

CS6843: Program Analysis

Week 5 : 06th Feb - 10th Feb 2017

Instructor : Dr. Rupesh Nasre

Scribes: Anju M.A(CS16D019), Joe Augustine(CS16S031)

Steensgaard's Analysis

5.1 Why Steensgaard's Analysis?

The points-to information gathered by Andersen's Analysis is precise enough for enabling many optimizations in the code generated by a compiler. But the drawback of Andersen's method is that it has a run time complexity of $O(n^3)$ even when the difference in information computed with respect to the previous iteration alone is propagated (n represents the number of variables in the given program which is being analysed). For many practical applications, this run time was too high that another method which is faster, though more imprecise, was sought after. Steensgaard came up with a method, which runs in almost linear time, $O(m \cdot \alpha(m))$, where m is the number of statements in the program and $\alpha(m)$ can be taken almost to be a constant. (It is the inverse ackermann function).

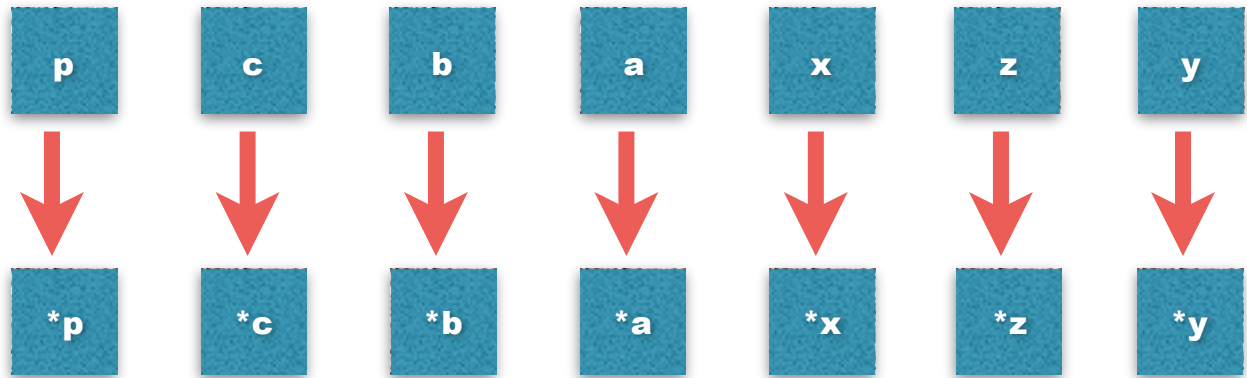
5.2 How does Steensgaard's analysis work?

It is a flow insensitive analysis. For a statement of the form $P=Q$, we merge the points-to-sets of P and Q and keep a single representative node with the merged information. Note that the analysis works on a normalized program which has only four types of pointer-manipulating statements namely, address-of, copy, load and store. The statement $P=Q$ can be one of these four types. Also, the analysis does exactly one iteration on the statements.

