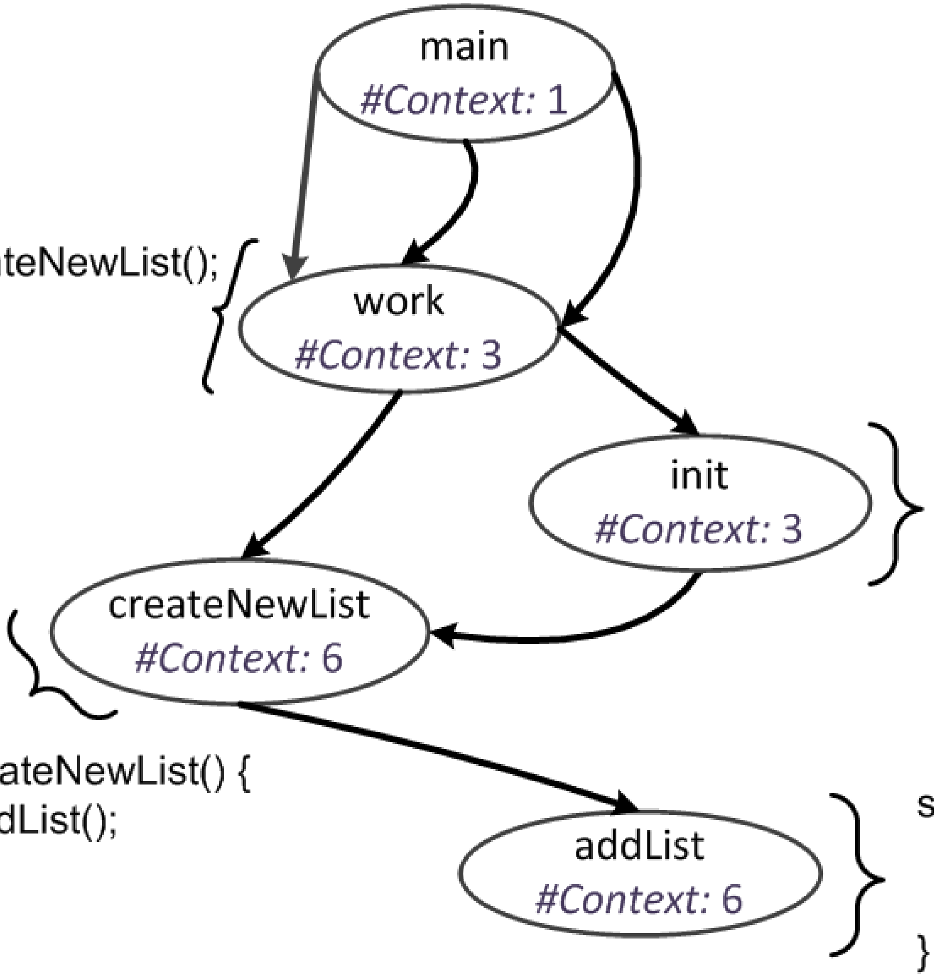


- Flow graph;
- Points-to relations can be obtained via the ***graph reachability analysis***;

Definition

- The previous demo is Anderson's analysis, which is *context insensitive*.
- It does not distinguish the runtime instances for the same syntactic variable.
- Cause any problem?

Context Sensitivity



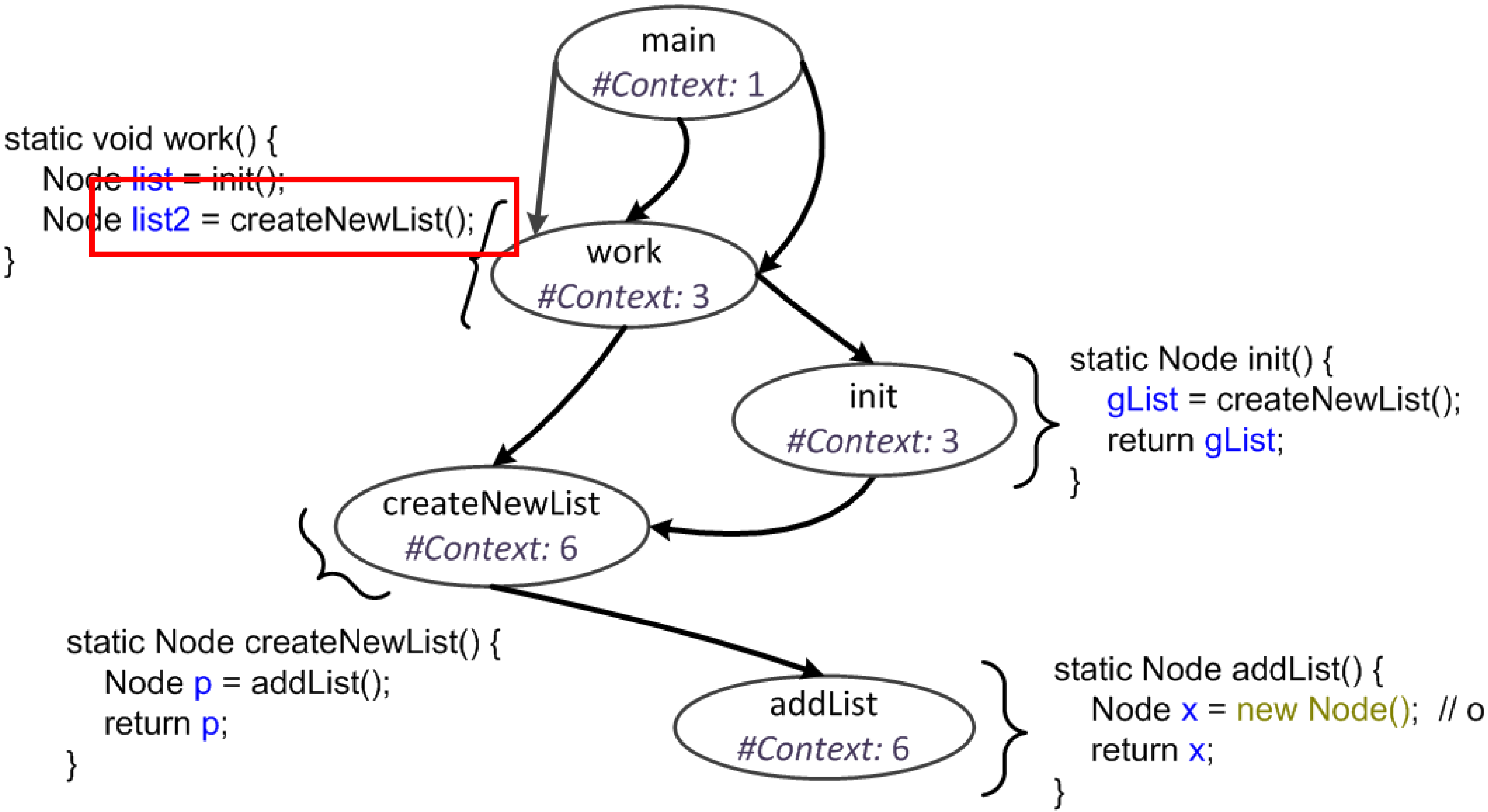
```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```

```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

Context Sensitivity



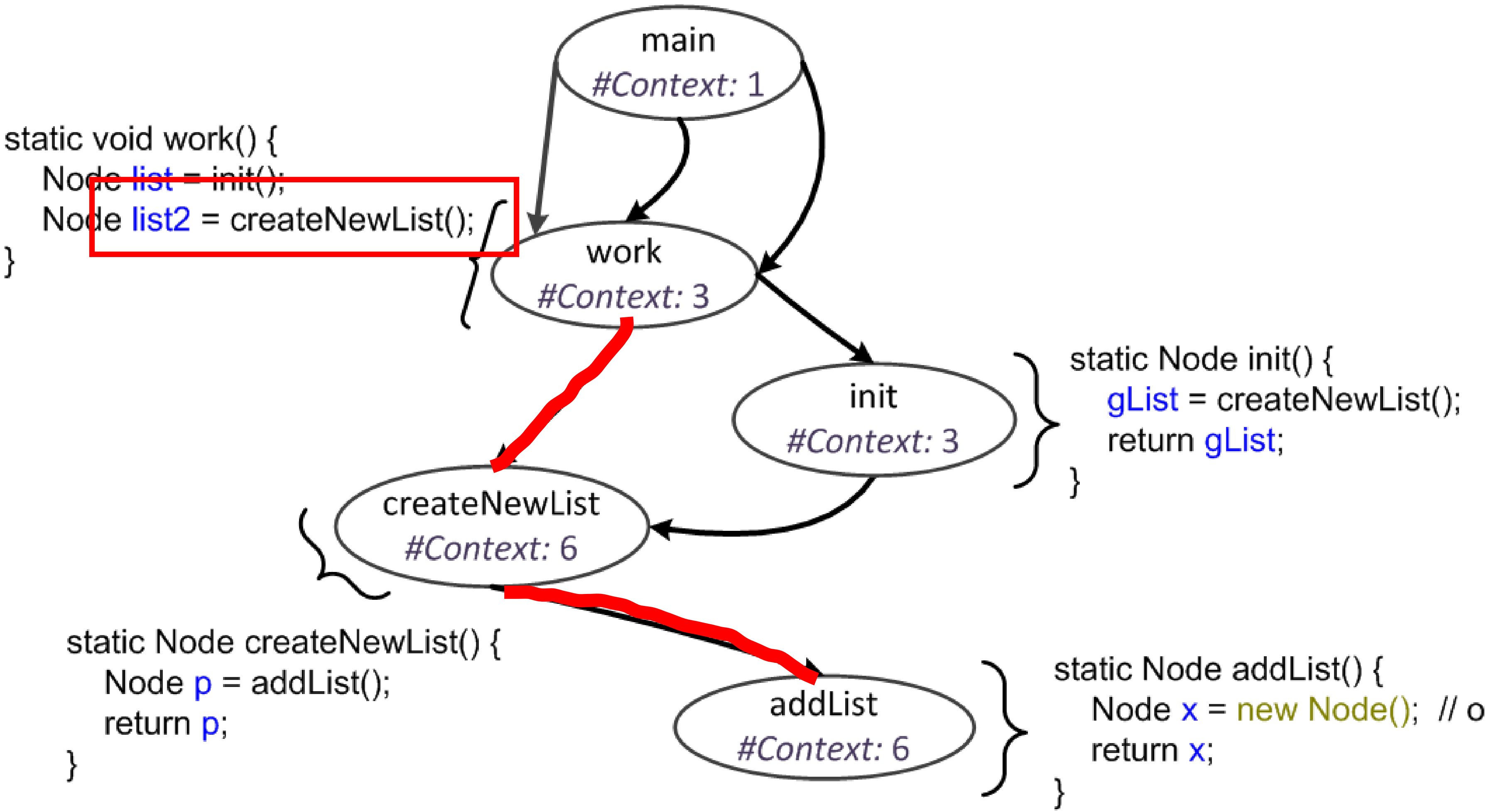
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



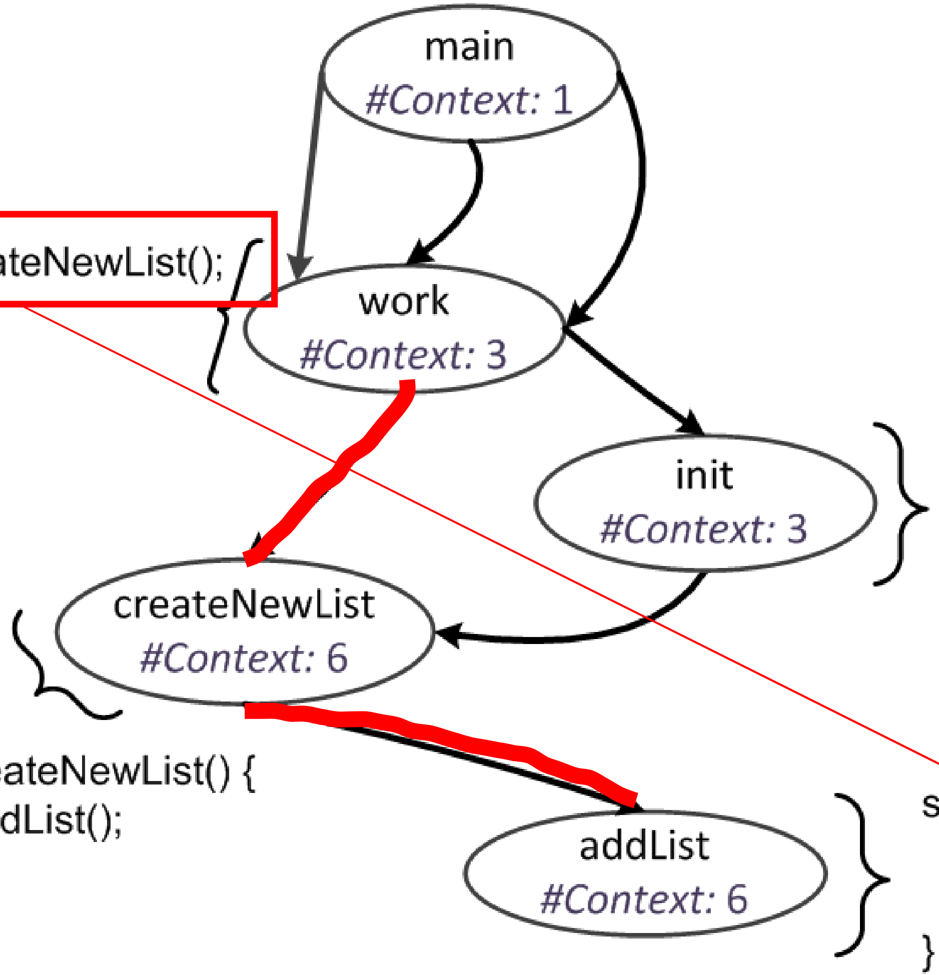
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



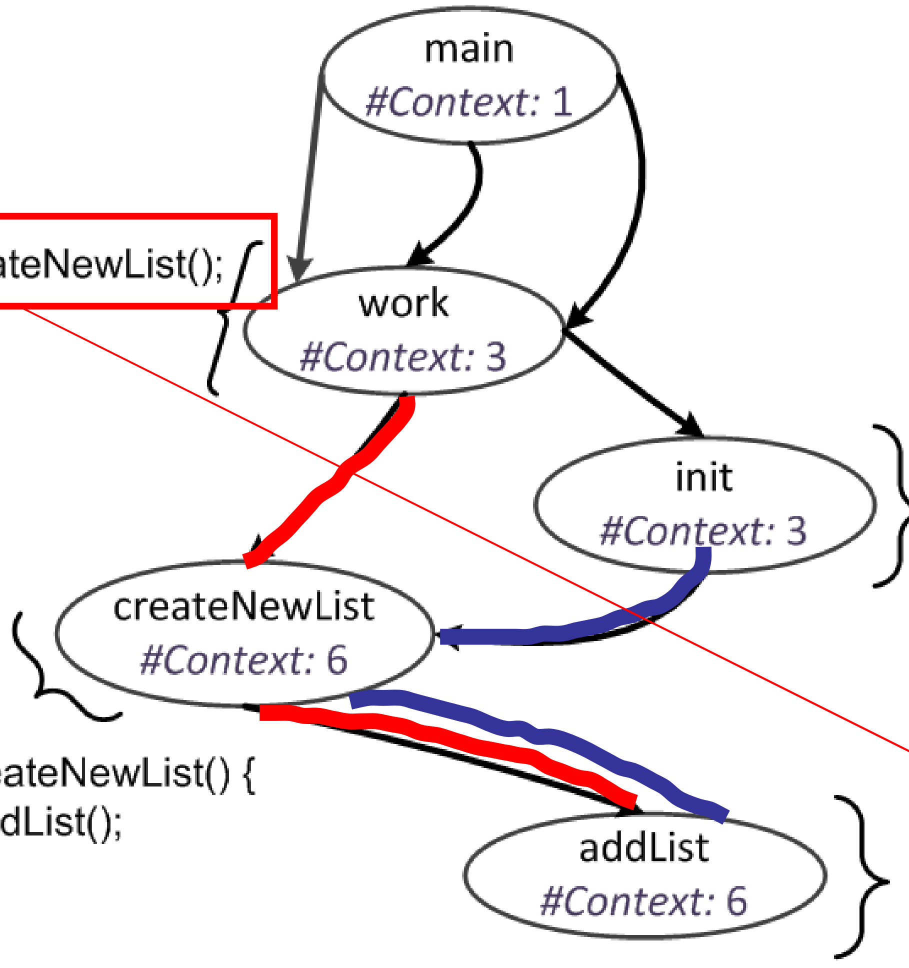
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



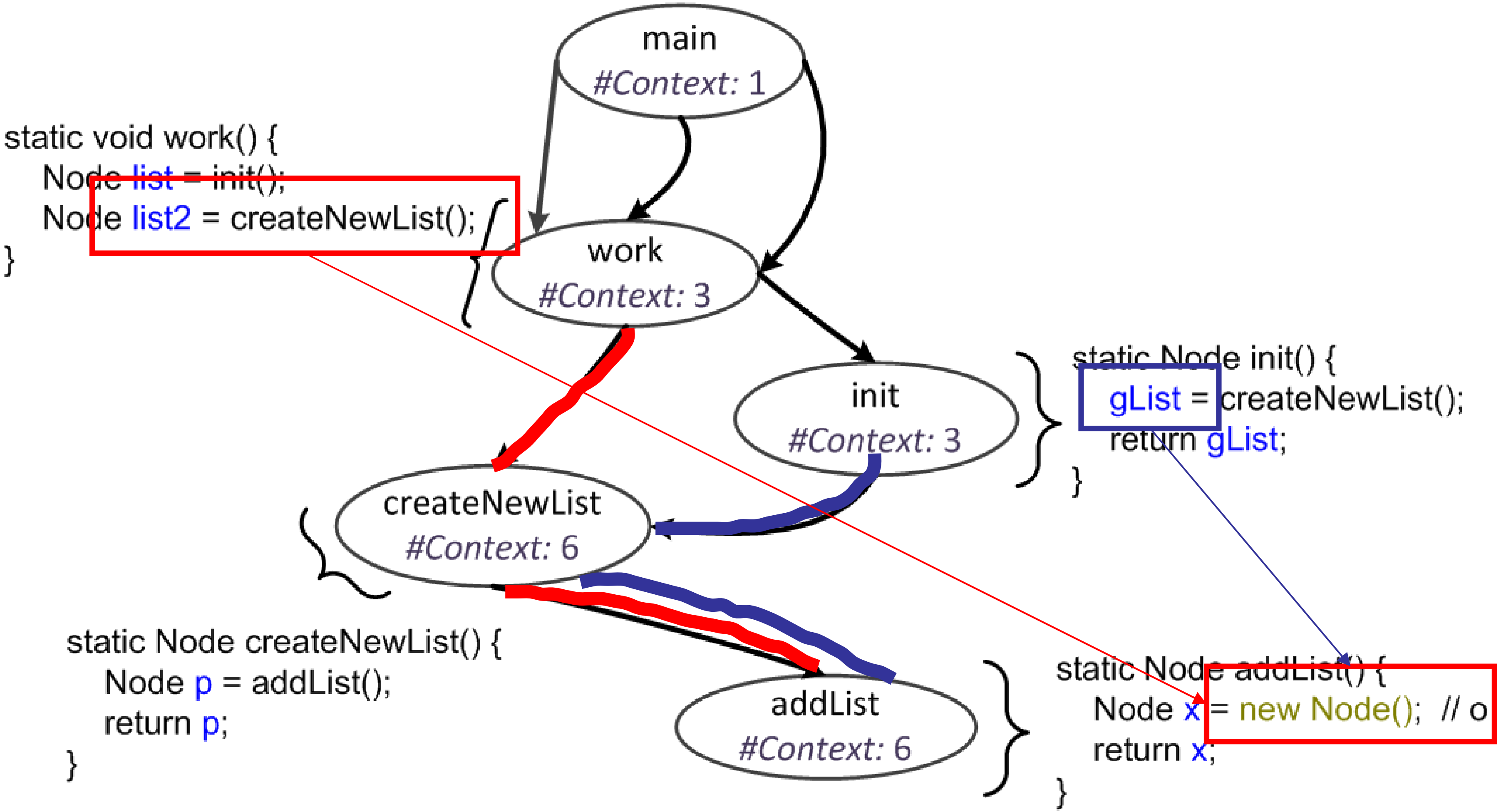
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

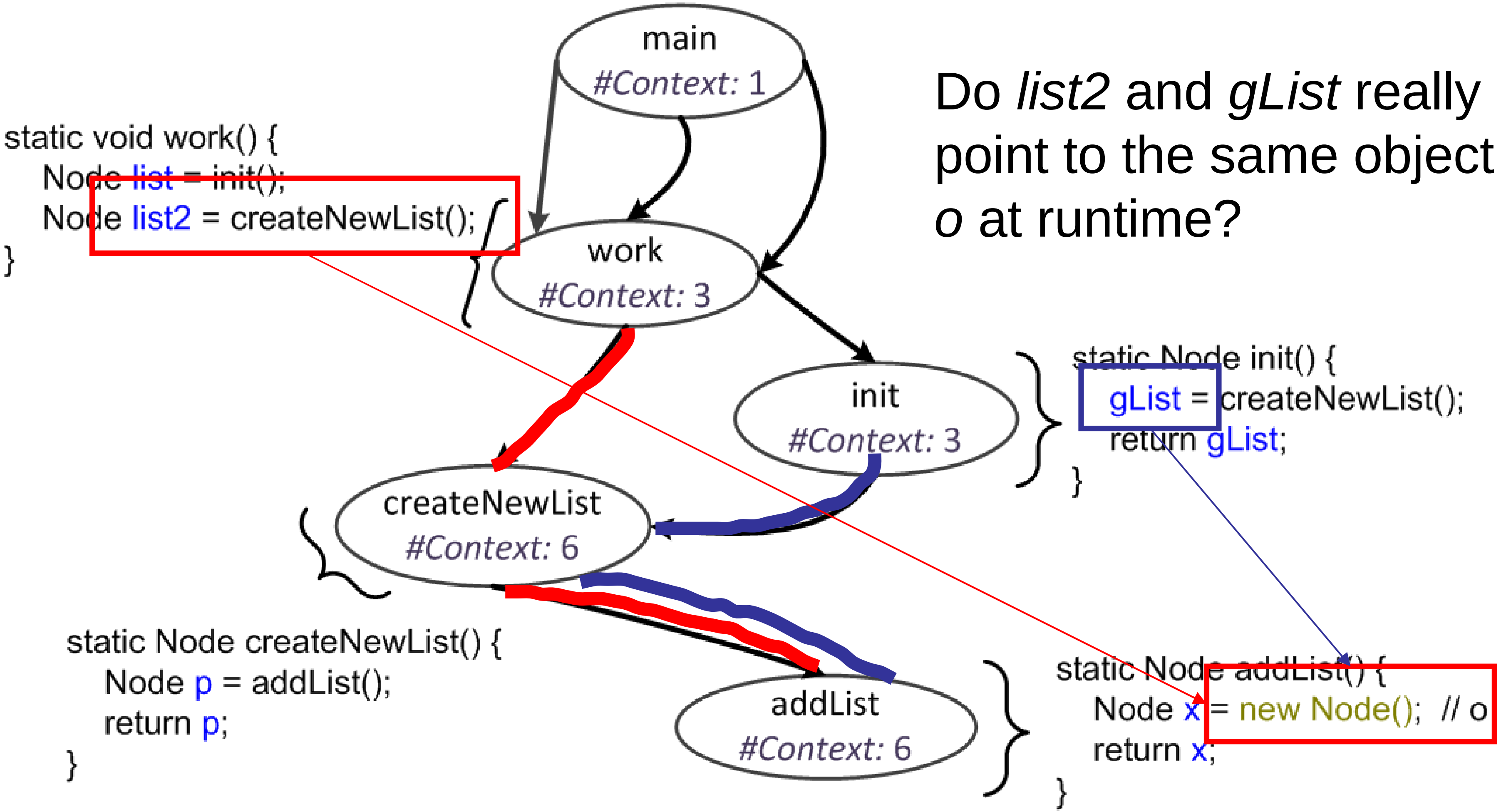
```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity

Do *list2* and *gList* really point to the same object *o* at runtime?



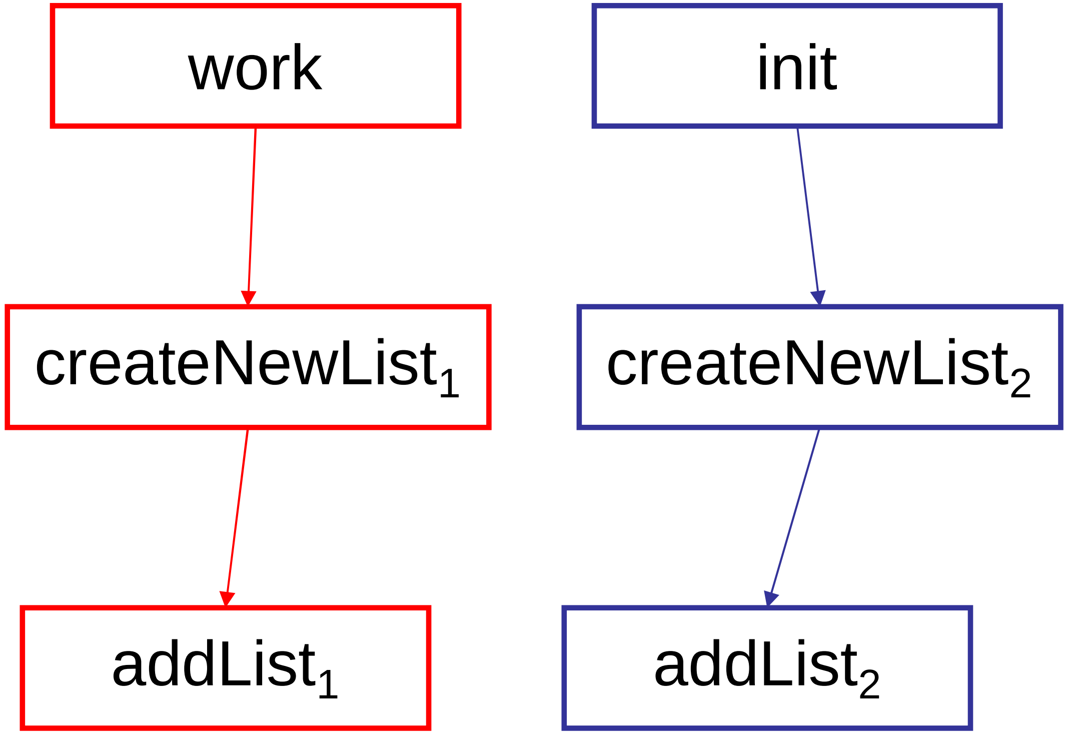
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

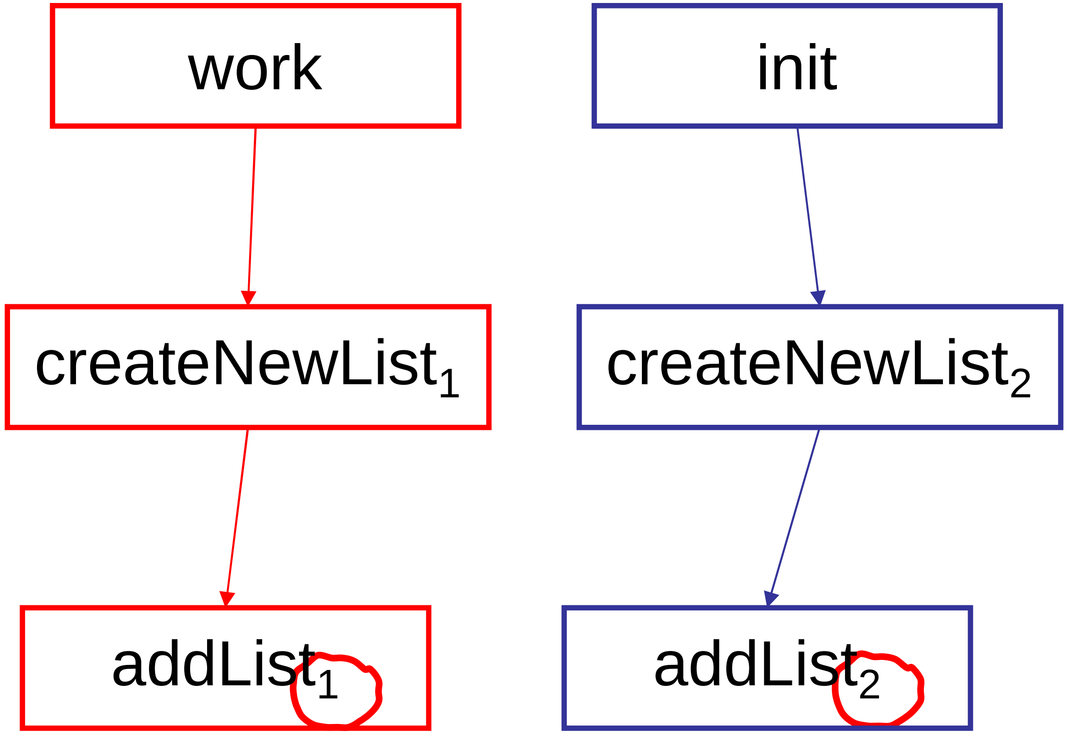
```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



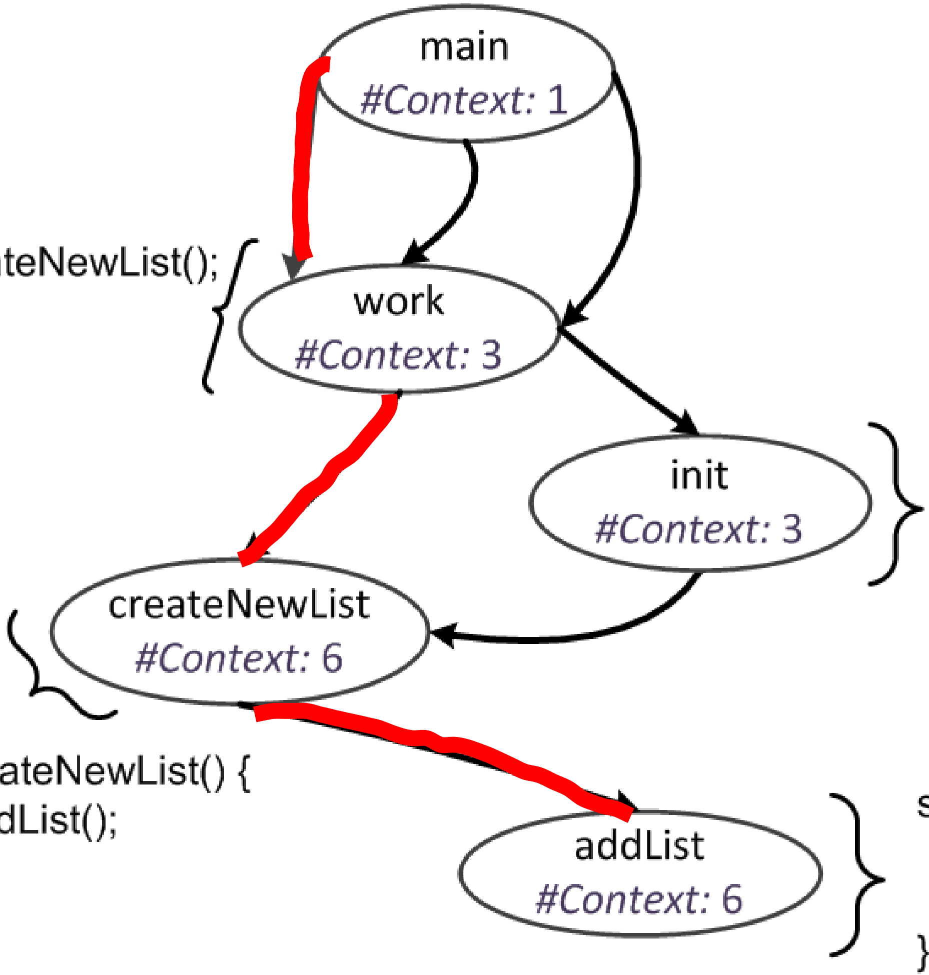
Context Sensitivity



Context Sensitivity

- 6 paths to **addList()** from **main()**;

Context Sensitivity



```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

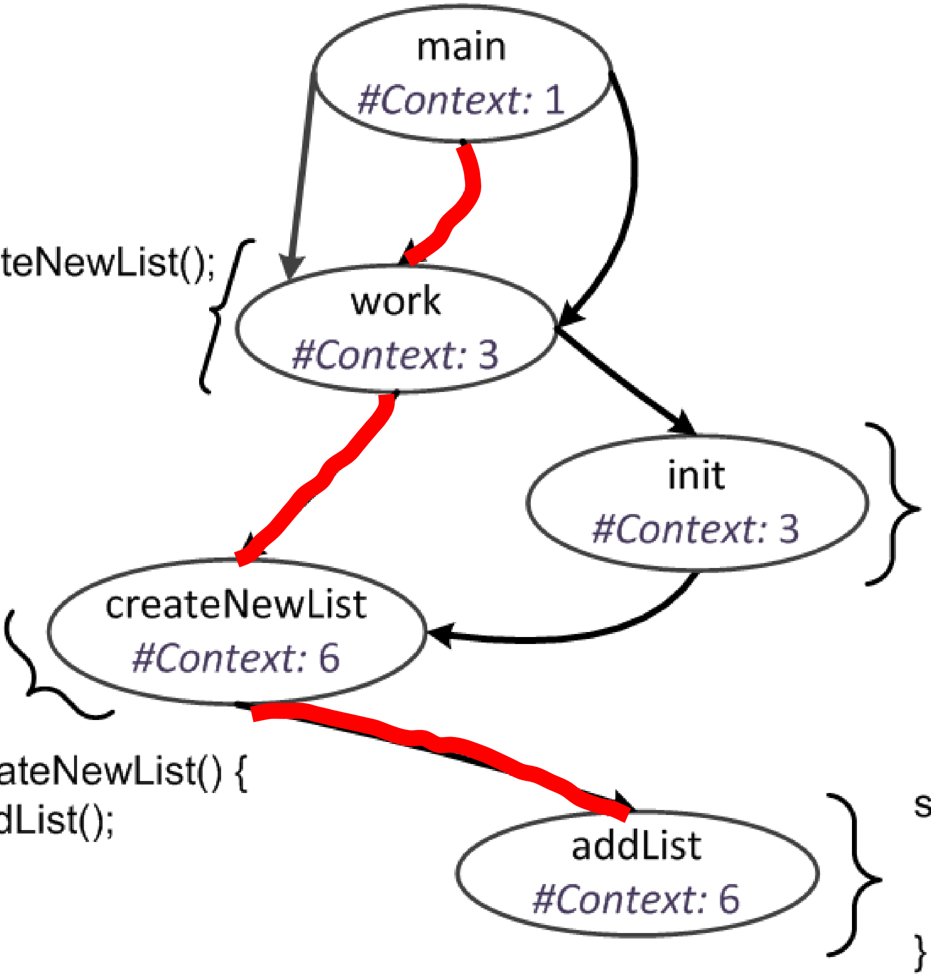
```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity

```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```



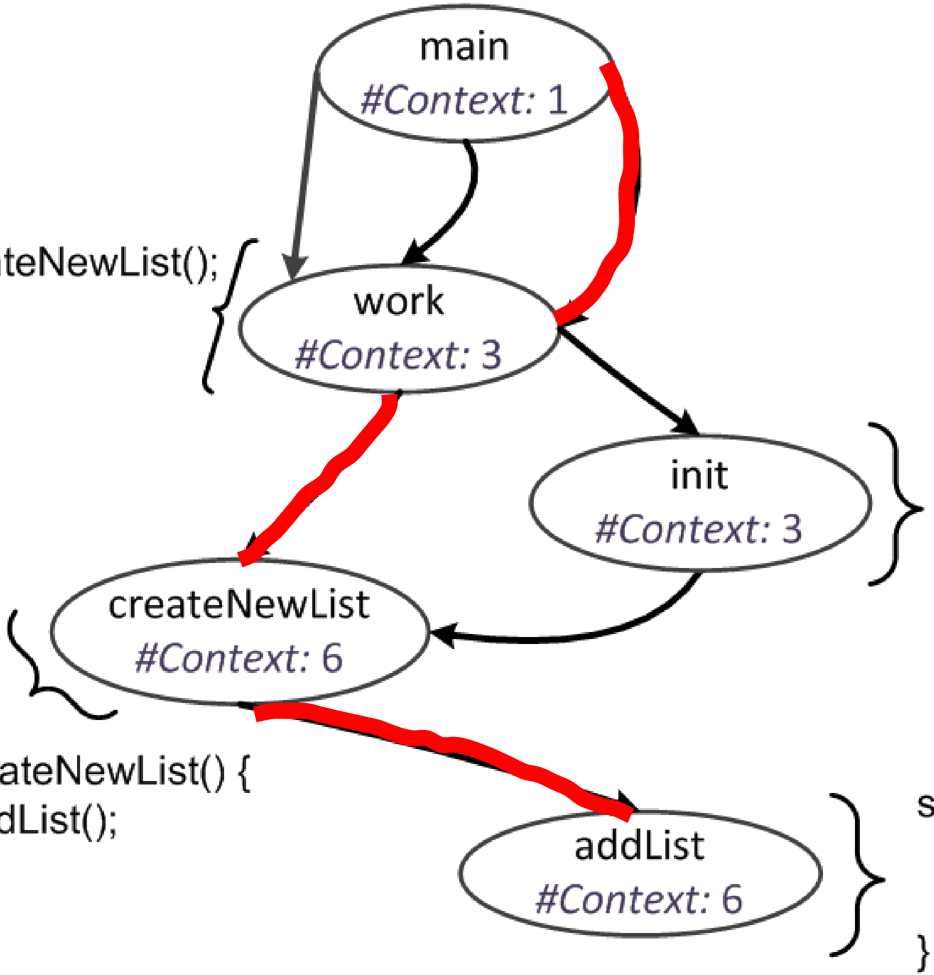
```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```

Context Sensitivity

```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```



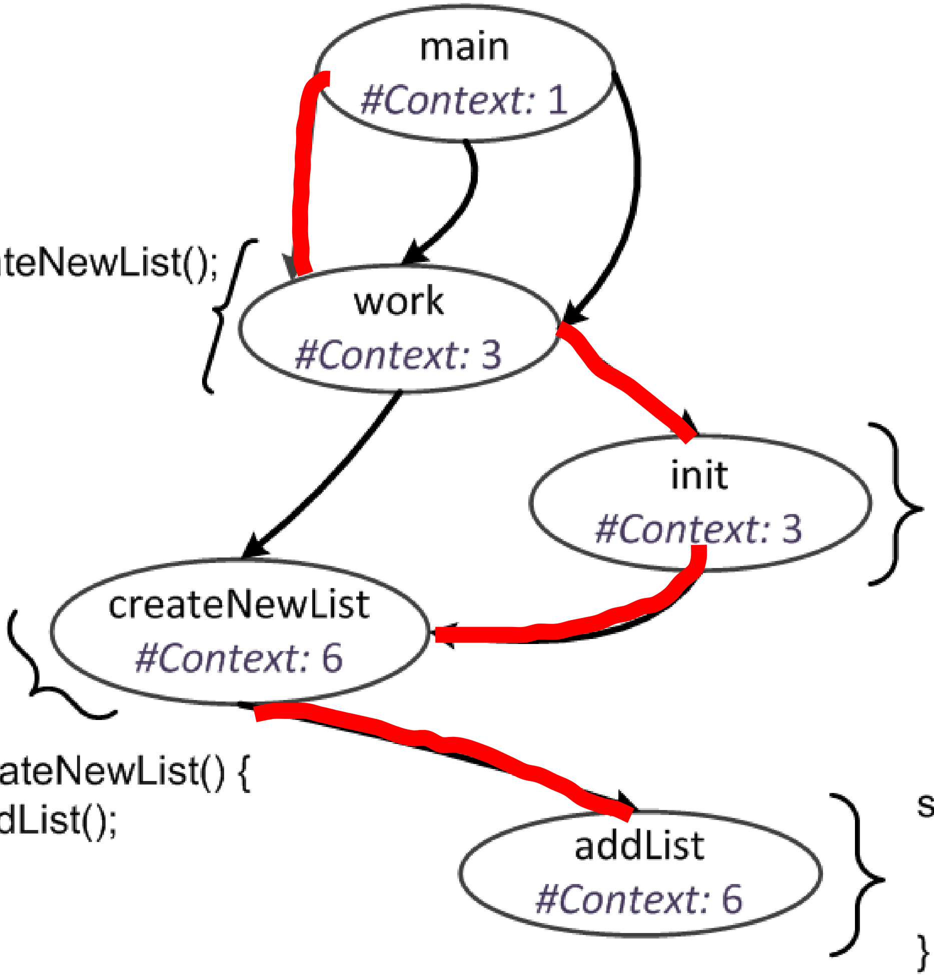
```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity

```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```



main

#Context: 1

work

#Context: 3

init

#Context: 3

createNewList

#Context: 6

addList

#Context: 6

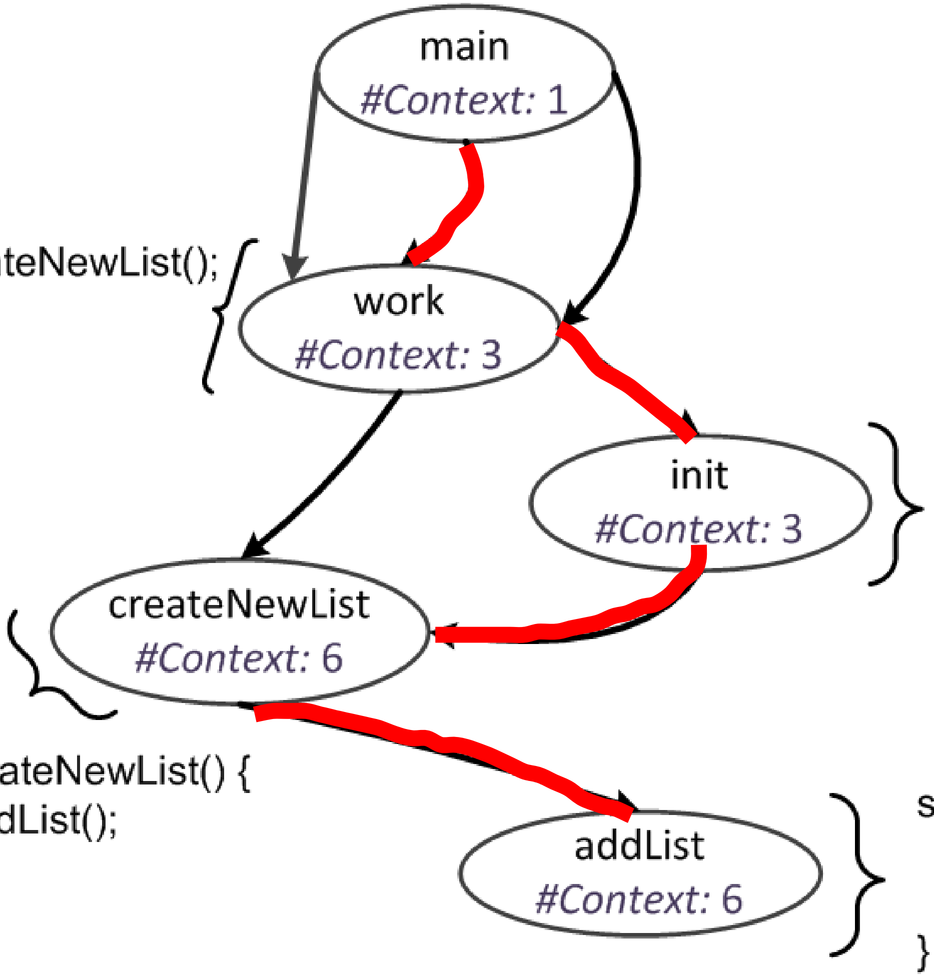
```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity

```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

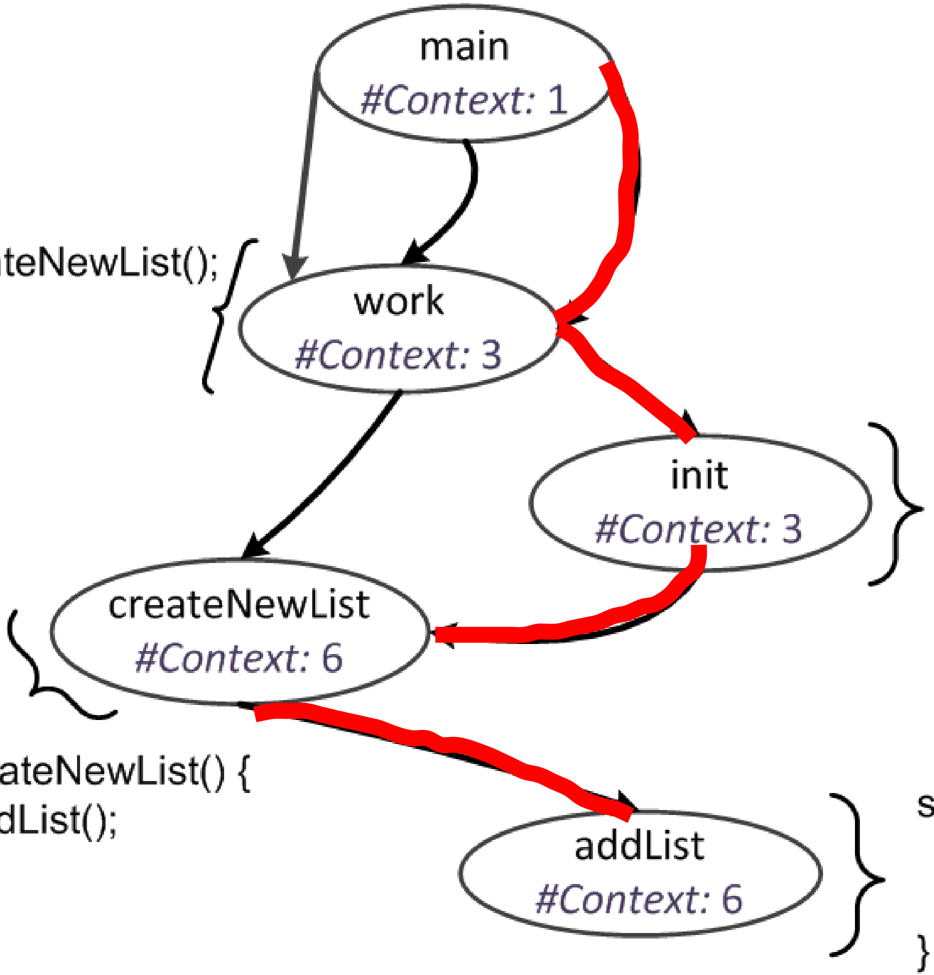


```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitivity



```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

New Algorithm

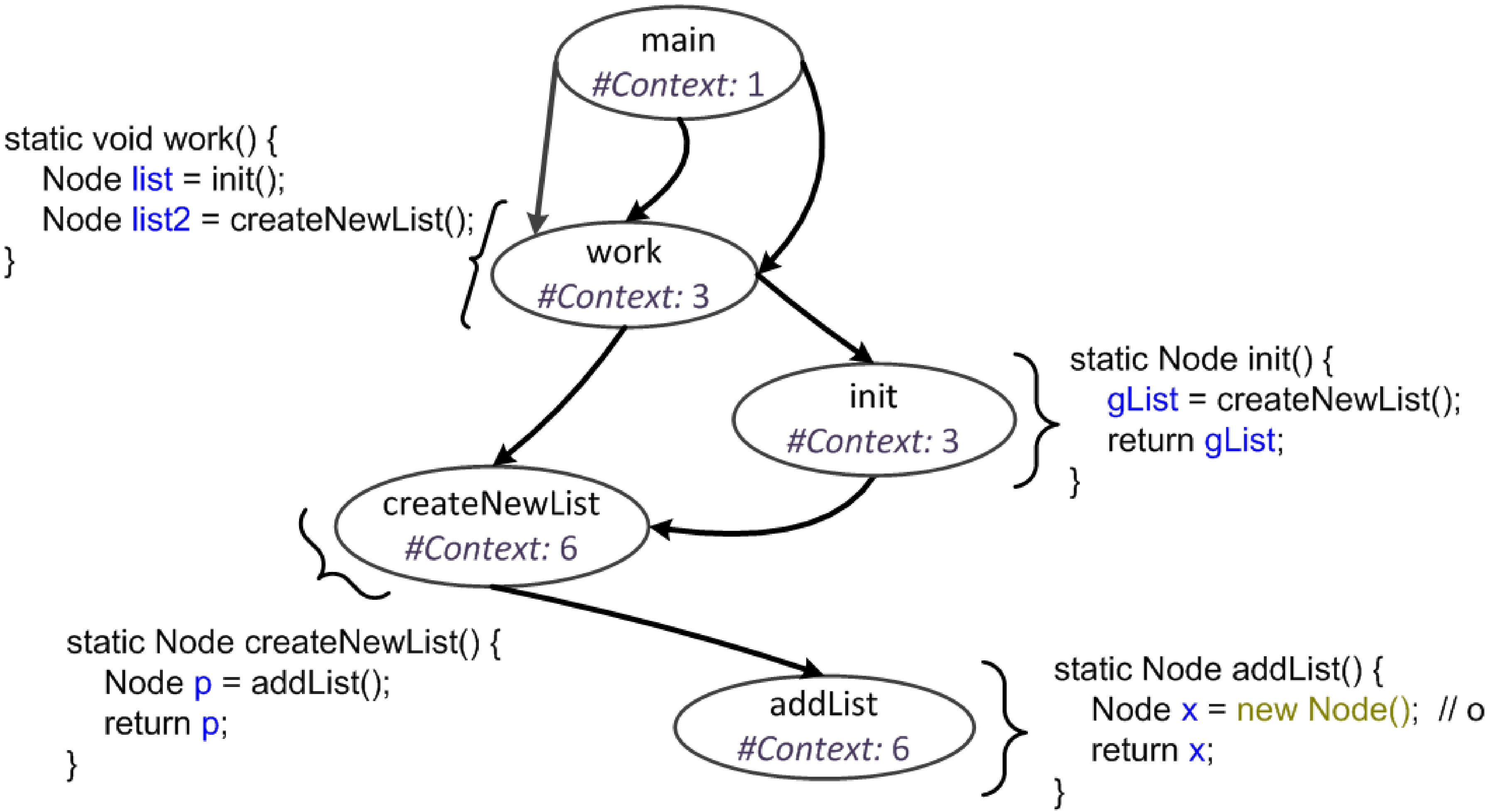
- How do we design an algorithm that can take advantage of the *context* ?

Context Sensitive Points-to Analysis

Step 1

- Build a context sensitive call graph by duplicating every function *N* times if it has *N* contexts.

Context Sensitive Call Graph



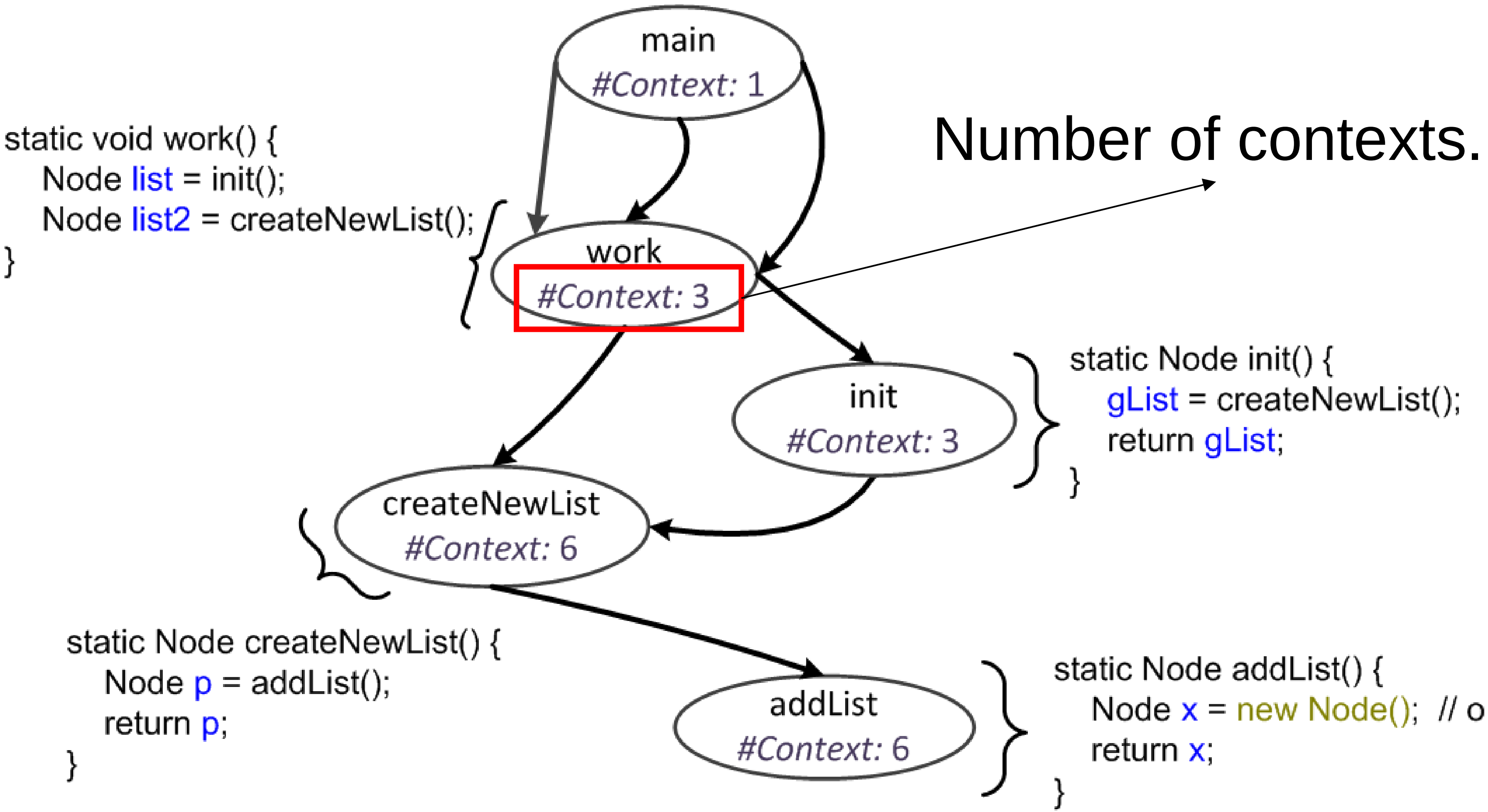
```
static void work() {
    Node list = init();
    Node list2 = createNewList();
}
```