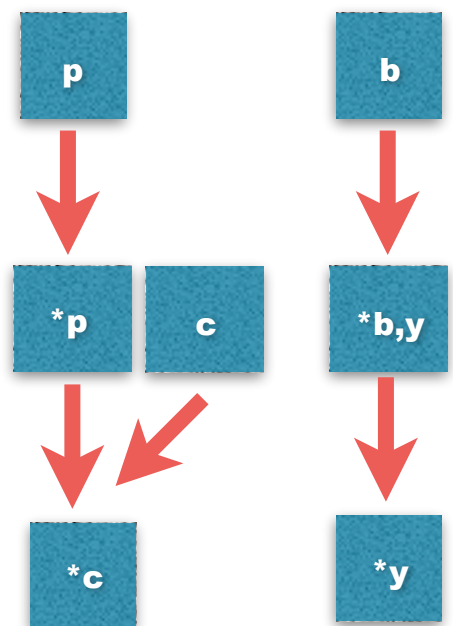
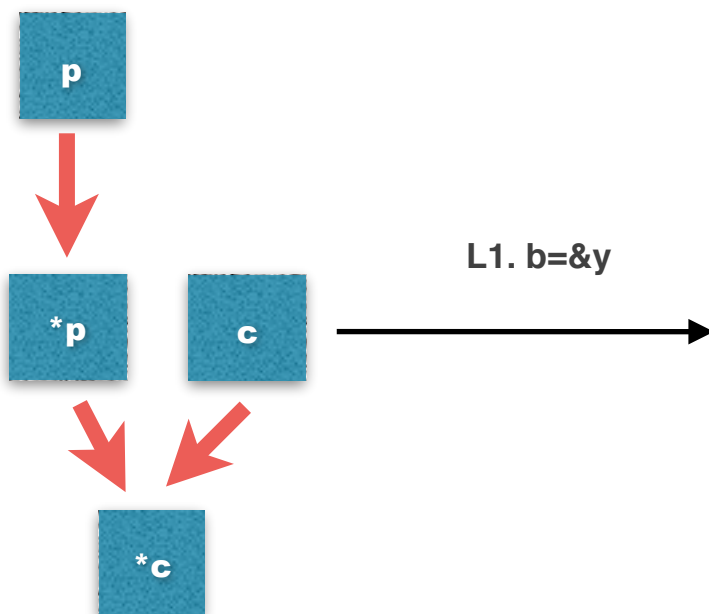
Note : There is a requirement that each pointer variable is initialized to a non-empty points-to set before starting Steensgaard's analysis. If the initialization is not done, an additional iteration may be needed to reach a fixed point. For example, if we have two statements $P=Q$; $Q=\&A$; and the points-to sets are initially empty, P will start pointing to A , only in the second iteration.

5.3 Example

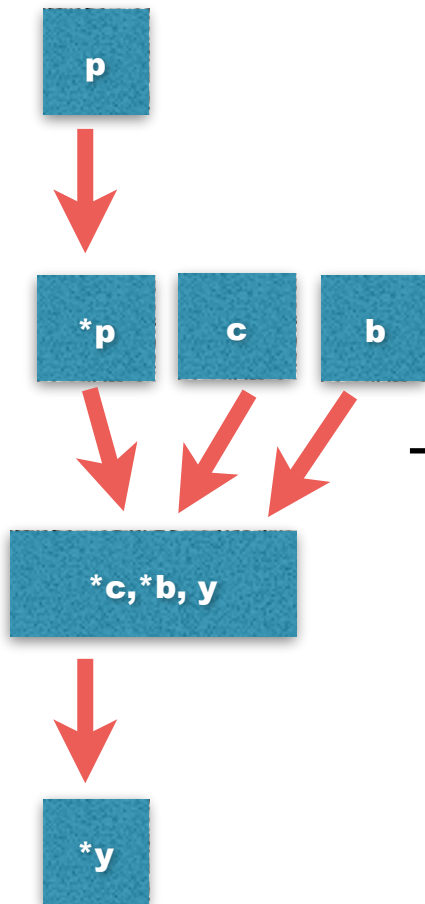
```
L0: *p = c;  
L1: b = &y;  
L2: b = *p;  
L3: p = &a;  
L4: a = &x;  
L5: *p = c;  
L6: c = p;  
L7: c = &z;
```