```
static Node init() {
    gList = createNewList();
    return gList;
}
```

```
static Node createNewList() {
    Node p = addList();
    return p;
}
```

```
static Node addList() {
    Node x = new Node();  // o
    return x;
}
```

Context Sensitive Call Graph



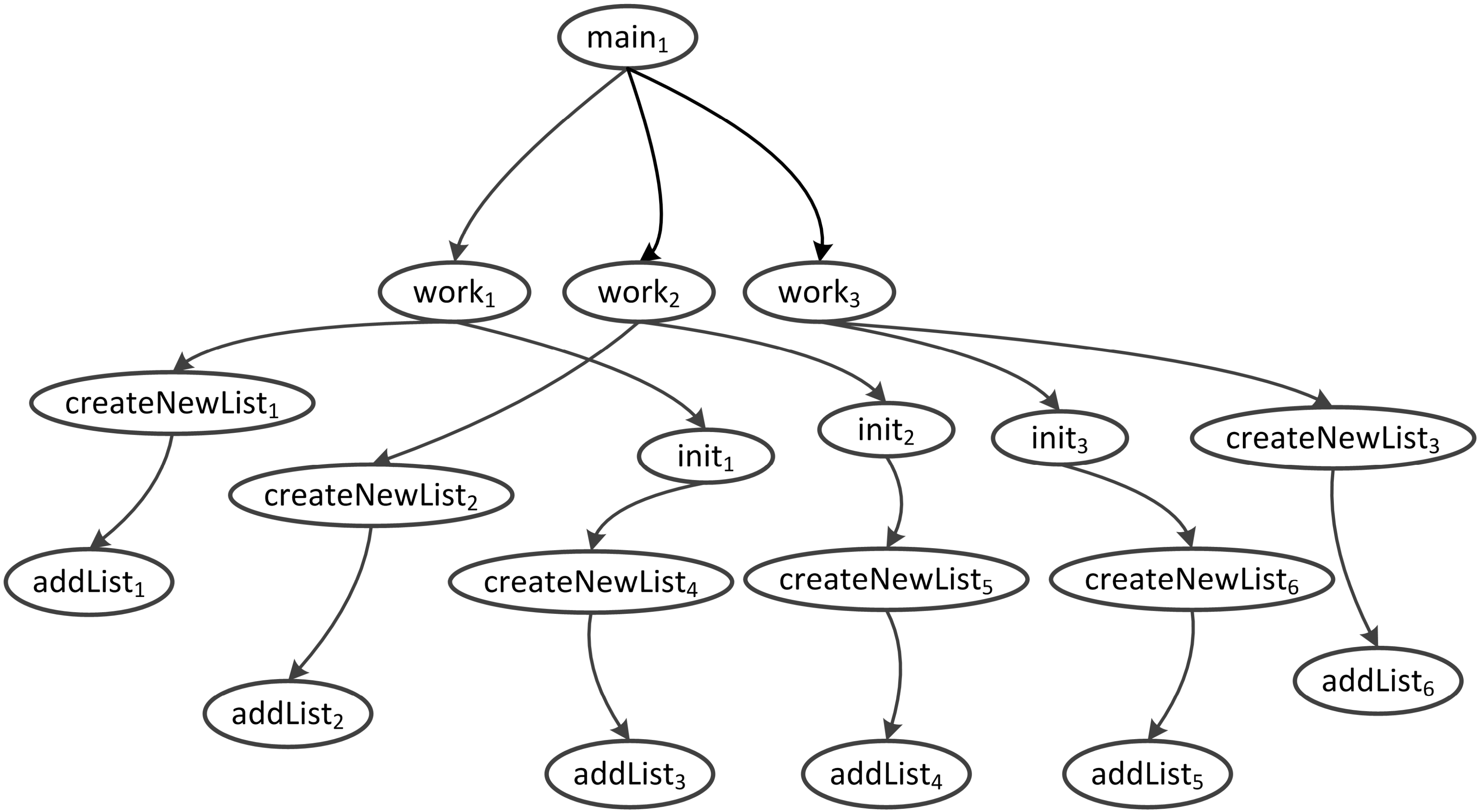
Number of contexts.

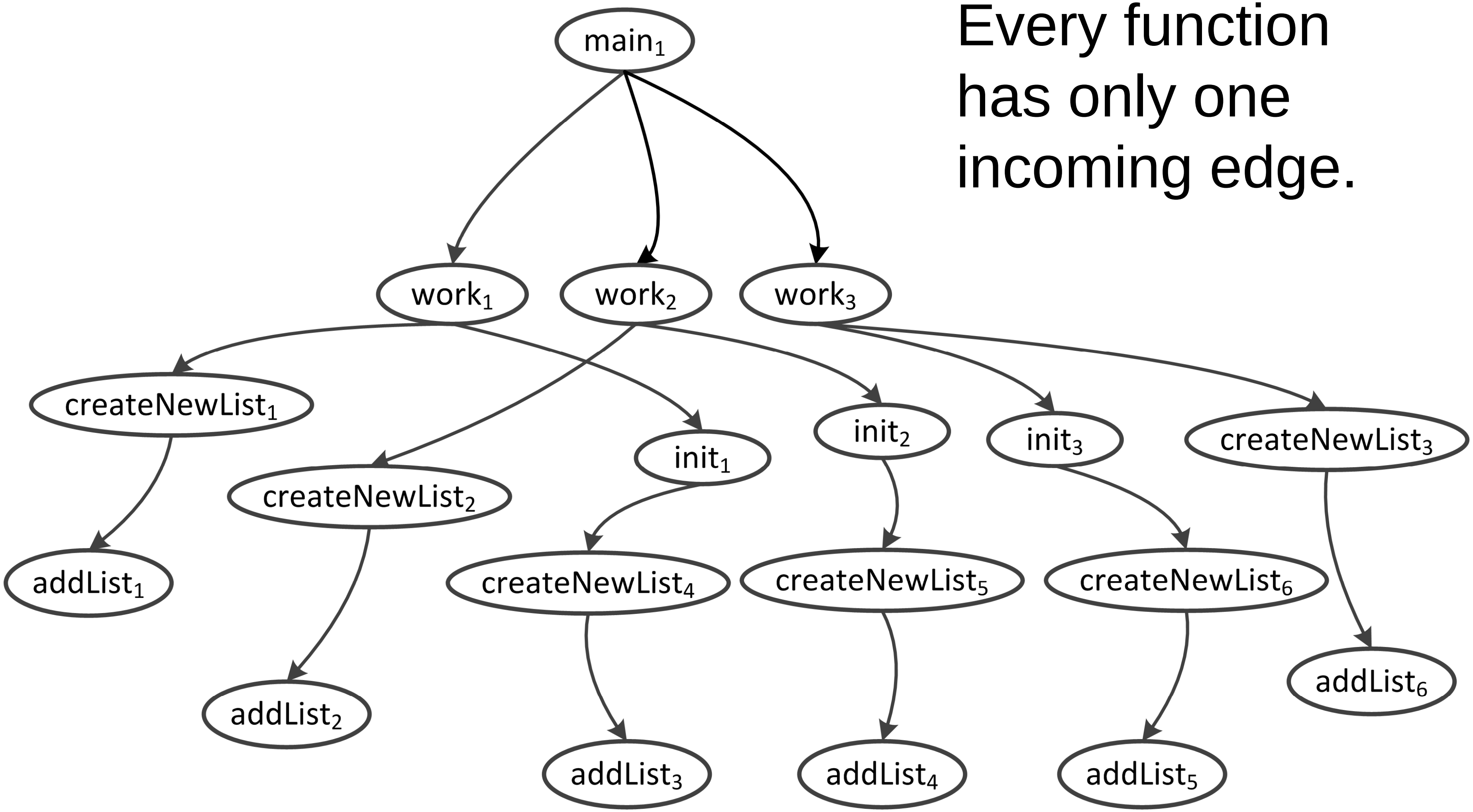
```
static void work() {
   Node list = init();
   Node list2 = createNewList();
}
```

```
static Node init() {
   gList = createNewList();
   return gList;
}
```

```
static Node createNewList() {
   Node p = addList();
   return p;
}
```

```
static Node addList() {
   Node x = new Node();  // o
   return x;
}
```


main$_1$
work$_1$
work$_2$
work$_3$
createNewList$_1$
createNewList$_2$
init$_1$
init$_2$
init$_3$
createNewList$_3$
addList$_1$
createNewList$_4$
createNewList$_5$
createNewList$_6$
addList$_6$
addList$_2$
addList$_3$
addList$_4$
addList$_5$



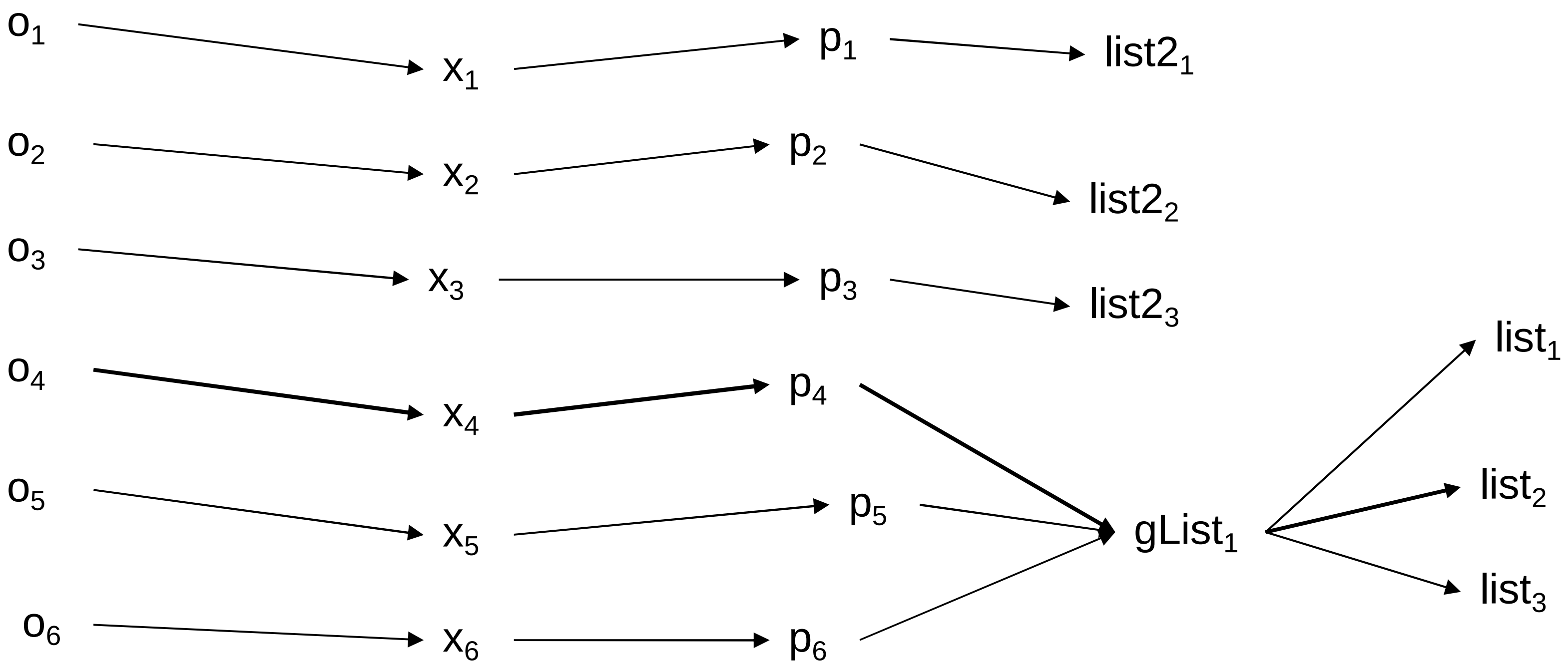
Every function has only one incoming edge.

Context Sensitive Points-to Analysis

Step 2

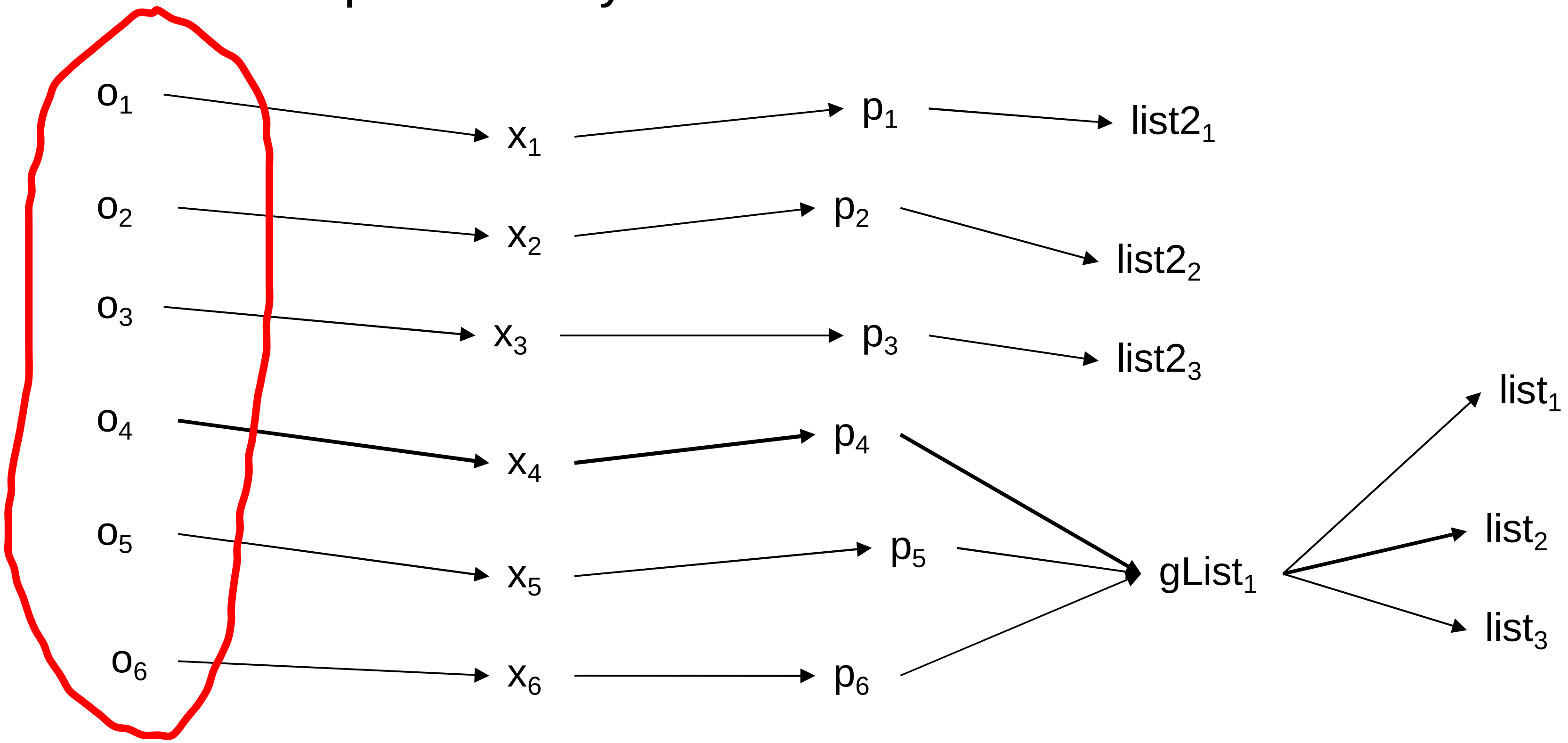
- Apply the Anderson's analysis to the context sensitive call graph.

Context Sensitive Flow Graph

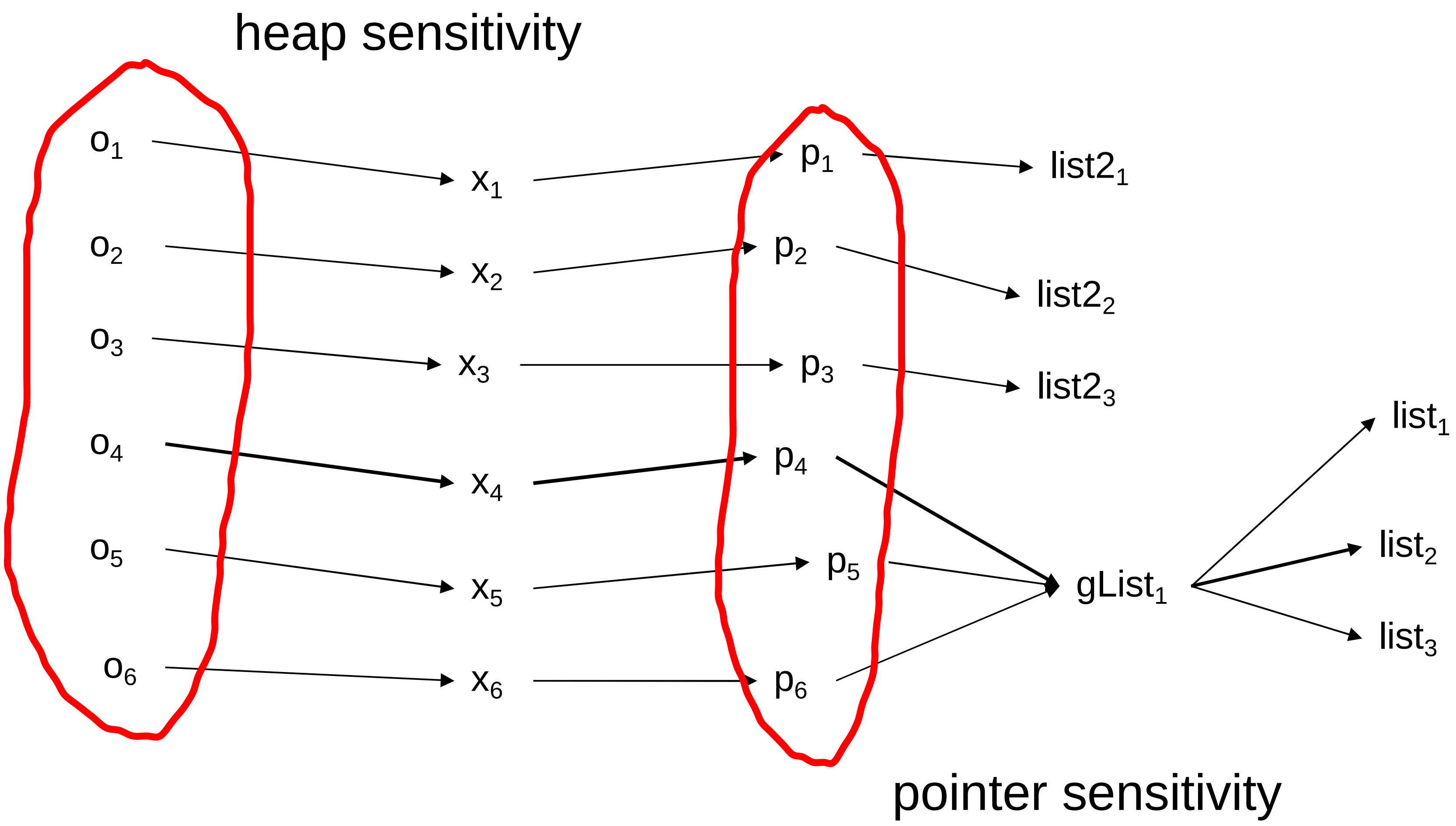


Context Sensitive Flow Graph

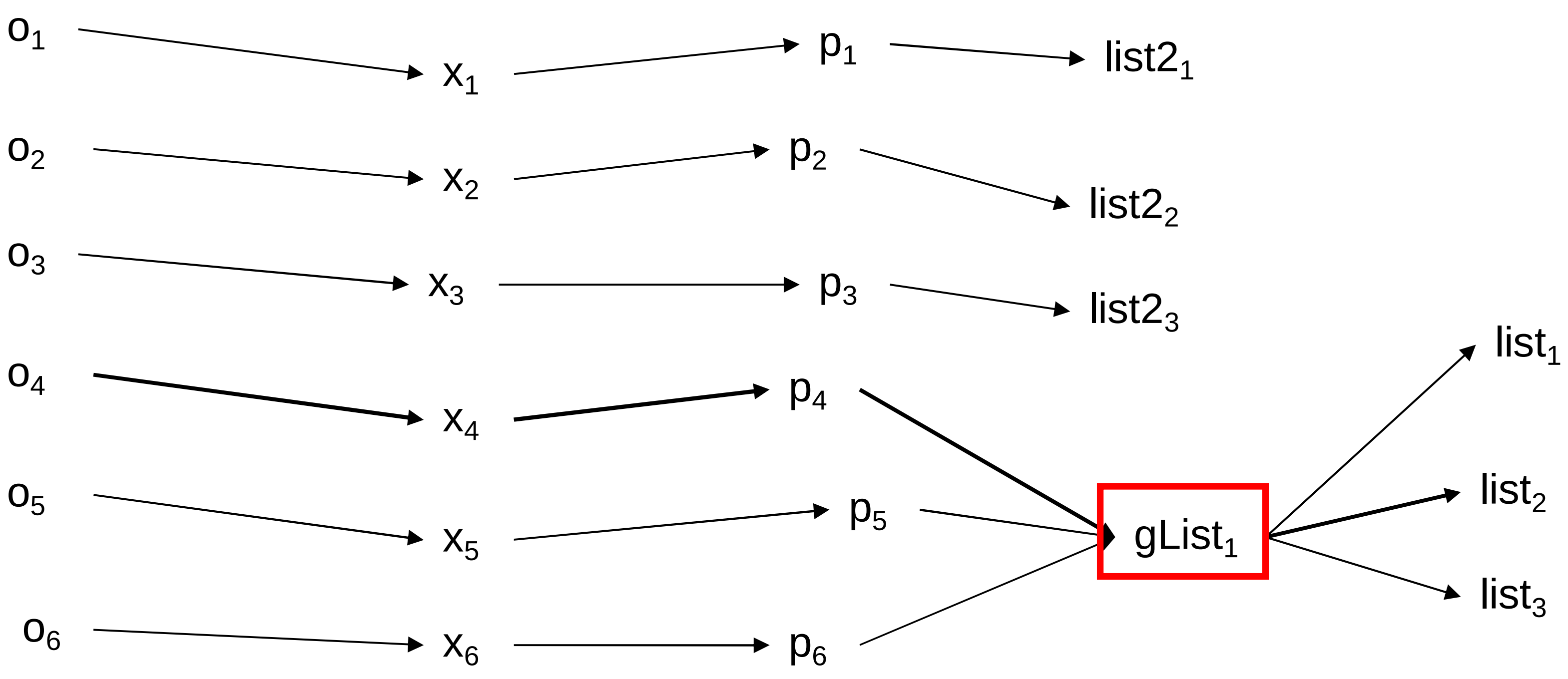
heap sensitivity



Context Sensitive Flow Graph

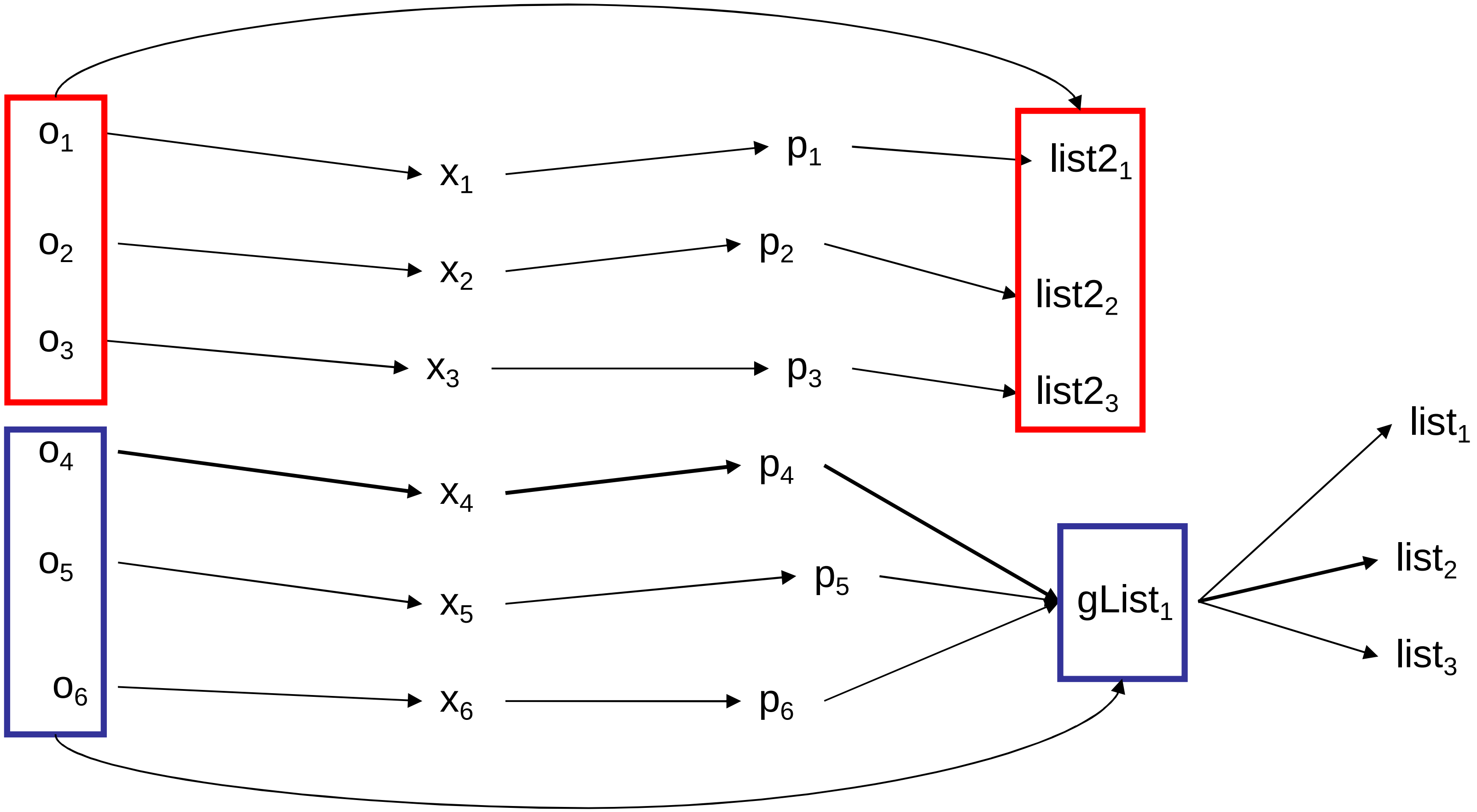


Context Sensitive Flow Graph



香港科技大學
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Context Sensitive Flow Graph



Context Sensitive Points-to Analysis

- How can we do reachability analysis with the context sensitive flow graph efficiently?

Challenge

- The flow graph is extremely large!
- Handling a graph with billions of nodes is not an easy job.

bddbddb [PLDI04]

- Store the initial assigns-to relations into the BDD;
- Store the initial points-to relations into another BDD;
- Iteratively apply the BDD join operator until the points-to relations fixed;

bddbddb [PLDI04]

- Advantages:
  - A very large flow-to graph can be handled within 1GB memory;
  - Programming with BDD is not so difficult.

bddbddb [PLDI04]

- Problems:
  - – Multi-dimensional sensitivity (both pointer and heap) support are inefficient;
  - – Although consider the pointer sensitivity only, the time efficiency is still not satisfiable;

EPA [ISSTA08]

- High level idea:
  - Do not number the contexts to 1, 2, 3... but use callsite string to represent contexts;
  - Extract the shared prefix of the context string to achieve compression.

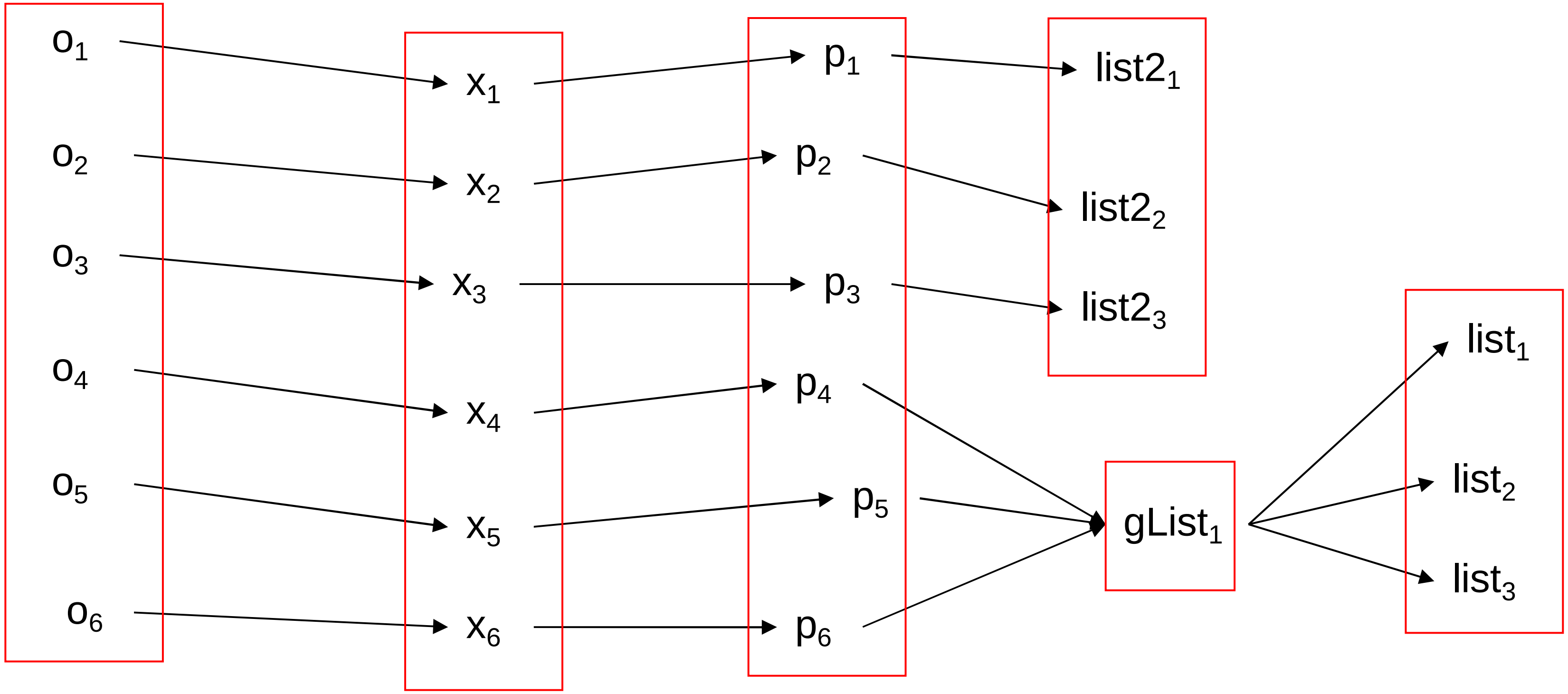
EPA [ISSTA08]

- Problems:
  - Procedure summary based design: less efficient compared to whole program analysis because of more actions (e.g. method escape analysis, symbolic placeholder instantiation, etc.).

Context Sensitive Flow Graph

- Let's see the flow graph again.....

Context Sensitive Flow Graph



香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

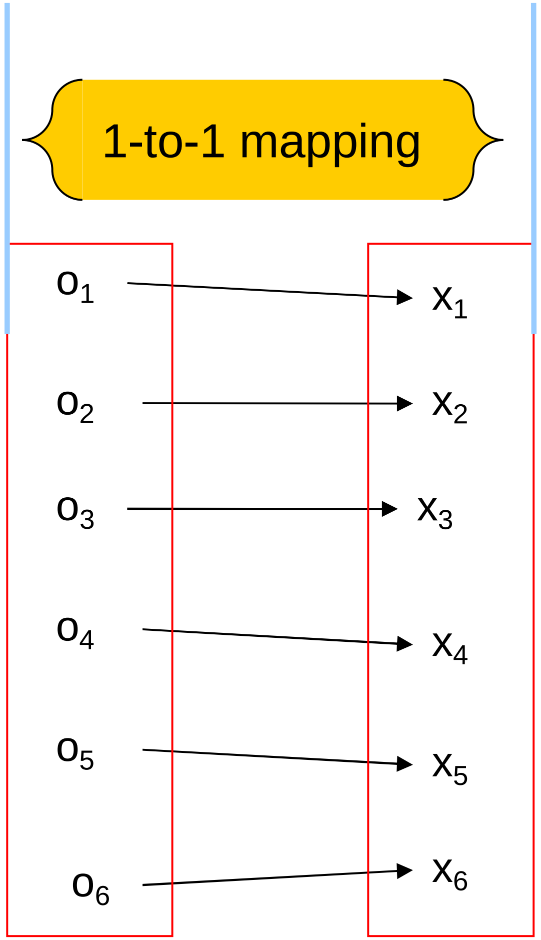
Context Sensitive Flow Graph

- Can you find anything interesting from the *group view* of the flow graph?

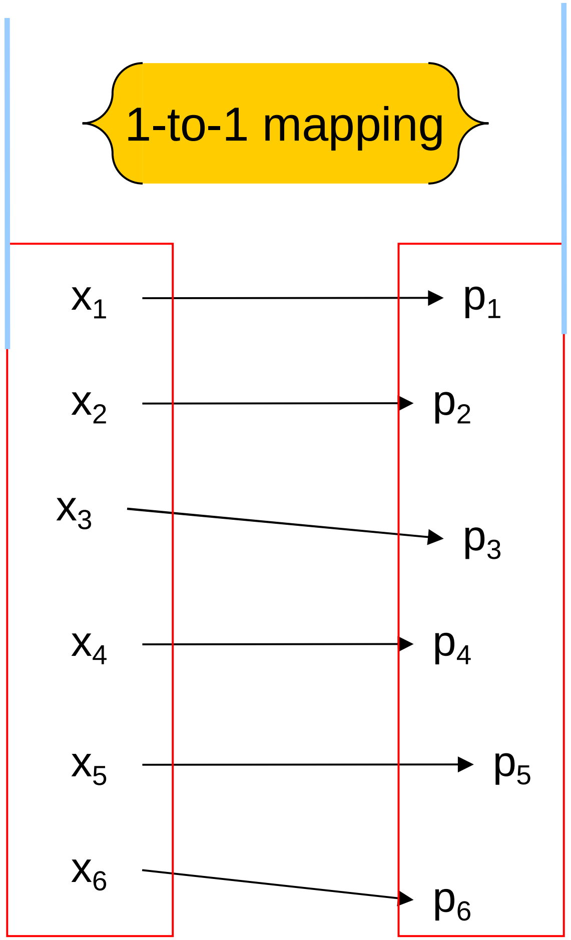
Context Sensitive Flow Graph

- Yes, the flow graph is constructed by many mapping structures.

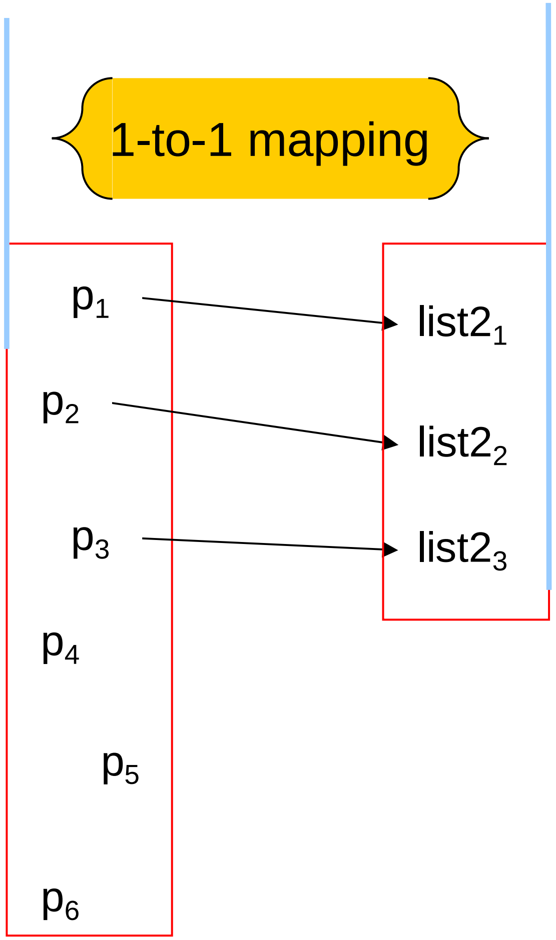
Context Sensitive Flow Graph



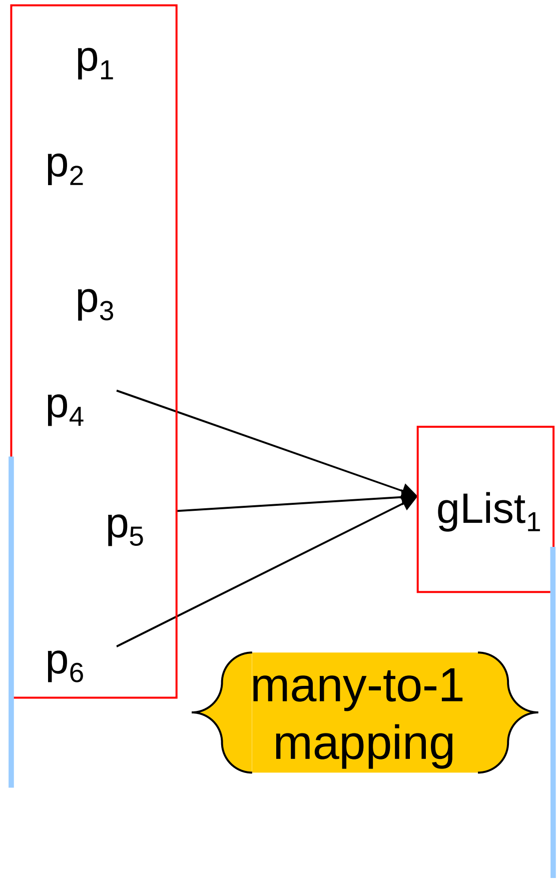
Context Sensitive Flow Graph



Context Sensitive Flow Graph

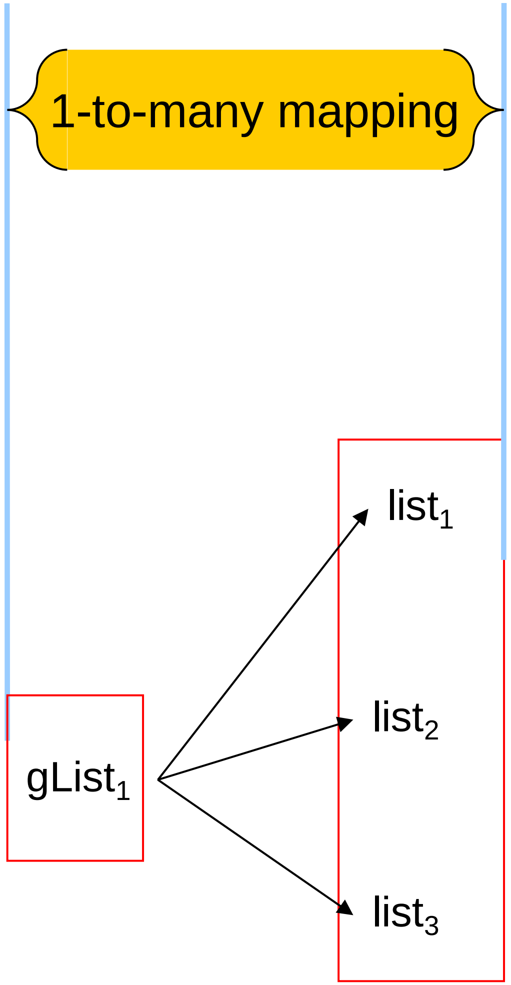


Context Sensitive Flow Graph



many-to-1 mapping

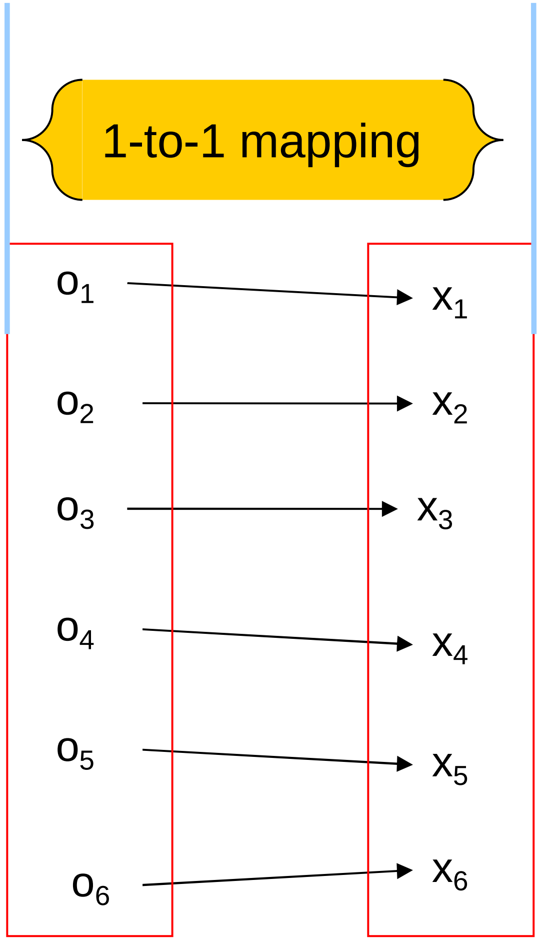
Context Sensitive Flow Graph



Context Sensitive Flow Graph

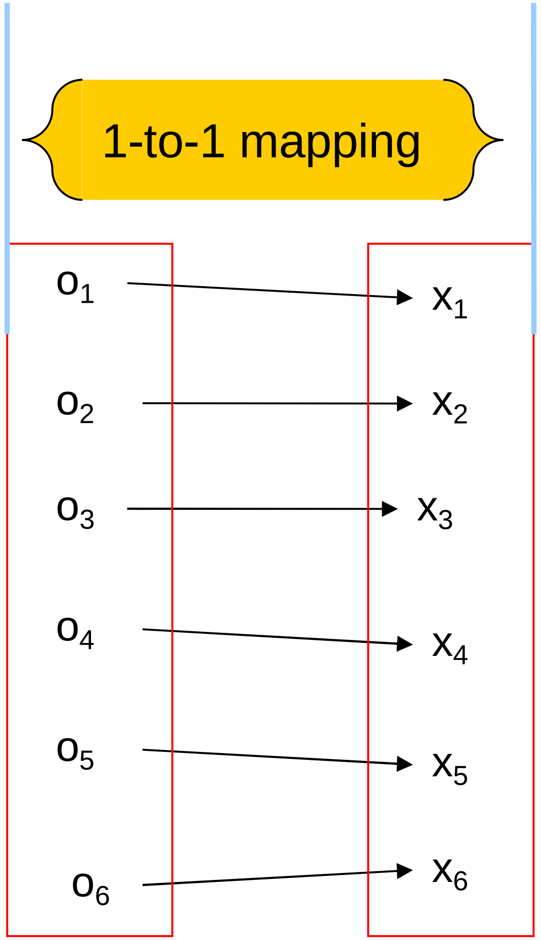
- Any other observations?

Context Sensitive Flow Graph



Please look at the subscripts....

Context Sensitive Flow Graph

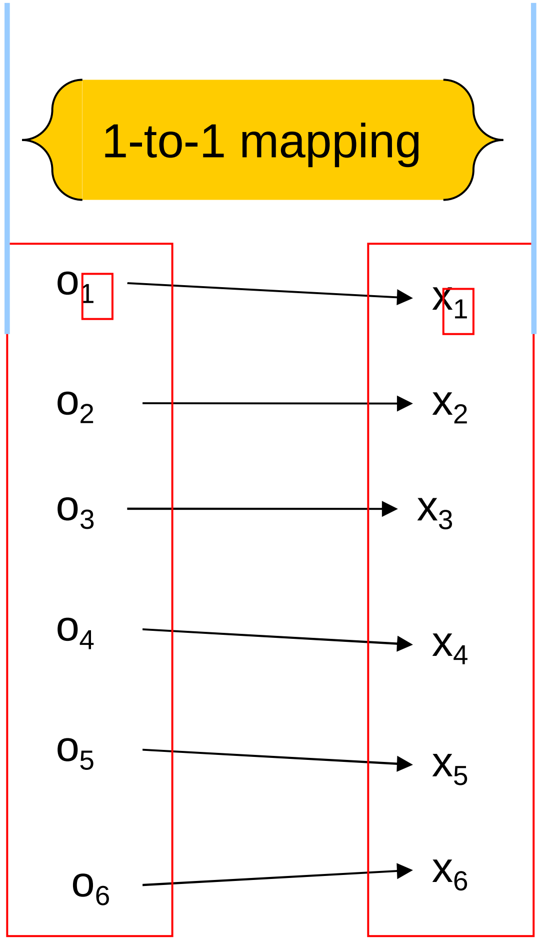


Please look at the subscripts....

o : [1, 6]
x : [1, 6]

Consecutive numbers

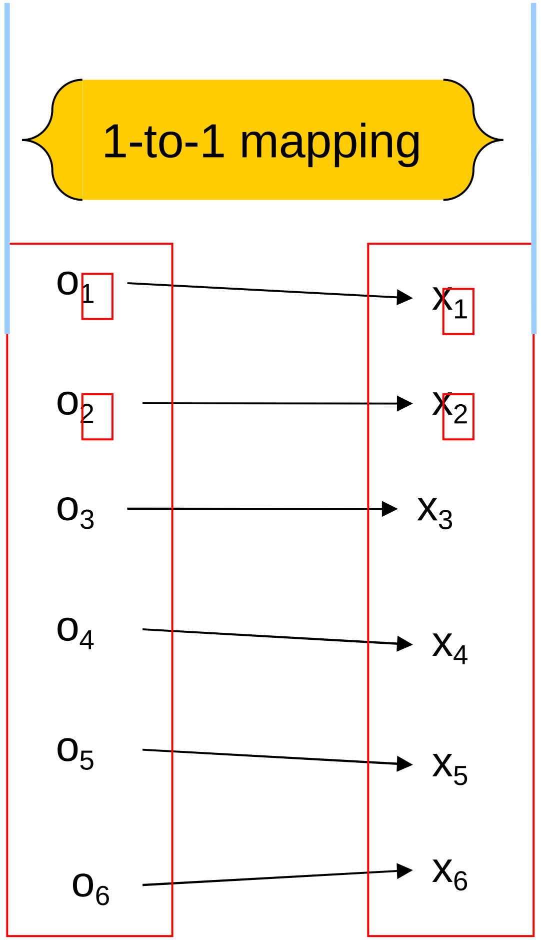
Context Sensitive Flow Graph



Please look at the subscripts....

1 -> 1

Context Sensitive Flow Graph

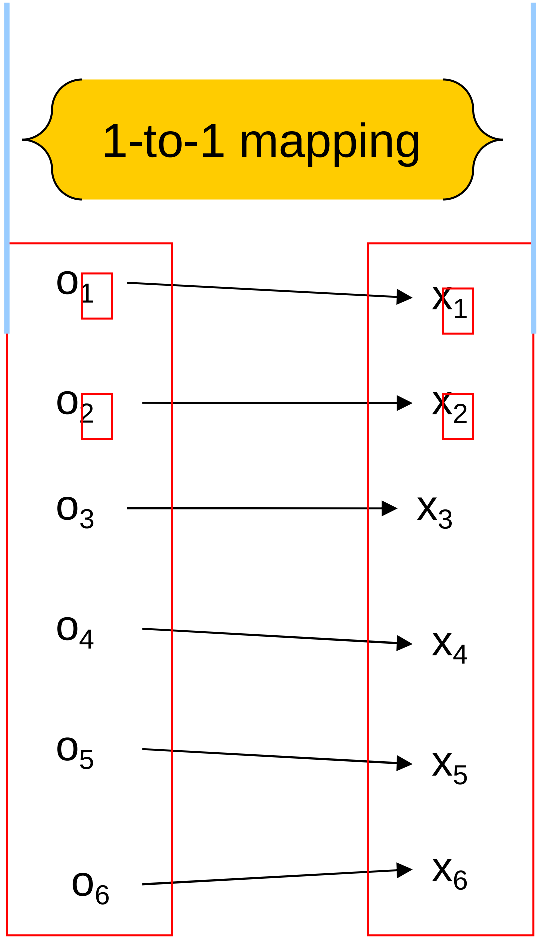


Please look at the subscripts....

1 -> 1

2 -> 2

Context Sensitive Flow Graph



Please look at the subscripts....

1 -> 1

2 -> 2

……………..

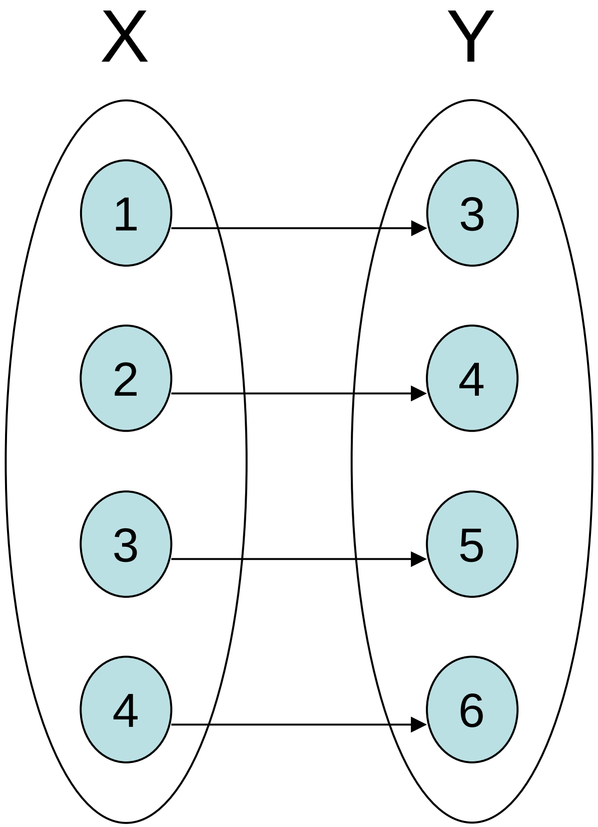
Context Sensitive Flow Graph

- We call this *ordering law*:
  - – Y = map(X);
  - – Elements in X are numbered consecutively;
  - – Elements in Y are numbered consecutively;
  - – a < b in X   => map(a) < map(b).

Context Sensitive Flow Graph

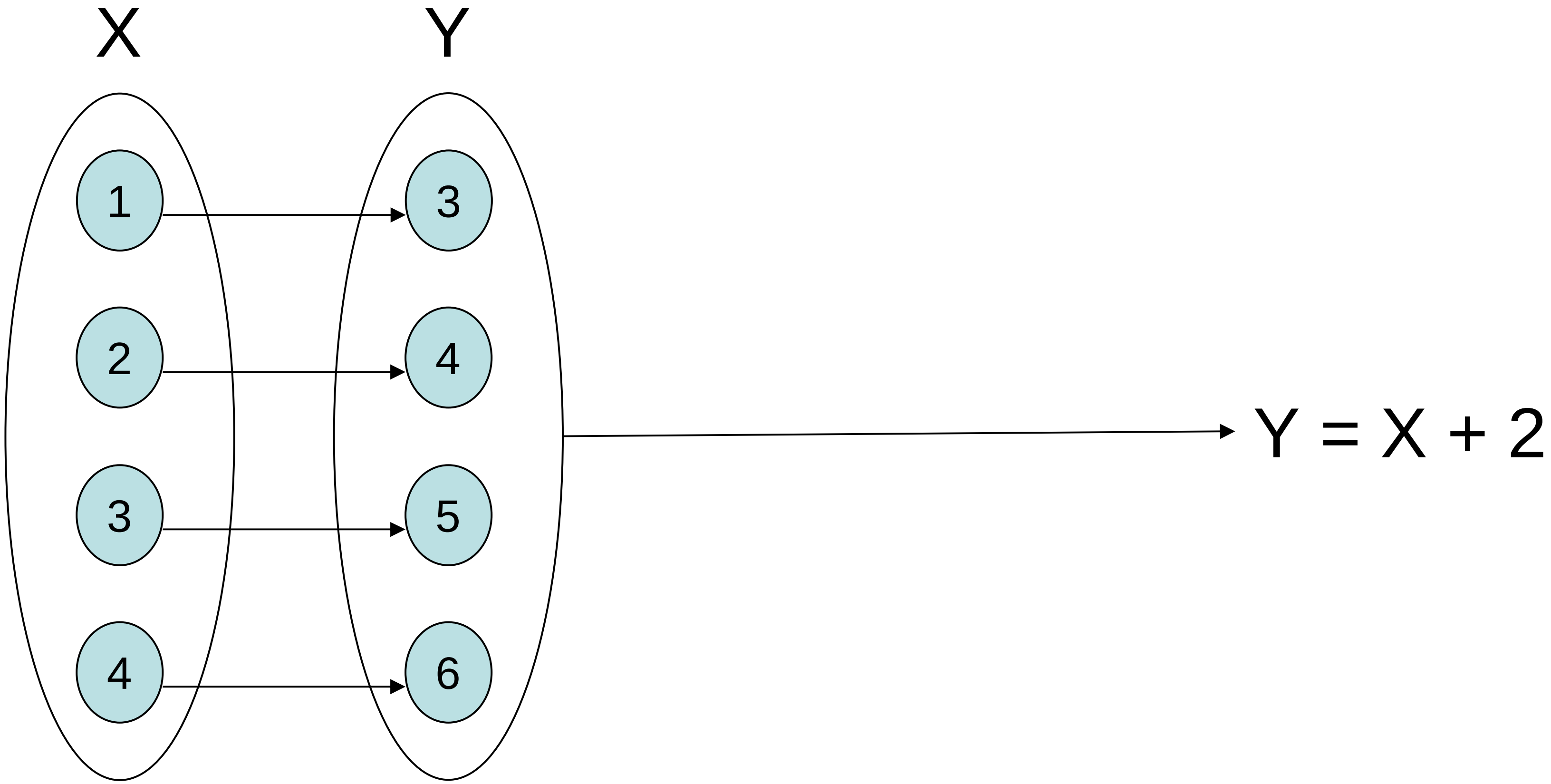
- Is it useful for our points-to analysis?

Geometric Encoding

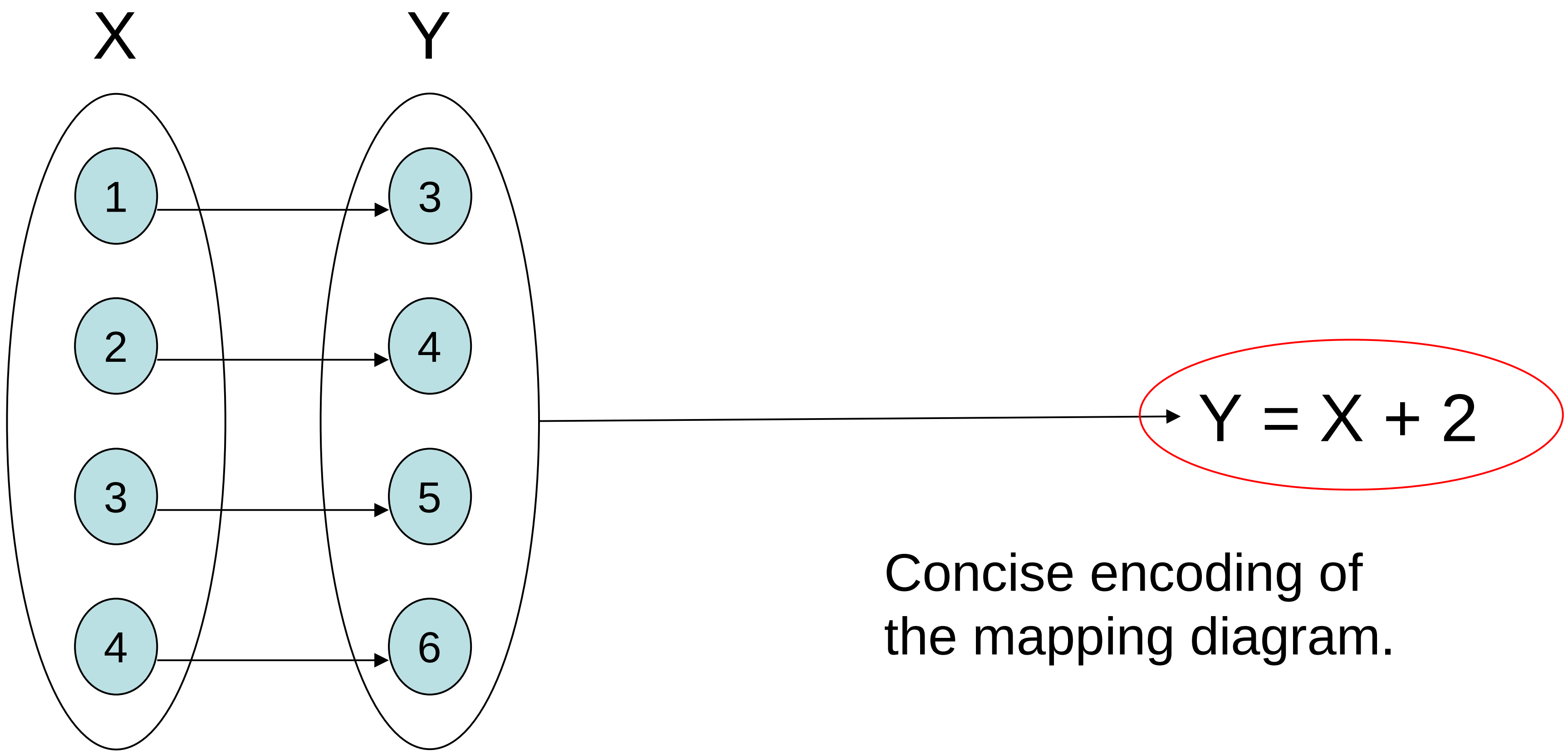


1-to-1 mapping with ordering law.......

Geometric Encoding



Geometric Encoding



Concise encoding of the mapping diagram.

Geometric Encoding

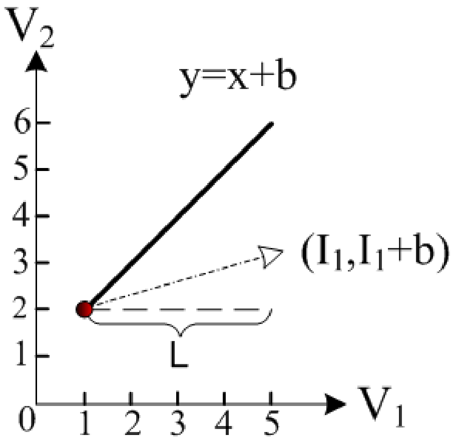
- Next, we visualize all the mapping relations that conform to the ordering law.

Geometric Encoding

1-to-1 mapping

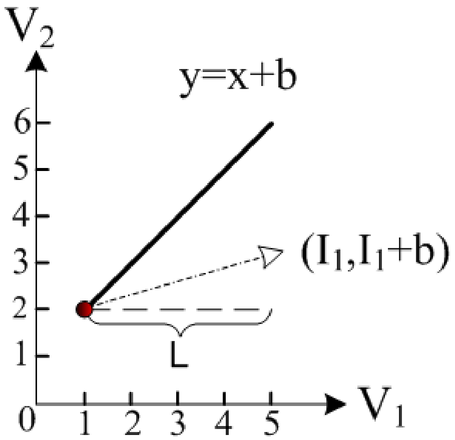
Geometric Encoding

1-to-1 mapping



Geometric Encoding

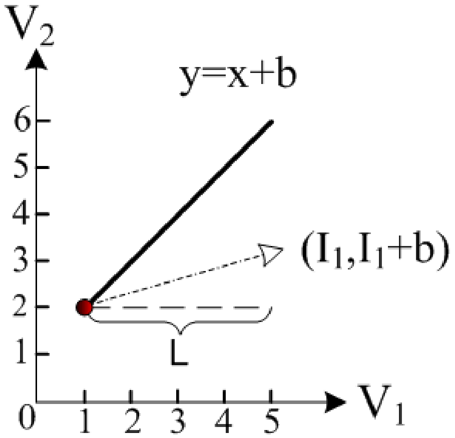
1-to-1 mapping



$(I_1, I_1 + b, L)$

Geometric Encoding

1-to-1 mapping

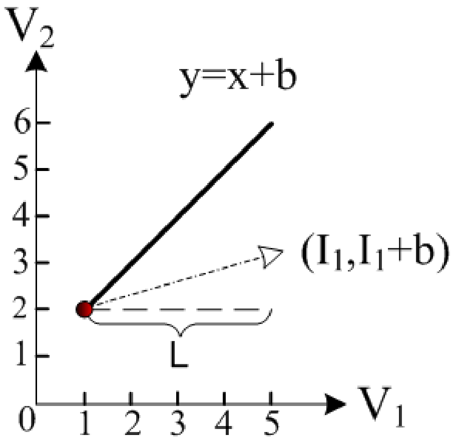


$(I_1, I_1 + b, L)$

many-to-many
(1-to-many,
many-to-1)

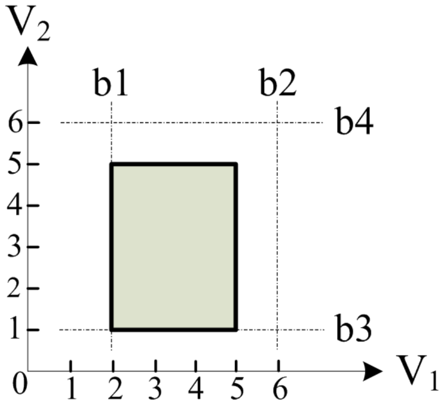
Geometric Encoding

1-to-1 mapping



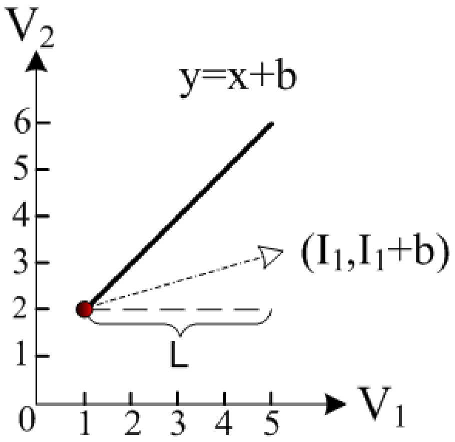
$(I_1, I_1 + b, L)$

many-to-many (1-to-many, many-to-1)



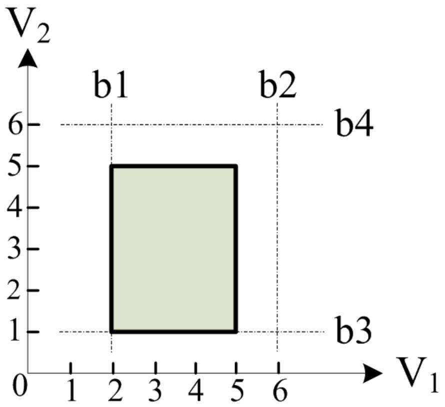
Geometric Encoding

1-to-1 mapping



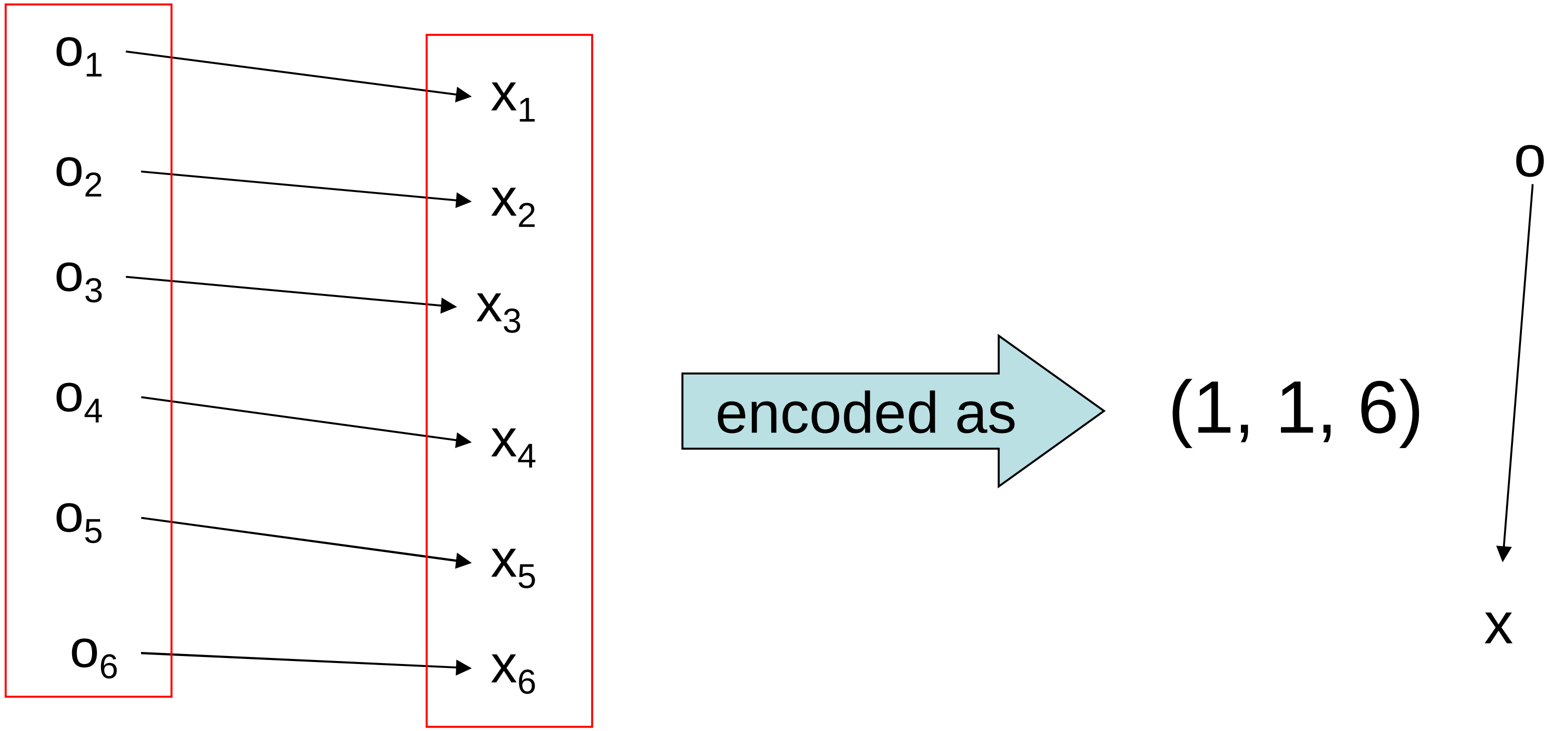
$(I_1, I_1 + b, L)$

many-to-many (1-to-many, many-to-1)

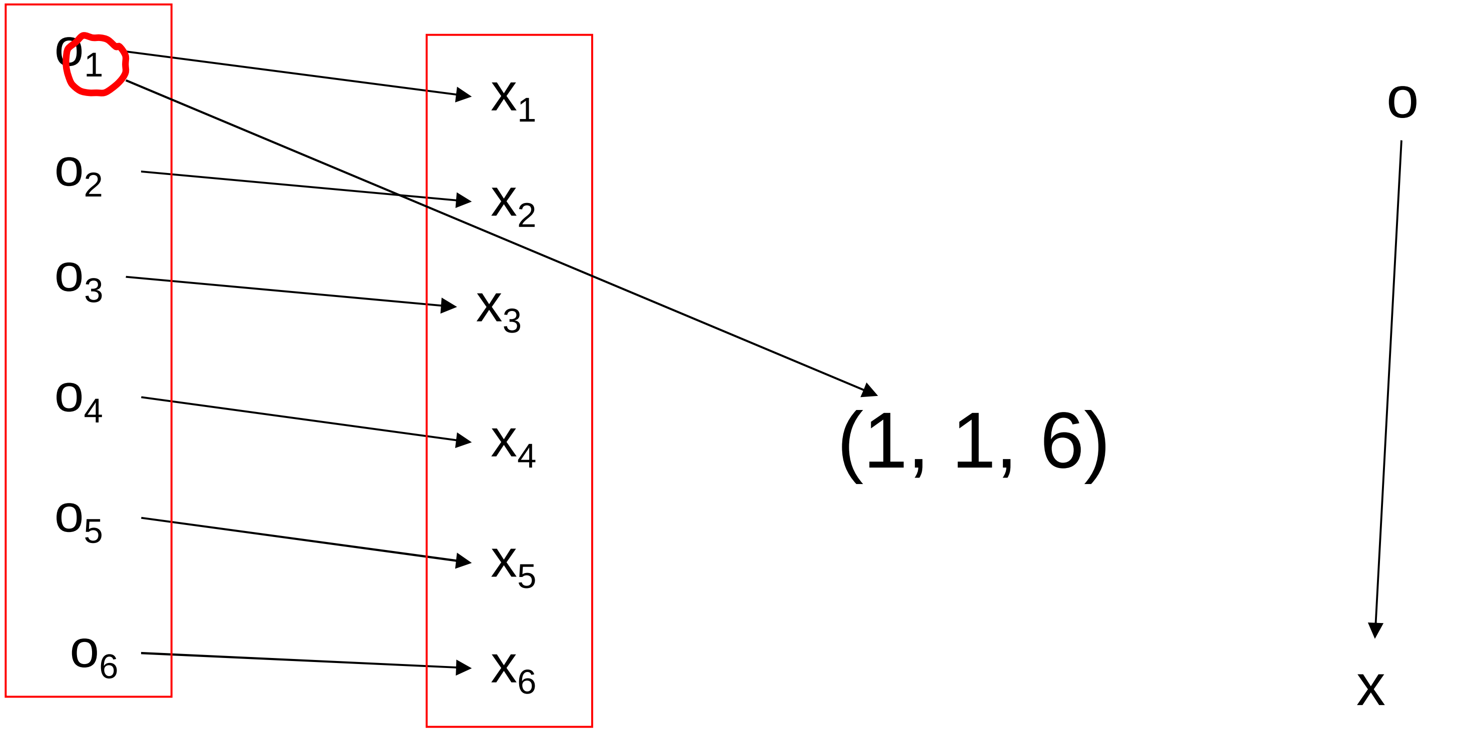


$(b_1, b_3, b_2 - b_1, b_4 - b_3)$

Geometric Encoding



Geometric Encoding



o_1 o_2 o_3 o_4 o_5 o_6

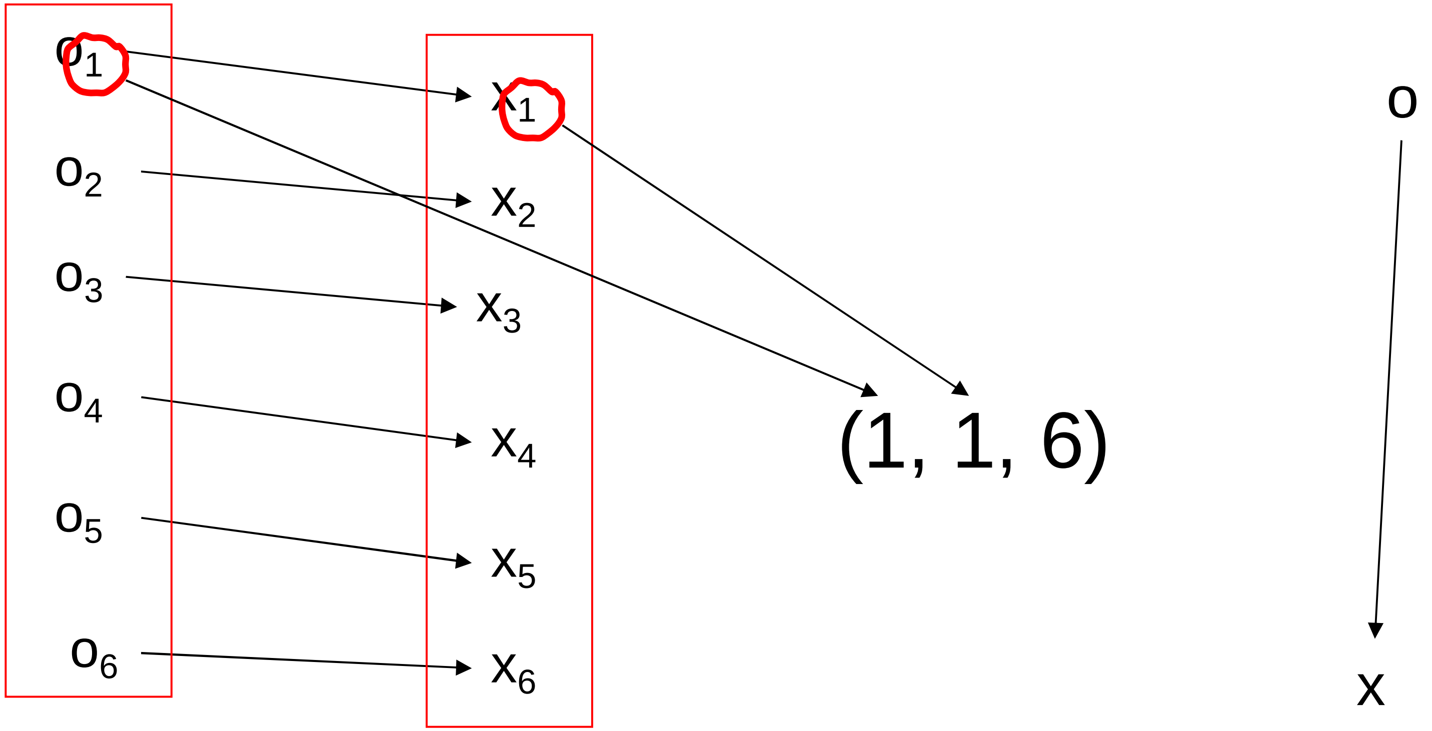
x_1 x_2 x_3 x_4 x_5 x_6

(1, 1, 6)

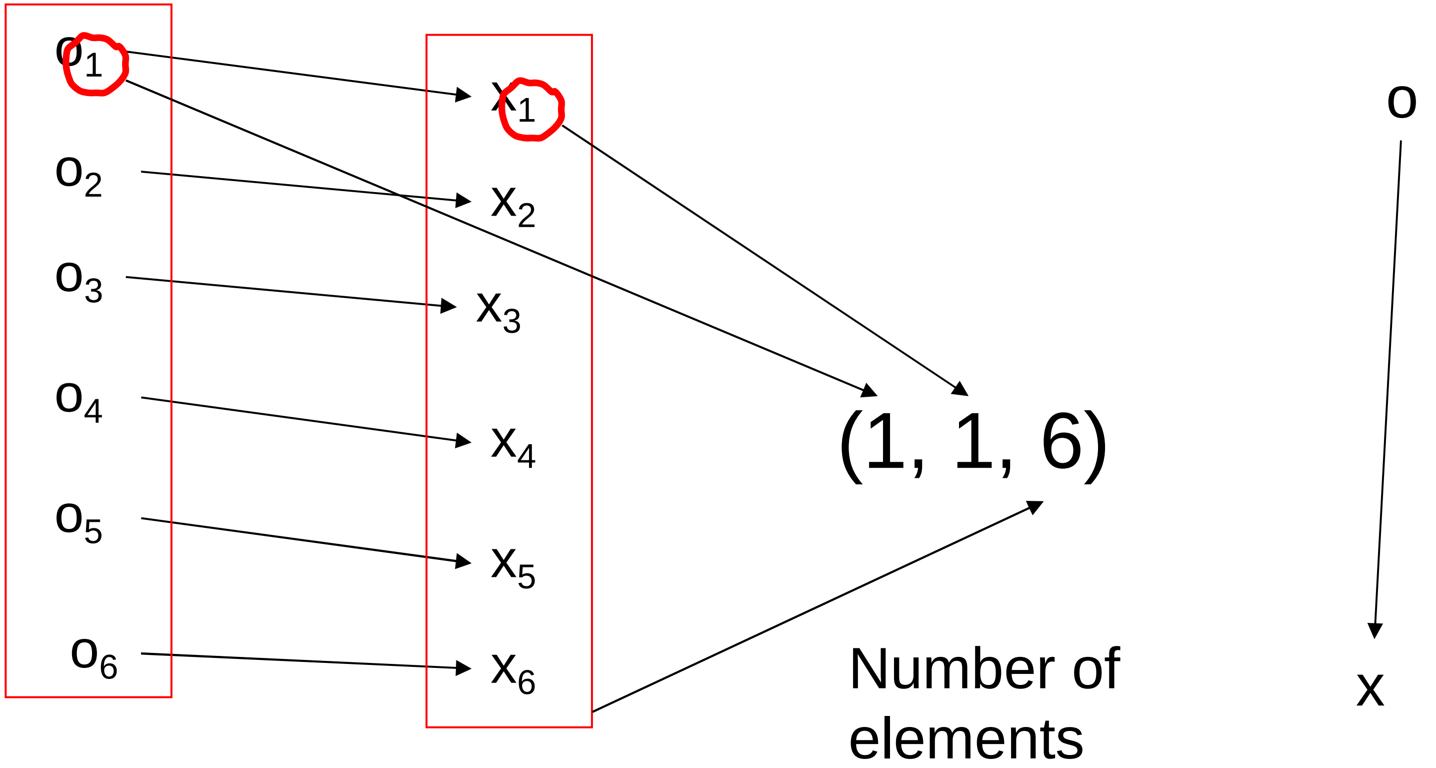
o

x

Geometric Encoding



Geometric Encoding



Geometric Encoding

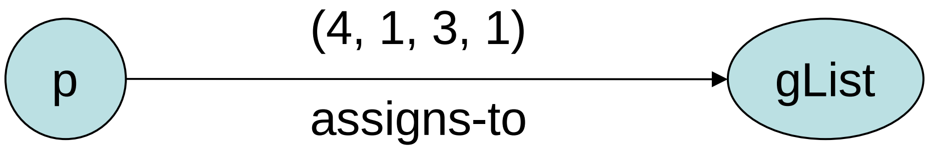
How can we do reachability analysis with the flow graph compressed by geometric encoding?

Inference Rules

- Pointer assignment:

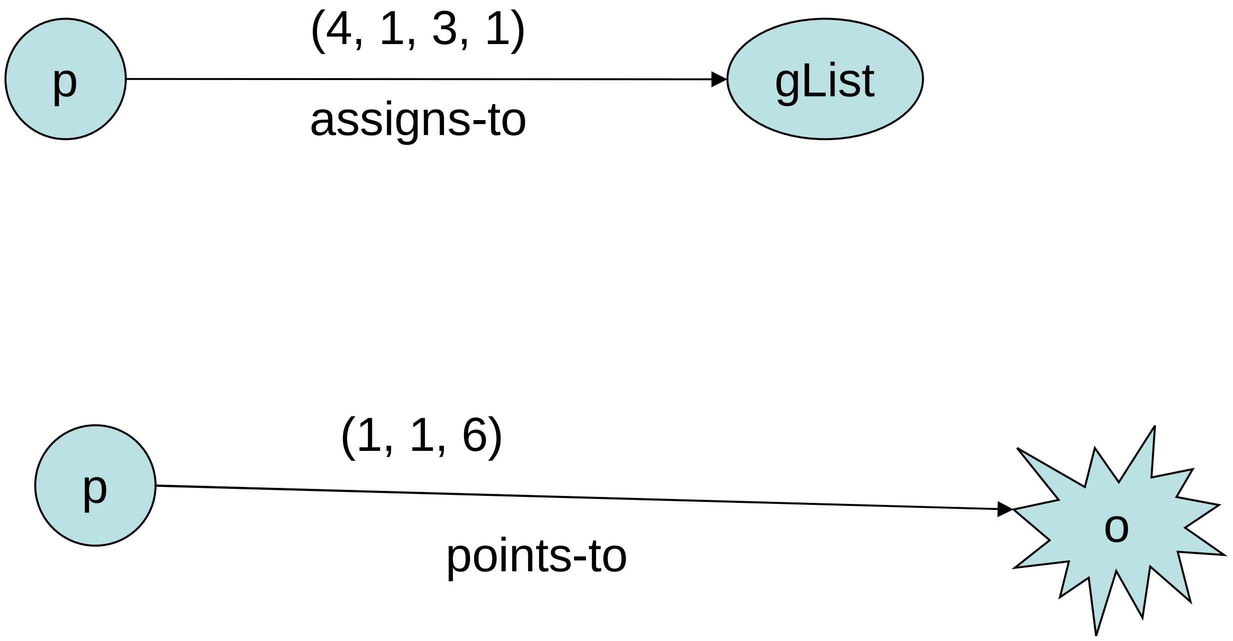
Inference Rules

- Pointer assignment:



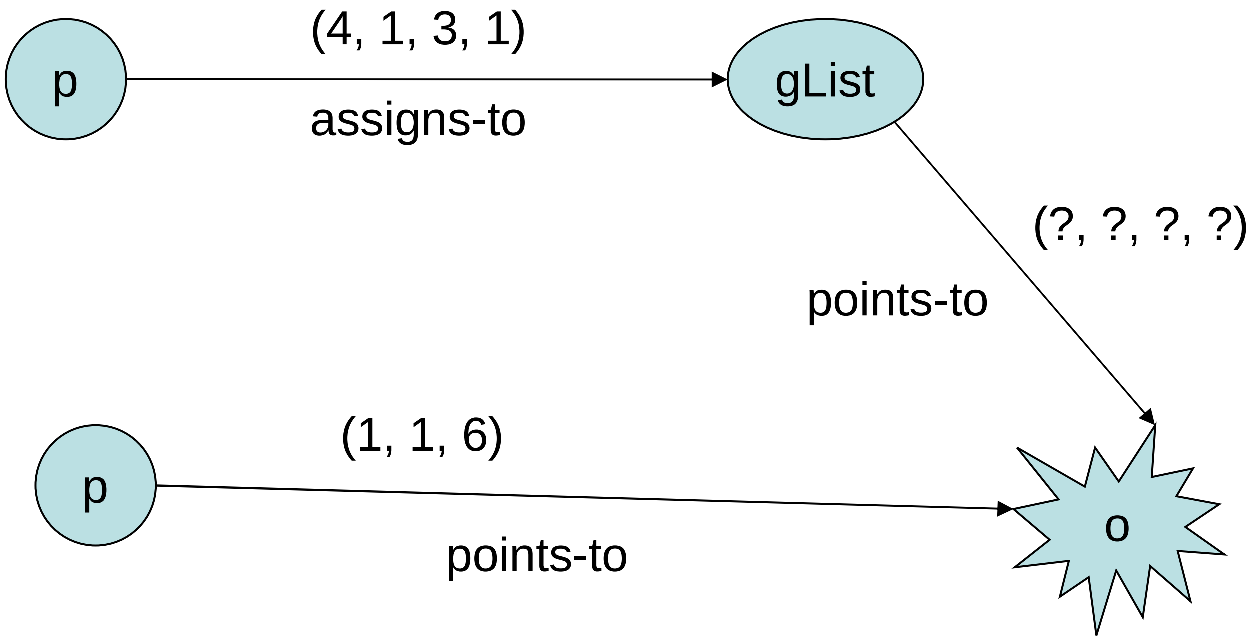
Inference Rules

- Pointer assignment:

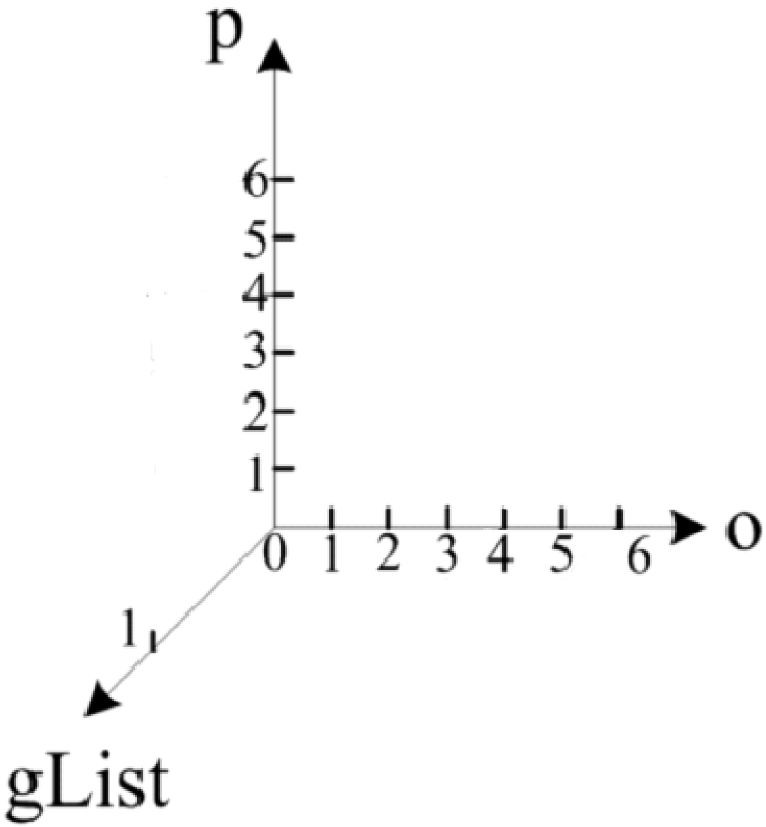


Inference Rules

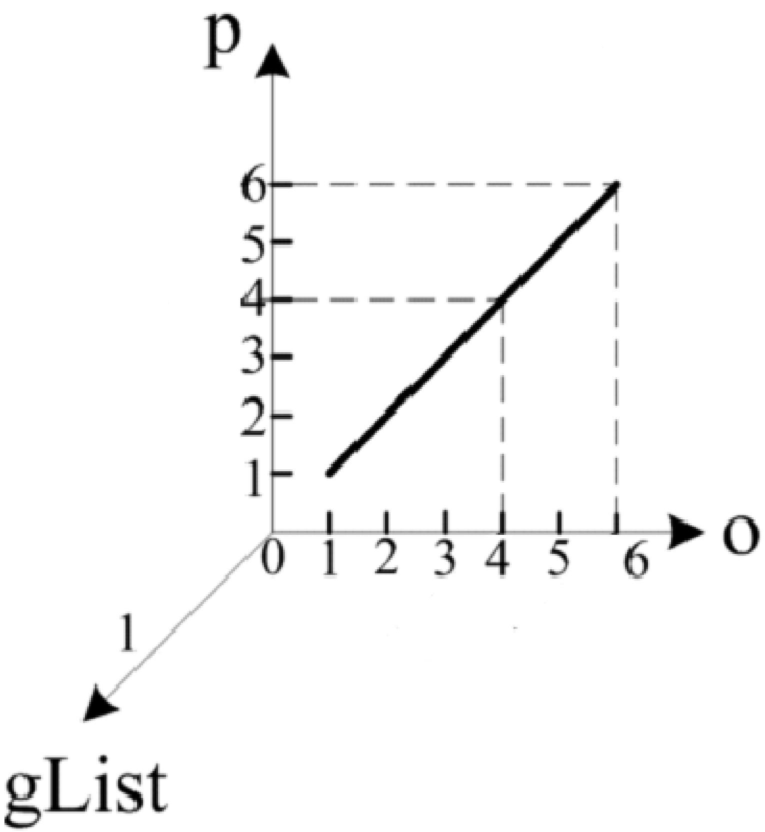
- Pointer assignment:

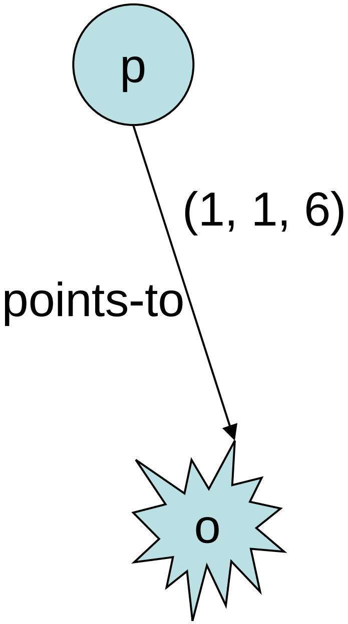


Inference Rules

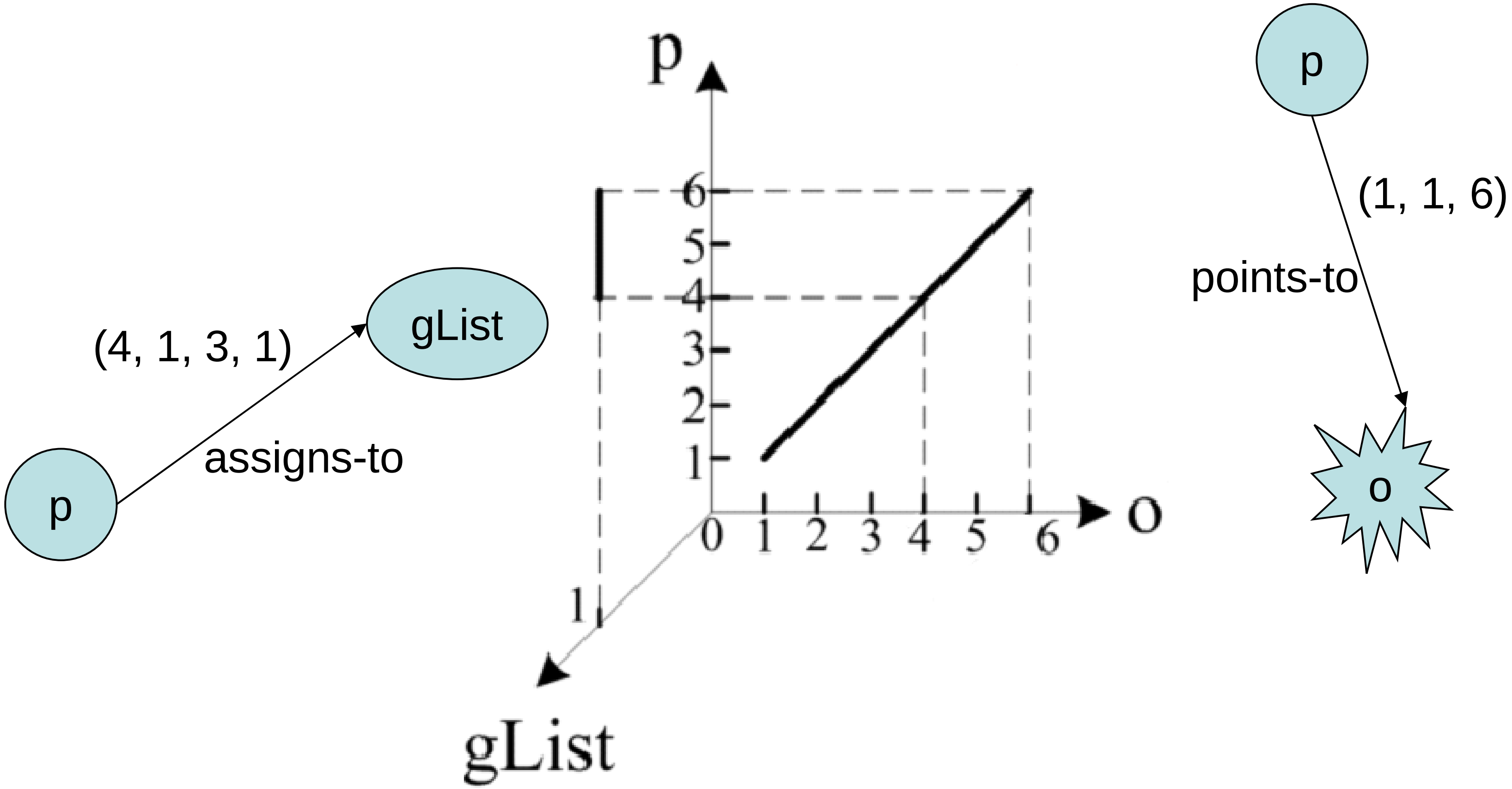


Inference Rules

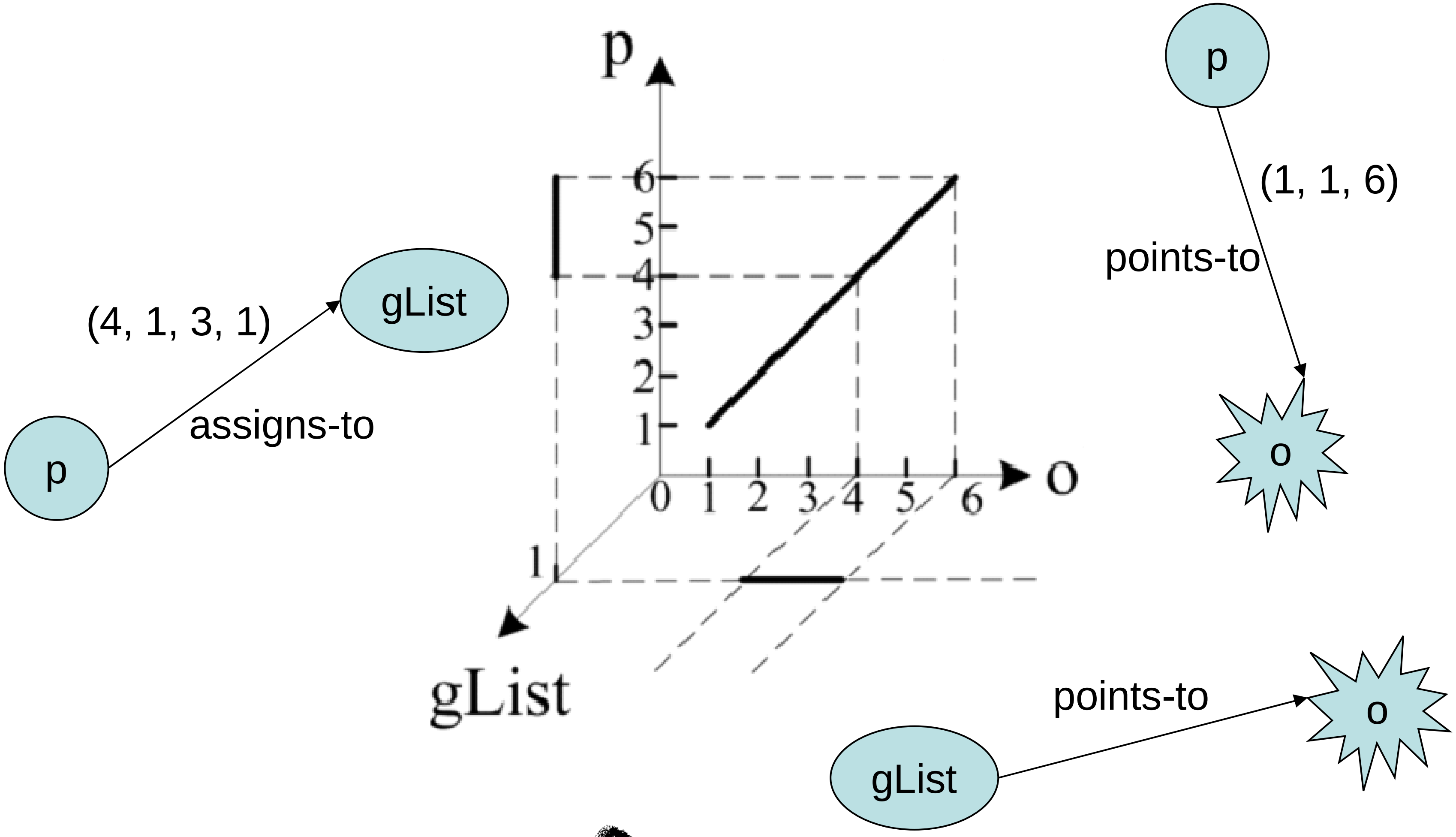




Inference Rules



Inference Rules



p

(4, 1, 3, 1)

assigns-to

gList

p

(1, 1, 6)

points-to

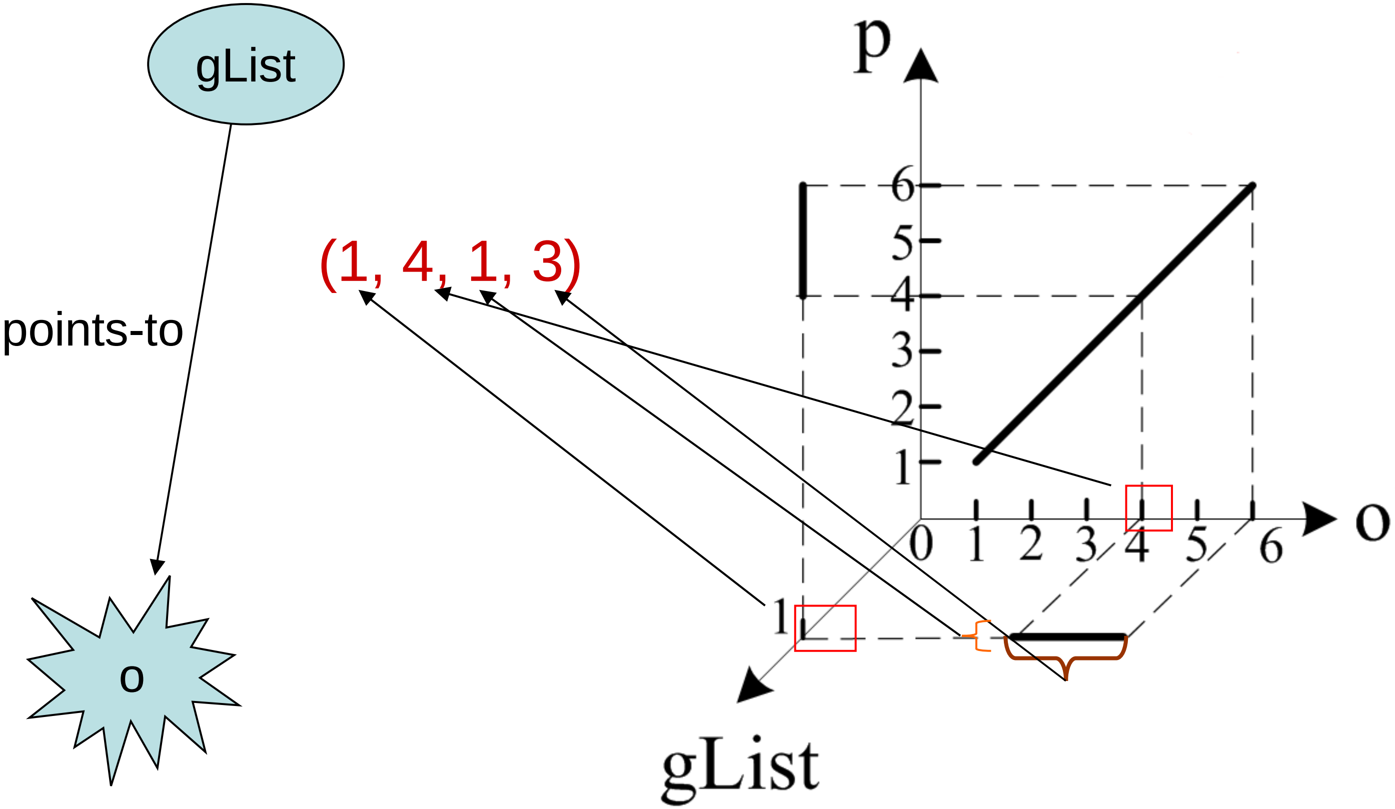
o

gList

points-to

o

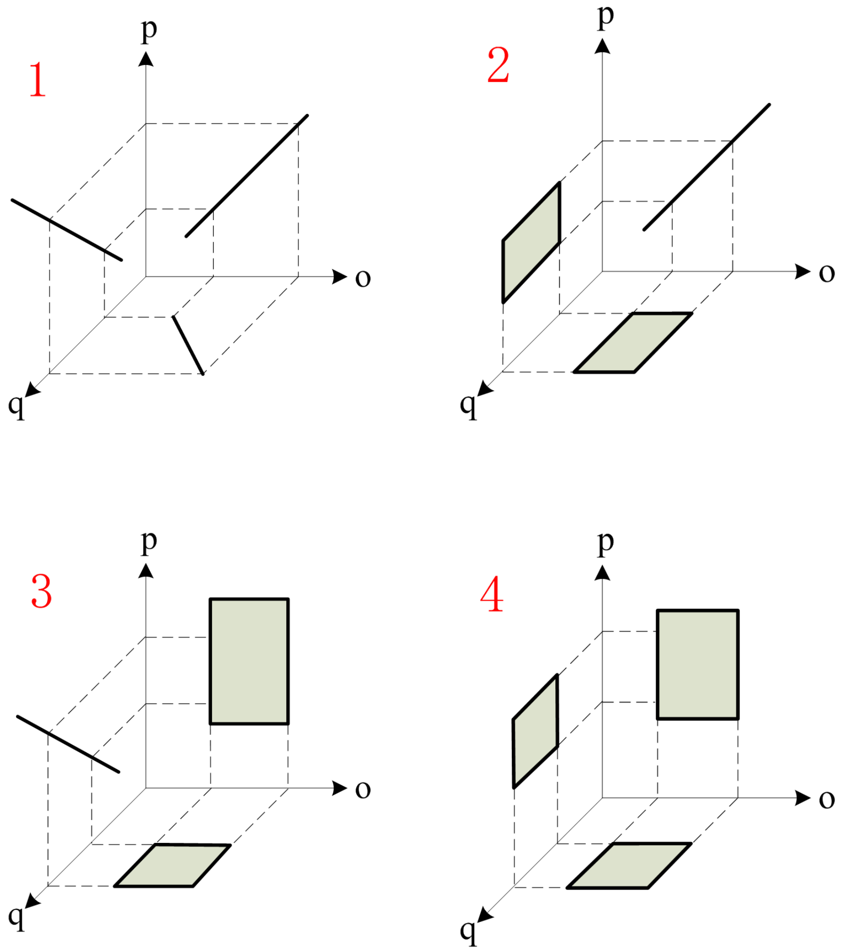
Inference Rules



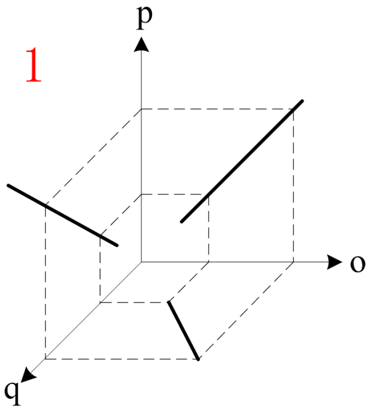
Inference Rules

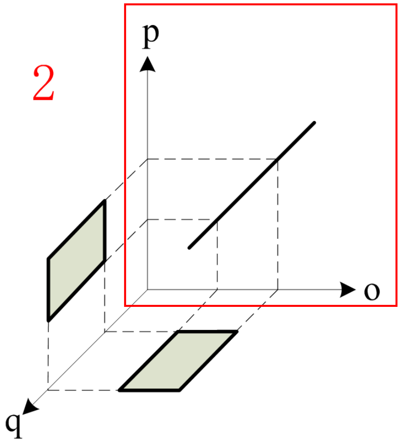
- Pointer assignment inference problem:
  - Input: p = q, p -> o
  - Infer: q -> o

Inference Rules

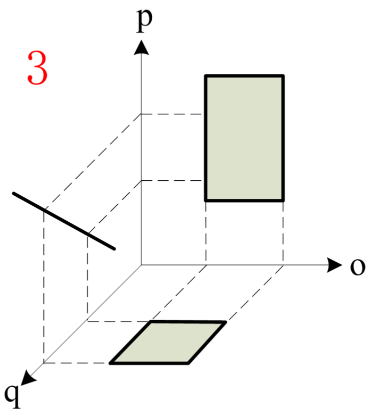


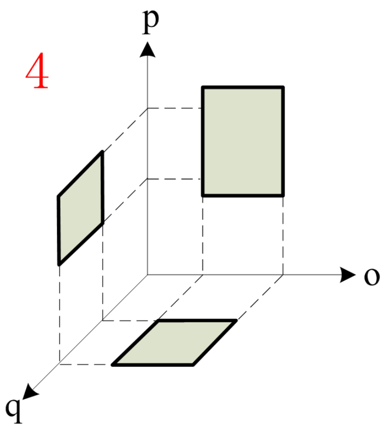
Inference Rules





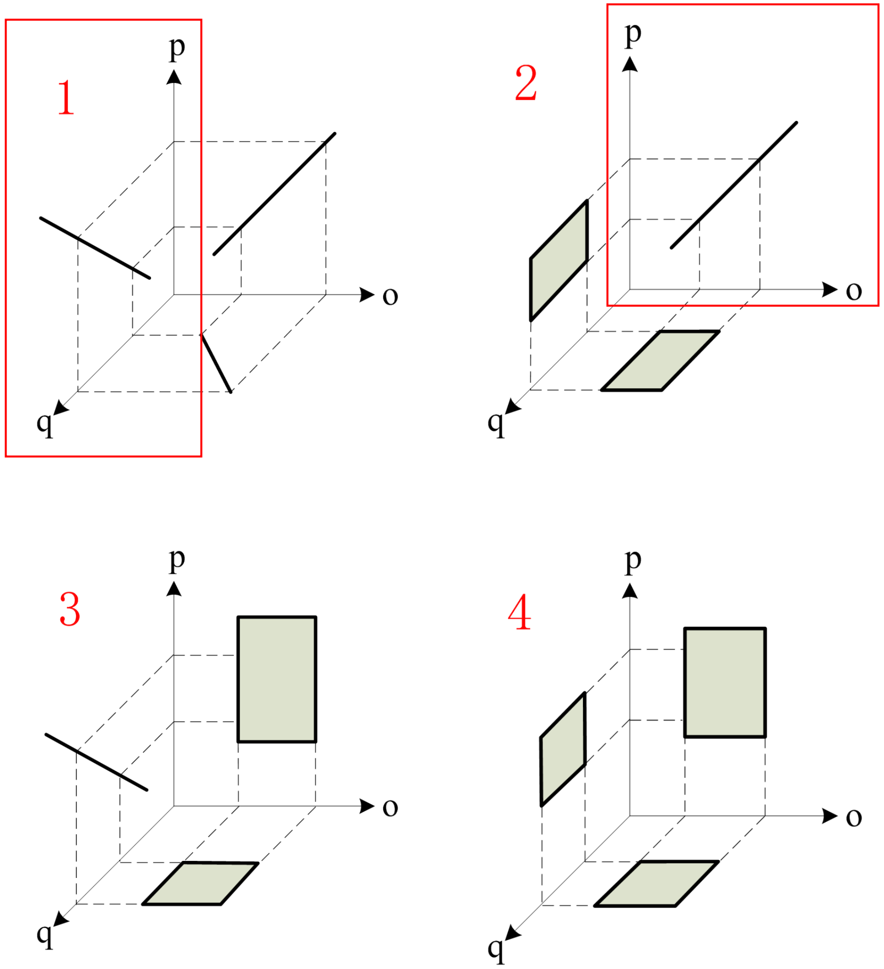
p points-to o





Inference Rules

p assigns-to q

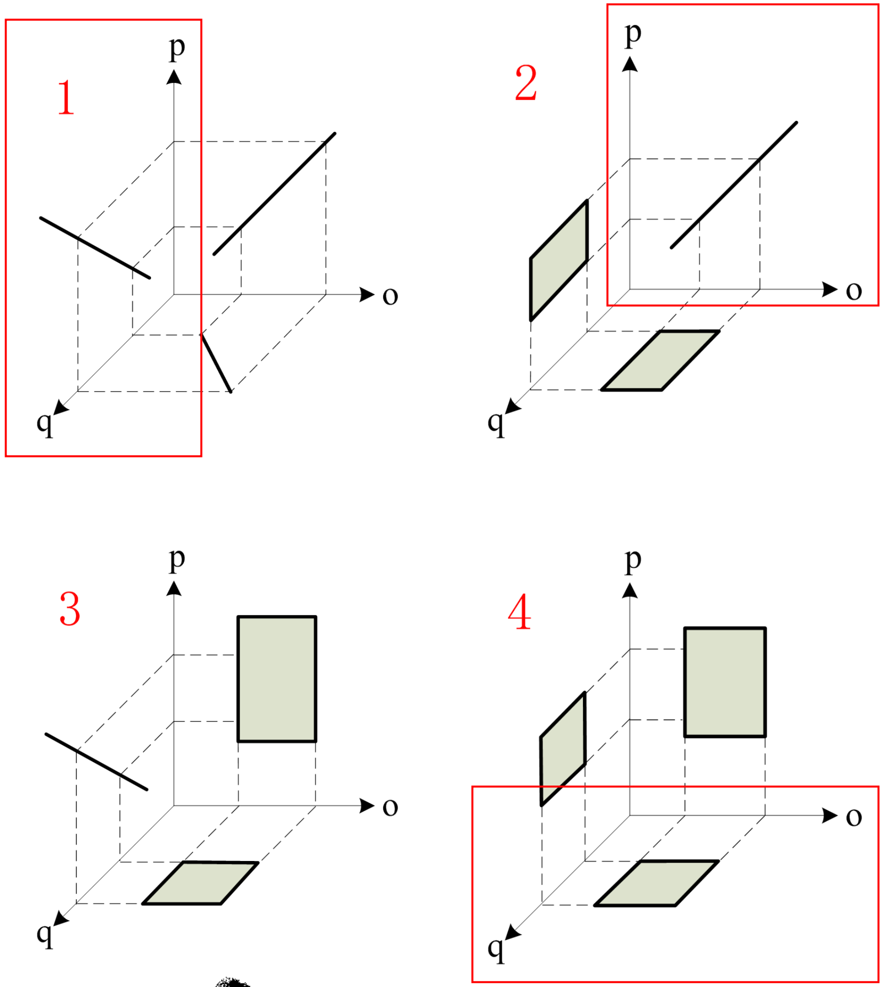


p points-to o

Inference Rules

p assigns-to q

p points-to o



q points-to o

Inference Rules

- Pointer Dereference:
  - a.k.a complex constraints instantiation
  - e.g.   p = q.f

- Please read our paper for details...

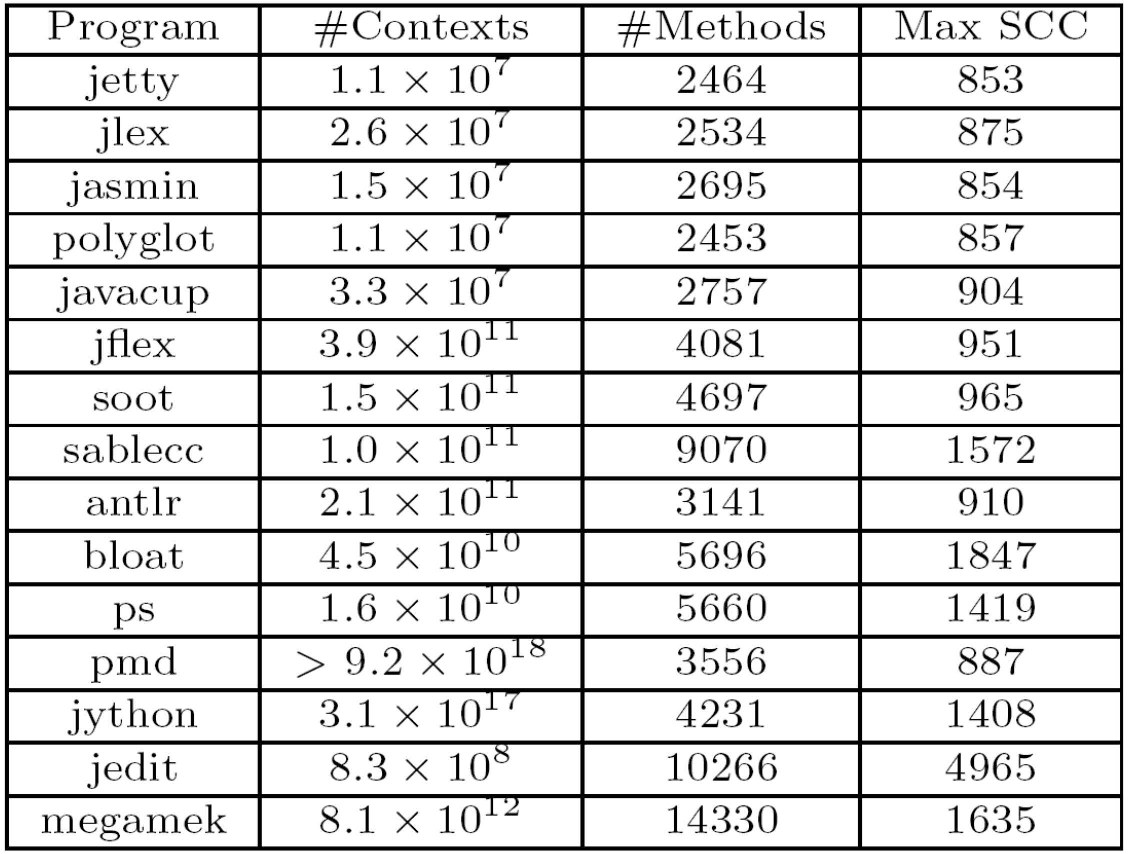
Experiment

- Questions:
  - Performance?
  - Precision?

Experimental Configuration

- Evaluated algorithm:
  - Two encoding instances: *Geom* and *HeapIns*
- State of the art:
  - Paddle 1-Object Sensitivity
  - Both BDD and worklist based
- Execution environment:
  - Soot 2.4.0
  - JDK 1.3.1_20 (analysis library)
  - JRockit 28.1 (backbone our algorithms and Paddle)
  - Intel Xeon 3.0 G
  - 15G RAM

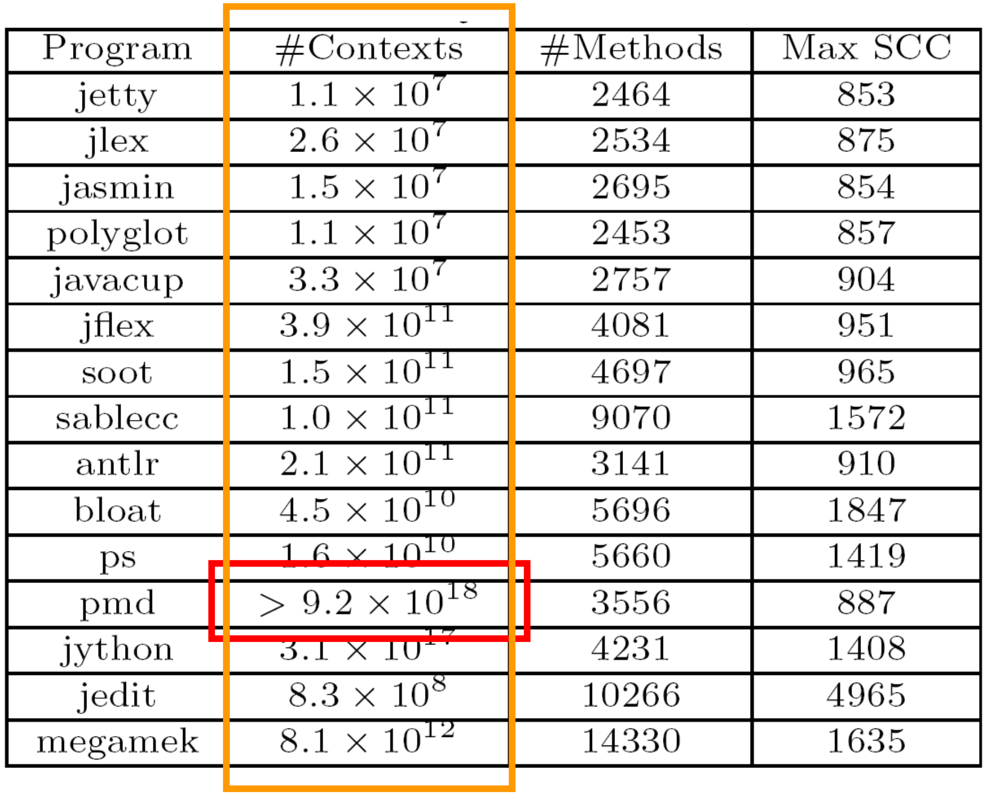
Benchmark



Program	#Contexts	#Methods	Max SCC
jetty	1.1×10^{7}	2464	853
jlex	2.6×10^{7}	2534	875
jasmin	1.5×10^{7}	2695	854
polyglot	1.1×10^{7}	2453	857
javacup	3.3×10^{7}	2757	904
jflex	3.9×10^{11}	4081	951
soot	1.5×10^{11}	4697	965
sablecc	1.0×10^{11}	9070	1572
antlr	2.1×10^{11}	3141	910
bloat	4.5×10^{10}	5696	1847
ps	1.6×10^{10}	5660	1419
pmd	$> 9.2 \times 10^{18}$	3556	887
jython	3.1×10^{17}	4231	1408
jedit	8.3×10^{8}	10266	4965
megamek	8.1×10^{12}	14330	1635

Benchmark

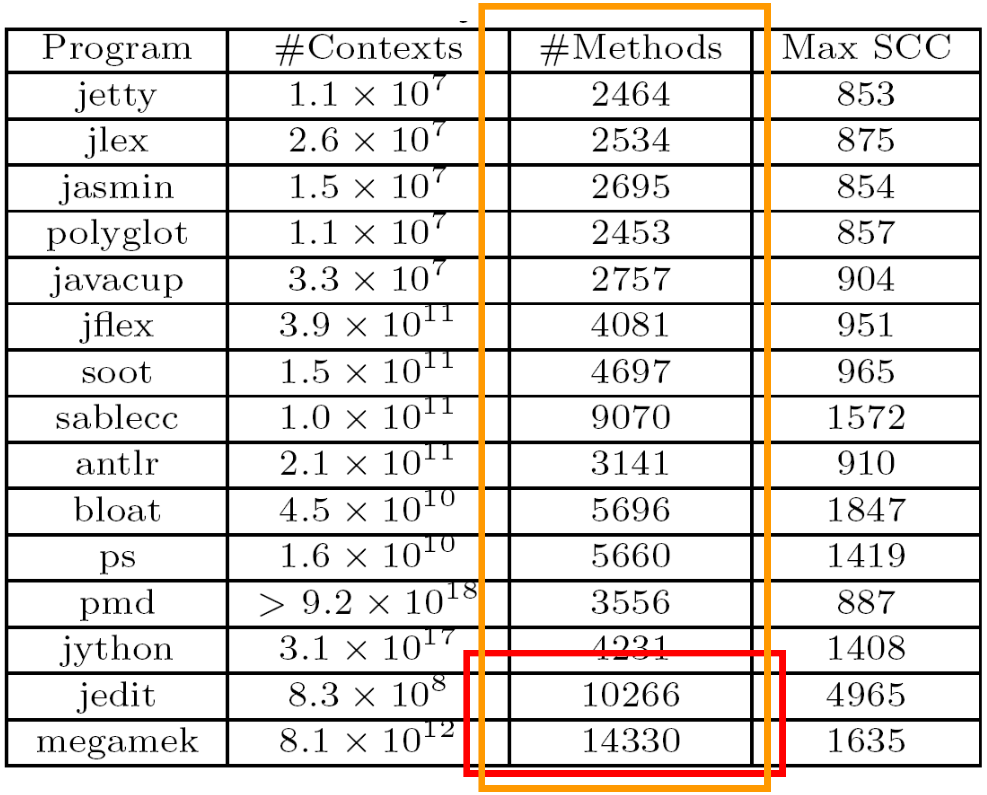
- The number of contexts is very large!
- 9.2 * 10^{18} = 2^63



Program	#Contexts	#Methods	Max SCC
jetty	1.1×10^7	2464	853
jlex	2.6×10^7	2534	875
jasmin	1.5×10^7	2695	854
polyglot	1.1×10^7	2453	857
javacup	3.3×10^7	2757	904
jflex	3.9×10^{11}	4081	951
soot	1.5×10^{11}	4697	965
sablecc	1.0×10^{11}	9070	1572
antlr	2.1×10^{11}	3141	910
bloat	4.5×10^{10}	5696	1847
ps	1.6×10^{10}	5660	1419
pmd	$> 9.2 \times 10^{18}$	3556	887
jython	3.1×10^{17}	4231	1408
jedit	8.3×10^8	10266	4965
megamek	8.1×10^{12}	14330	1635

Benchmark

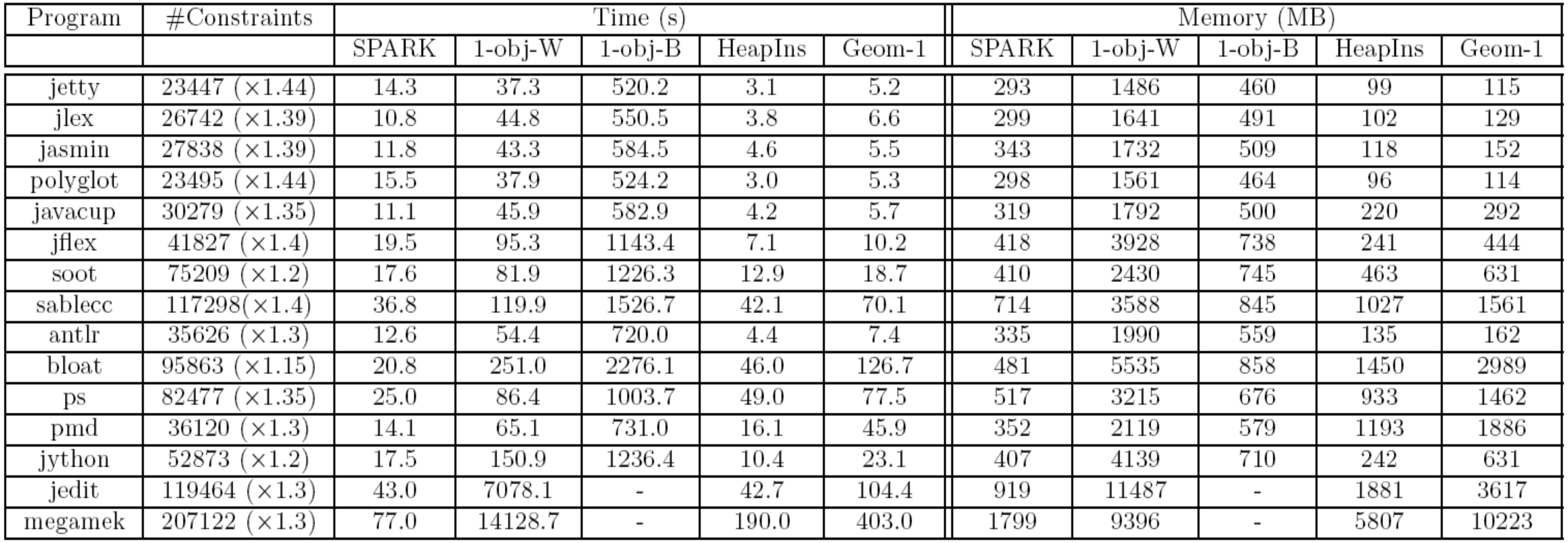
- The number of contexts is very large!
- 9.2 * 10^18 = 2^63
- The benchmark size is also very large!



Program	#Contexts	#Methods	Max SCC
jetty	1.1×10^7	2464	853
jlex	2.6×10^7	2534	875
jasmin	1.5×10^7	2695	854
polyglot	1.1×10^7	2453	857
javacup	3.3×10^7	2757	904
jflex	3.9×10^{11}	4081	951
soot	1.5×10^{11}	4697	965
sablecc	1.0×10^{11}	9070	1572
antlr	2.1×10^{11}	3141	910
bloat	4.5×10^{10}	5696	1847
ps	1.6×10^{10}	5660	1419
pmd	$> 9.2 \times 10^{18}$	3556	887
jython	3.1×10^{17}	4231	1408
jedit	8.3×10^8	10266	4965
megamek	8.1×10^{12}	14330	1635

Performance Evaluation

Table 3: Summary of the time and memory usage for all evaluated algorithms.



Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

Performance Evaluation

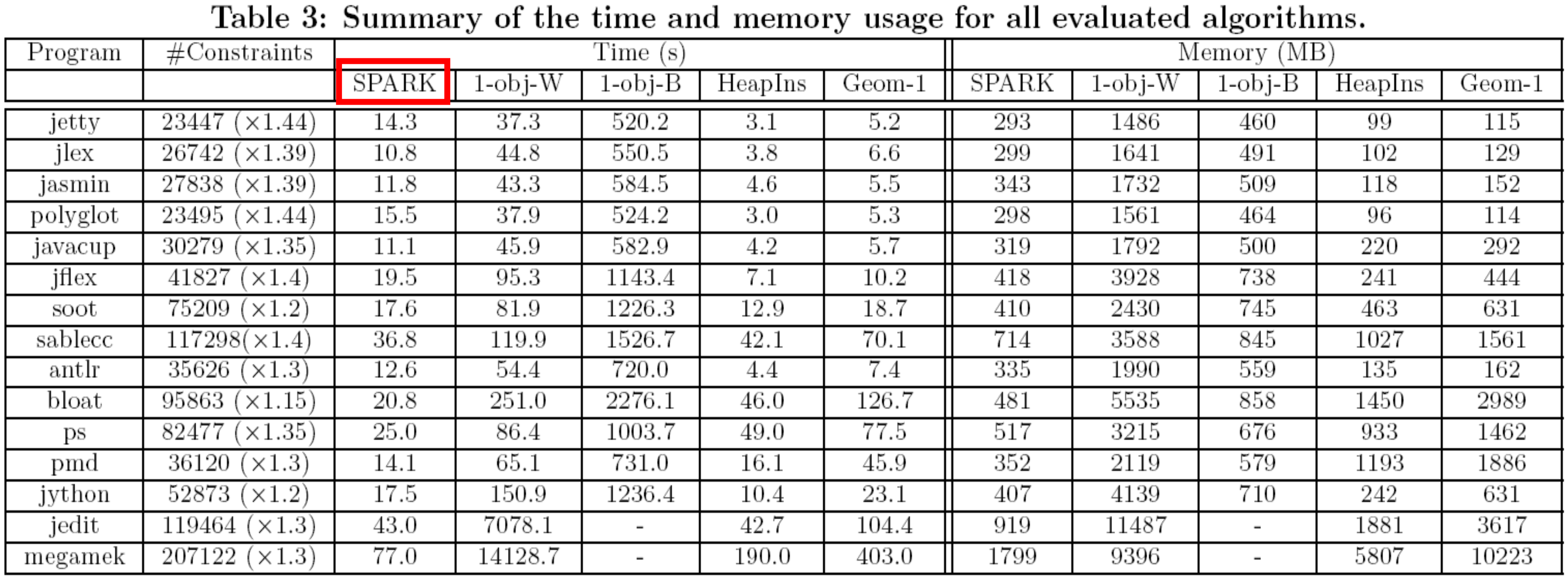


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

Performance Evaluation

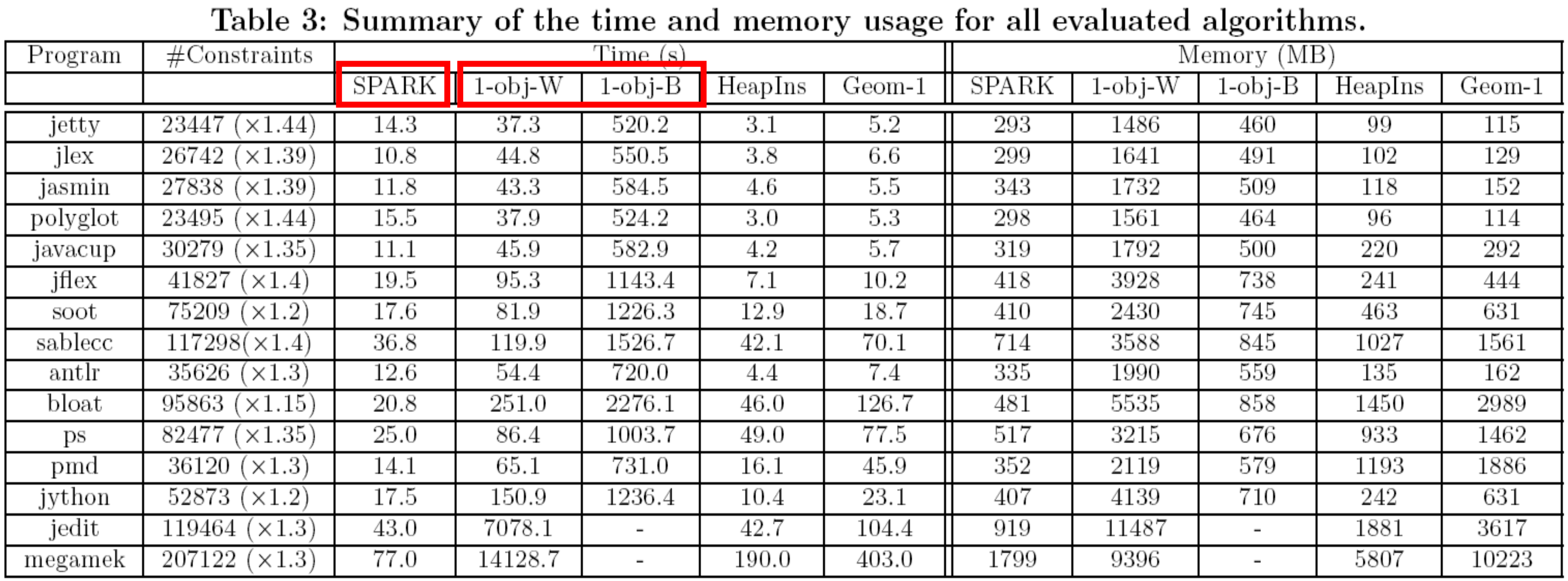


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

香港科技大學
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Performance Evaluation

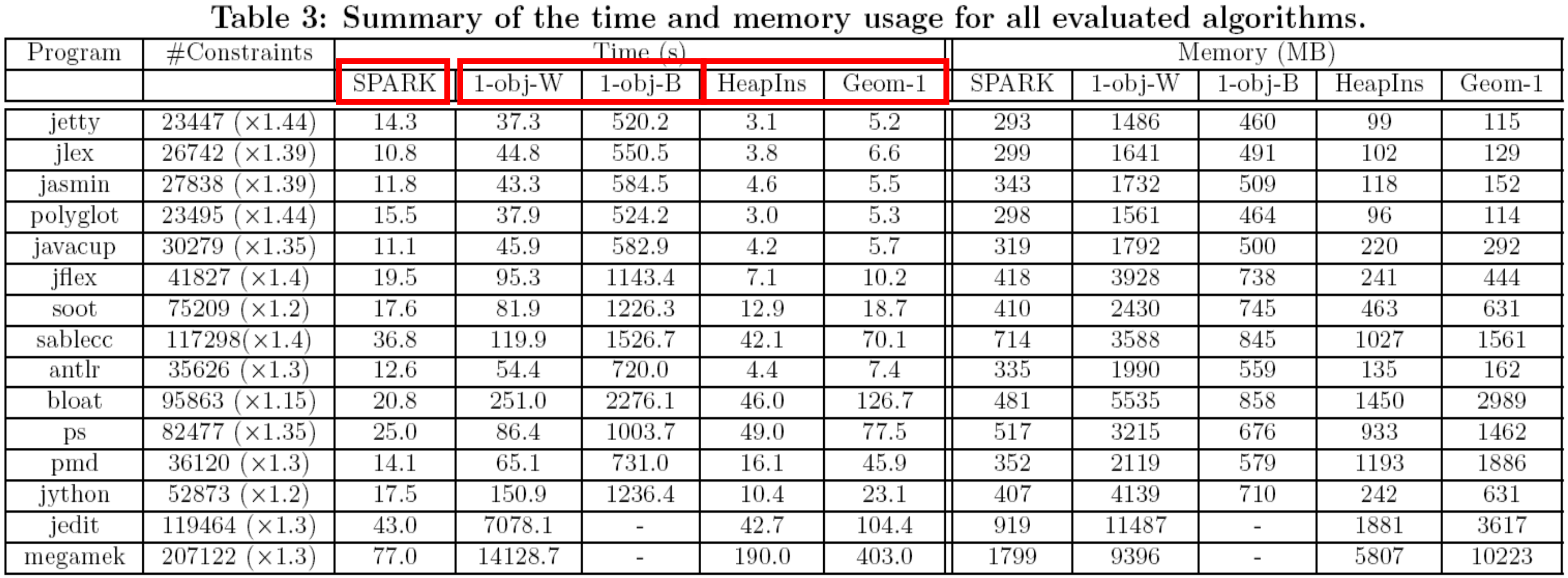


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

HeapIns: points-to analysis with simplified geometric encoding;

Geom-1: points-to analysis with geometric encoding and context sensitive modeling for the recursive calls

Performance Evaluation

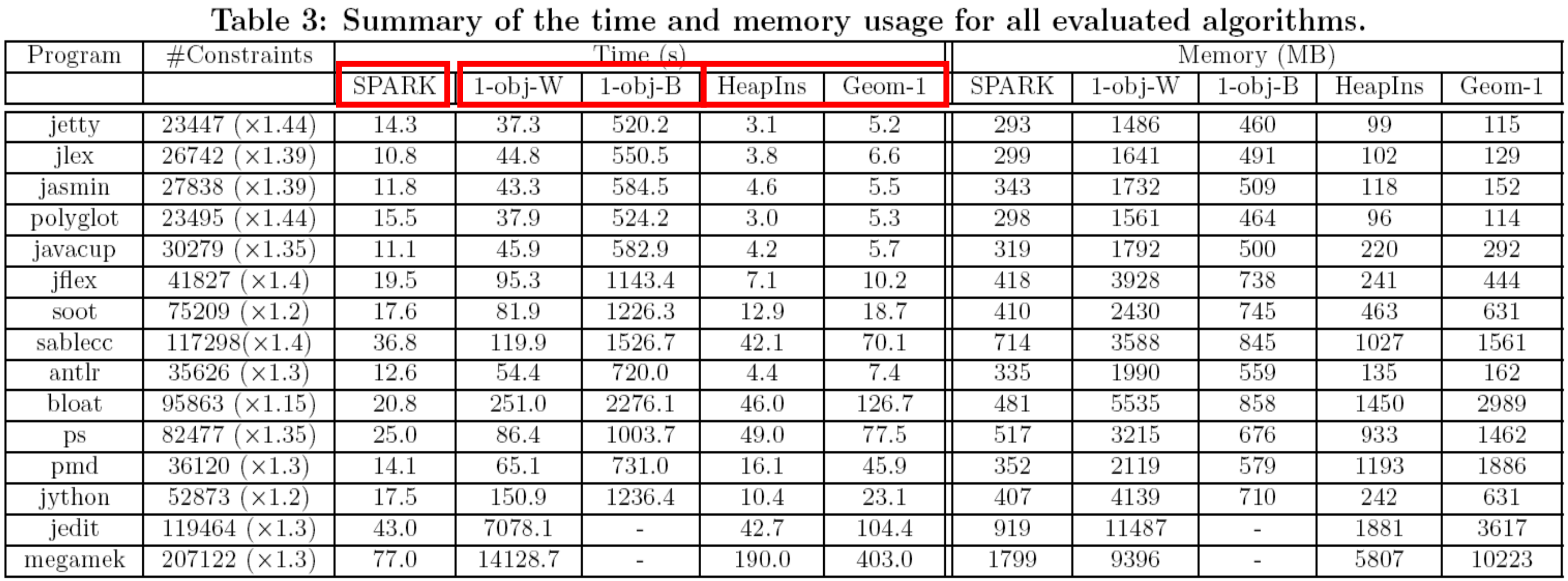


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

HeapIns and Geom-1 are:

- 23.9x and 11.6x faster than 1-obj-W;
- 111x and 68.3x faster than 1-obj-B;

Performance Evaluation

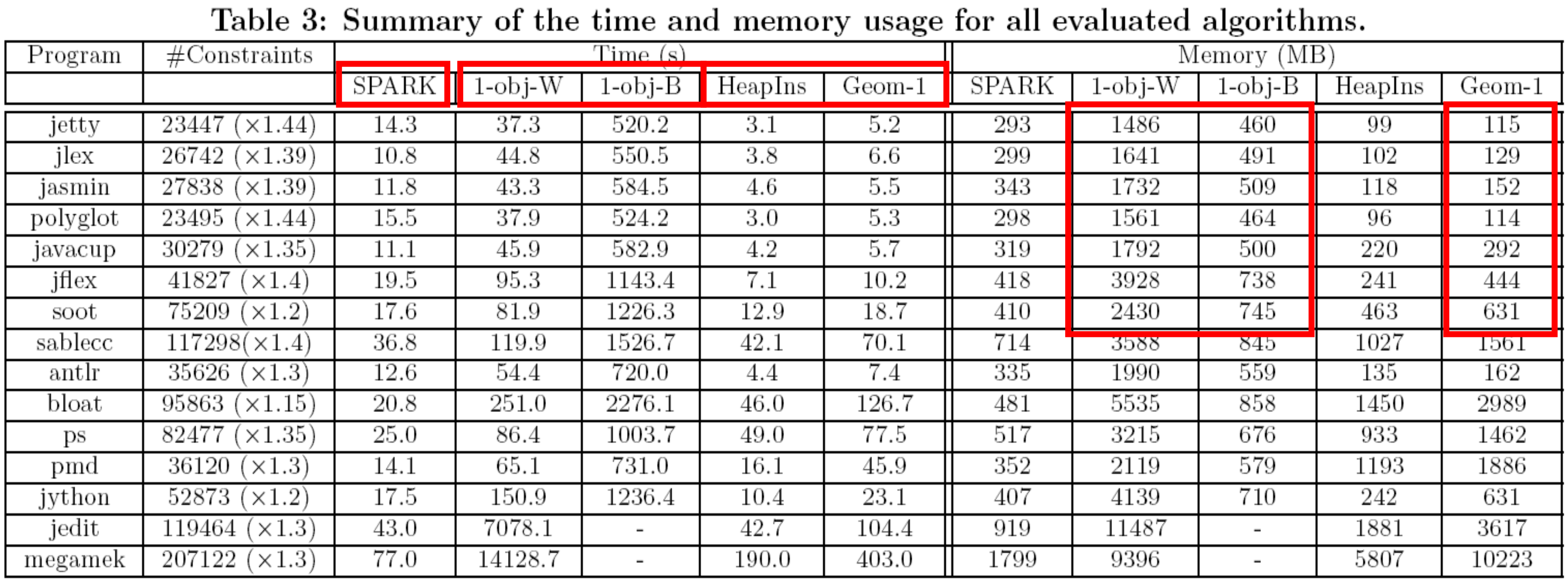


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

Performance Evaluation

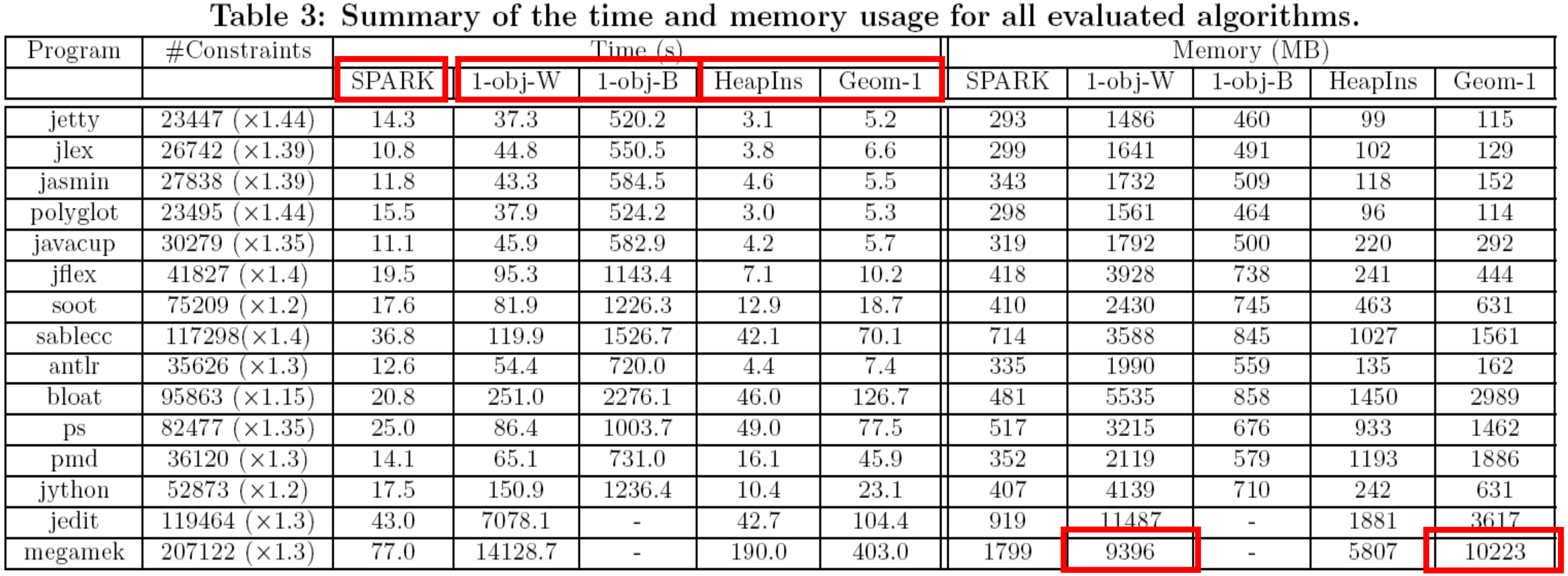


Table 3: Summary of the time and memory usage for all evaluated algorithms.

Program	#Constraints	Time (s)					Memory (MB)				
		SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1	SPARK	1-obj-W	1-obj-B	HeapIns	Geom-1
jetty	23447 (×1.44)	14.3	37.3	520.2	3.1	5.2	293	1486	460	99	115
jlex	26742 (×1.39)	10.8	44.8	550.5	3.8	6.6	299	1641	491	102	129
jasmin	27838 (×1.39)	11.8	43.3	584.5	4.6	5.5	343	1732	509	118	152
polyglot	23495 (×1.44)	15.5	37.9	524.2	3.0	5.3	298	1561	464	96	114
javacup	30279 (×1.35)	11.1	45.9	582.9	4.2	5.7	319	1792	500	220	292
jflex	41827 (×1.4)	19.5	95.3	1143.4	7.1	10.2	418	3928	738	241	444
soot	75209 (×1.2)	17.6	81.9	1226.3	12.9	18.7	410	2430	745	463	631
sablecc	117298(×1.4)	36.8	119.9	1526.7	42.1	70.1	714	3588	845	1027	1561
antlr	35626 (×1.3)	12.6	54.4	720.0	4.4	7.4	335	1990	559	135	162
bloat	95863 (×1.15)	20.8	251.0	2276.1	46.0	126.7	481	5535	858	1450	2989
ps	82477 (×1.35)	25.0	86.4	1003.7	49.0	77.5	517	3215	676	933	1462
pmd	36120 (×1.3)	14.1	65.1	731.0	16.1	45.9	352	2119	579	1193	1886
jython	52873 (×1.2)	17.5	150.9	1236.4	10.4	23.1	407	4139	710	242	631
jedit	119464 (×1.3)	43.0	7078.1	-	42.7	104.4	919	11487	-	1881	3617
megamek	207122 (×1.3)	77.0	14128.7	-	190.0	403.0	1799	9396	-	5807	10223

HeapIns and Geom-1 are:

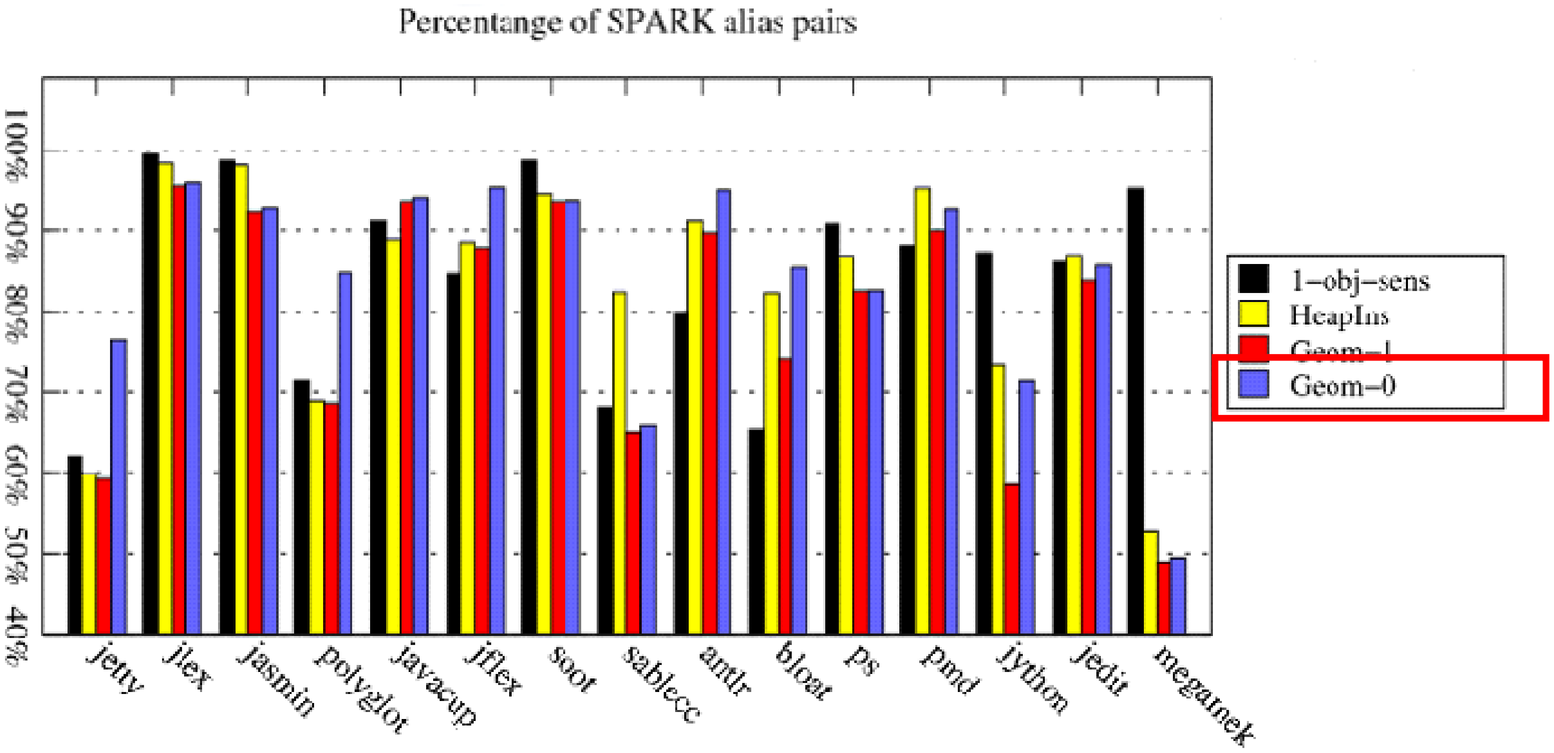
- requires 9.6x and 6.7x memory than 1-obj-W.

Alias Analysis

- Determine two pointers p and q may point to the same object or not;

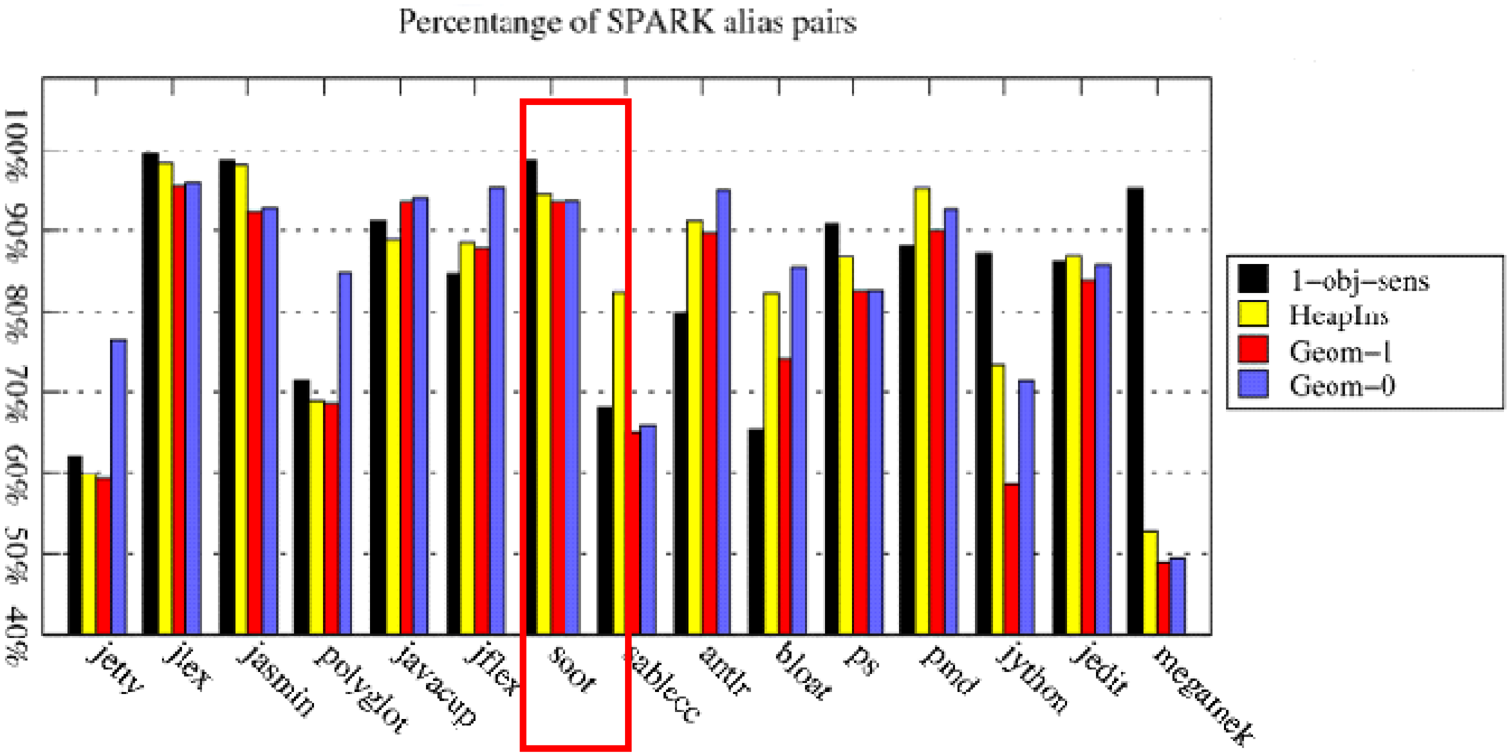
- Improving the precision of the alias analysis is not easy.

Alias Analysis



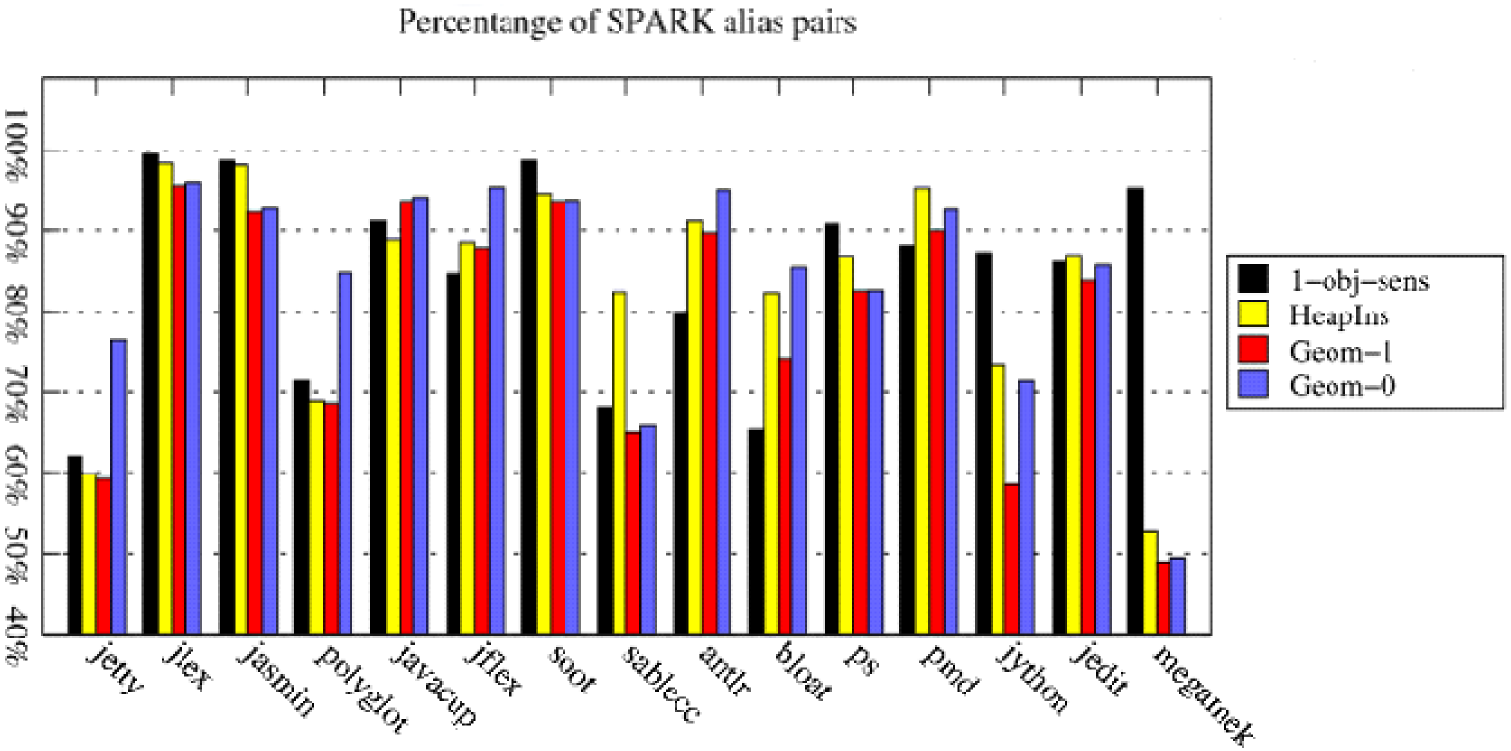
Geom-0 means the context insensitive modeling for the recursive calls.

Alias Analysis



Percentage of the alias pairs computed by SPARK.
The lower is better.

Alias Analysis



- 1-obj reduces 15.5%;
- HeapIns reduces 16.8%;
- Geom-1 reduces 21.2%.

Summary

- New data representation:
  - – We develop the geometric encoding, it has good compression rate for the working data of the points-to analysis.

Summary

- Soundness:
  - Geometric encoding has a sound calculating system for context sensitive points-to analysis.

Summary

- Easy to work with:
  - The complexity of writing a points-to analysis with geometric encoding is not too high.

Summary

- Good performance:
  - Geometric encoding based points-to analysis is 11.6x and 68.3x faster than 1-obj-W and 1-obj-B.

Summary

- Good precision:
  - – Geometric encoding based points-to analysis performs better than 1-obj-sens in alias analysis.

Other Results

- More experimental results and analyses can be found in our paper;

- We are still improving our methodology. Please visit us at

  www.cse.ust.hk/prism

  for up-to-date information.

Thanks for your attention!

香港科技大學
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Points-to Algorithm

Anderson's analysis

=

Worklist Selection

+

Points-to Propagation

+

Complex Constraints Instantiation

Points-to Algorithm

Anderson's analysis

=

Worklist Selection

+

Points-to Propagation

+

Complex Constraints Instantiation

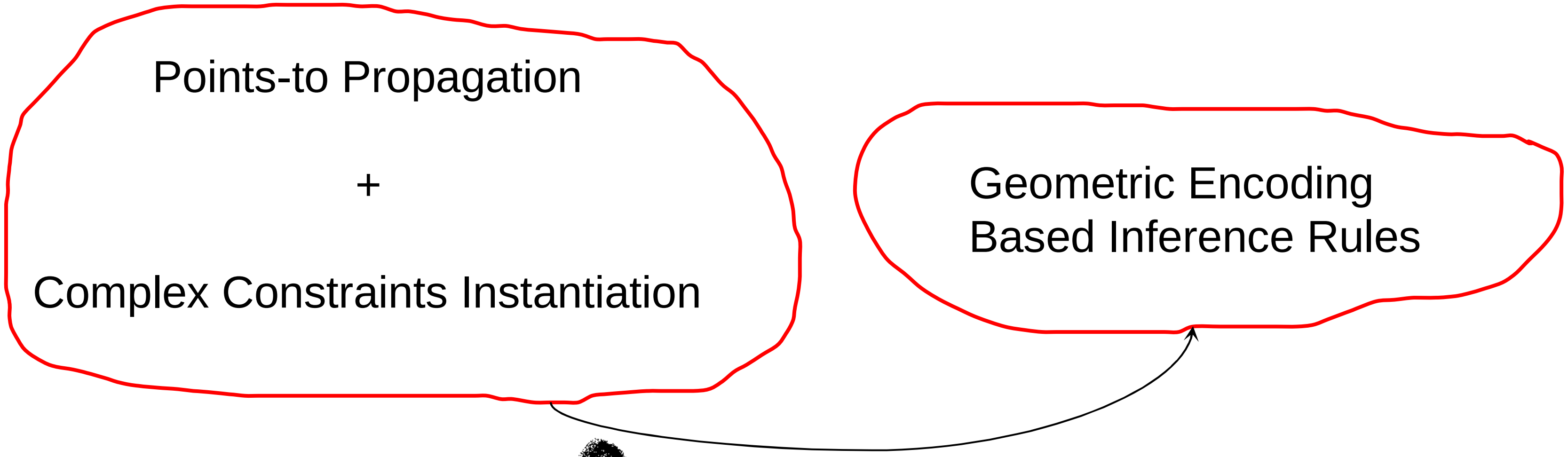
Points-to Algorithm

Anderson's analysis

=

Worklist Selection

+



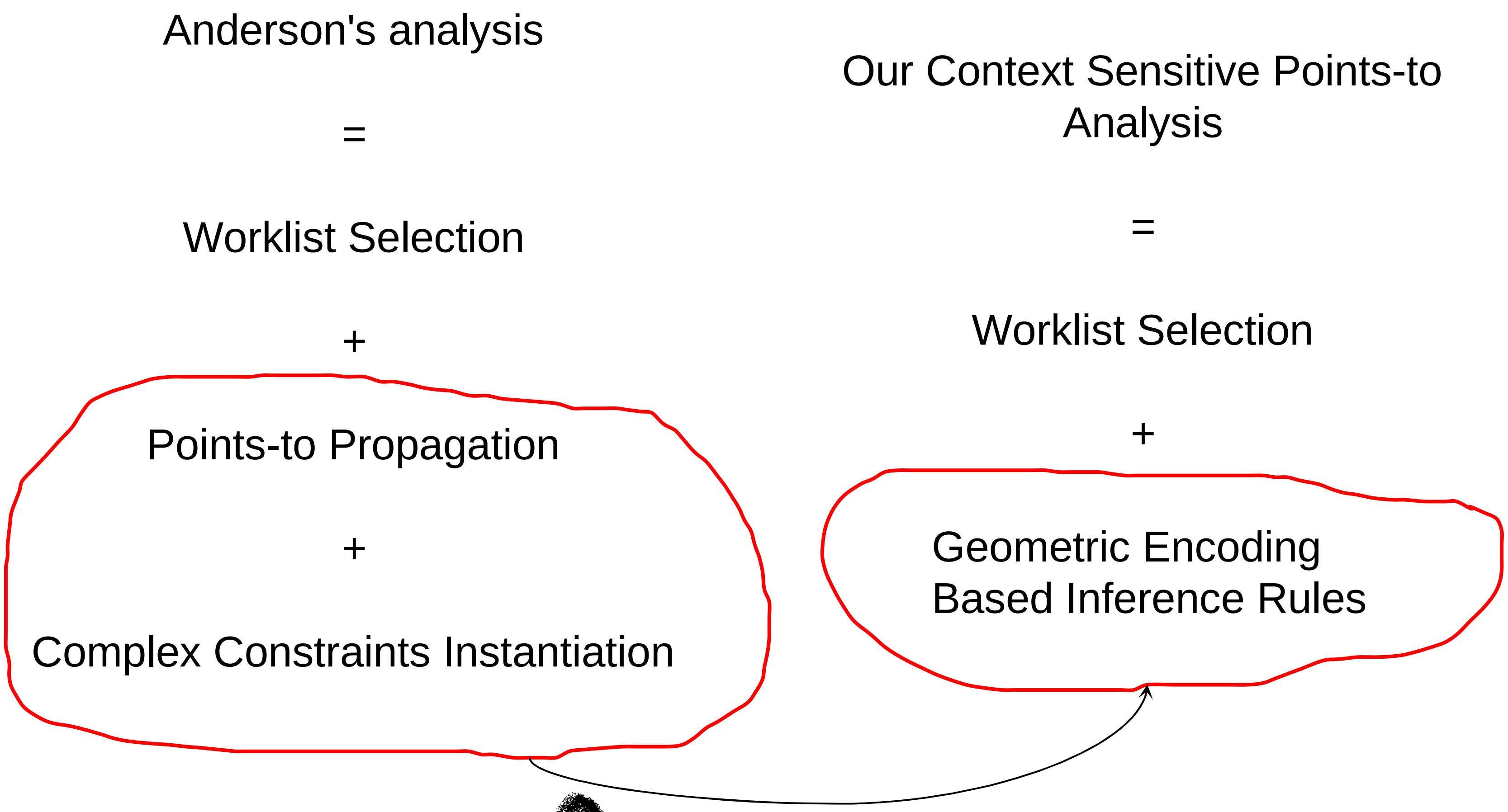
Points-to Propagation

+

Complex Constraints Instantiation

Geometric Encoding
Based Inference Rules

Points-to Algorithm



Anderson's analysis

=

Worklist Selection

+

Points-to Propagation

+

Complex Constraints Instantiation

Our Context Sensitive Points-to Analysis

=

Worklist Selection

+

Geometric Encoding Based Inference Rules

Other Characteristics

- With geometric encoding:
  - – Context sensitive model for the recursive calls;
  - – Simplified geometric encoding (HeapIns);
  - – Parameters to control the analysis time.

Other Characteristics

- Compared to EPA [ISSTA 08]:
  - We have the same compression capability with EPA;
  - We can handle global pointer correctly without trading-off precision.