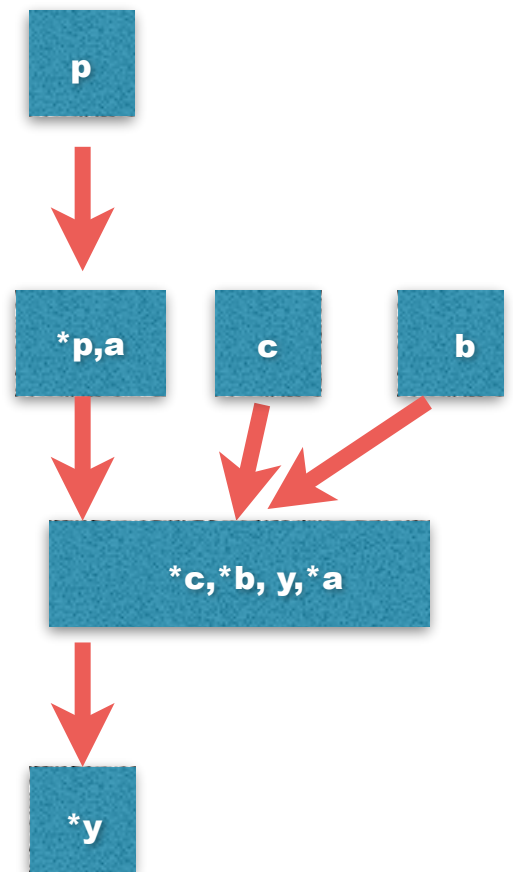
L0. ***p=c**



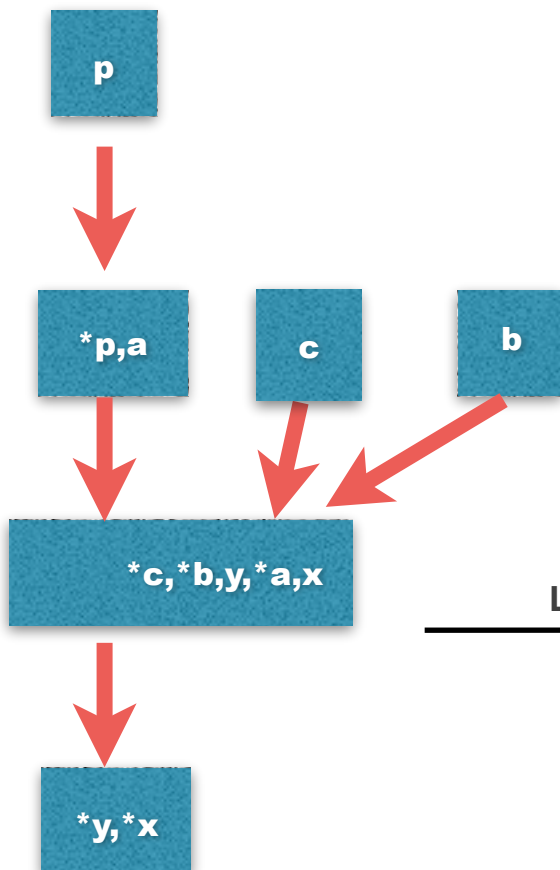
L2. $b = *p$



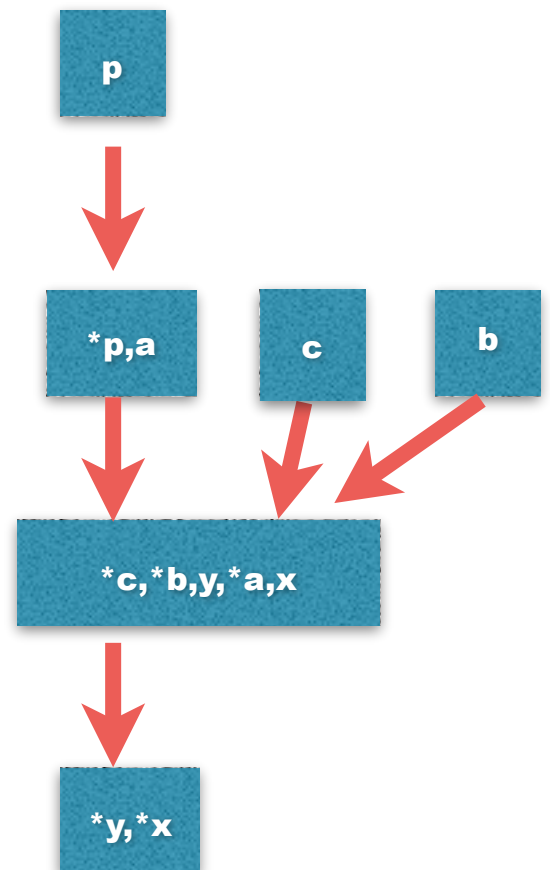
L3. $p = \&a$



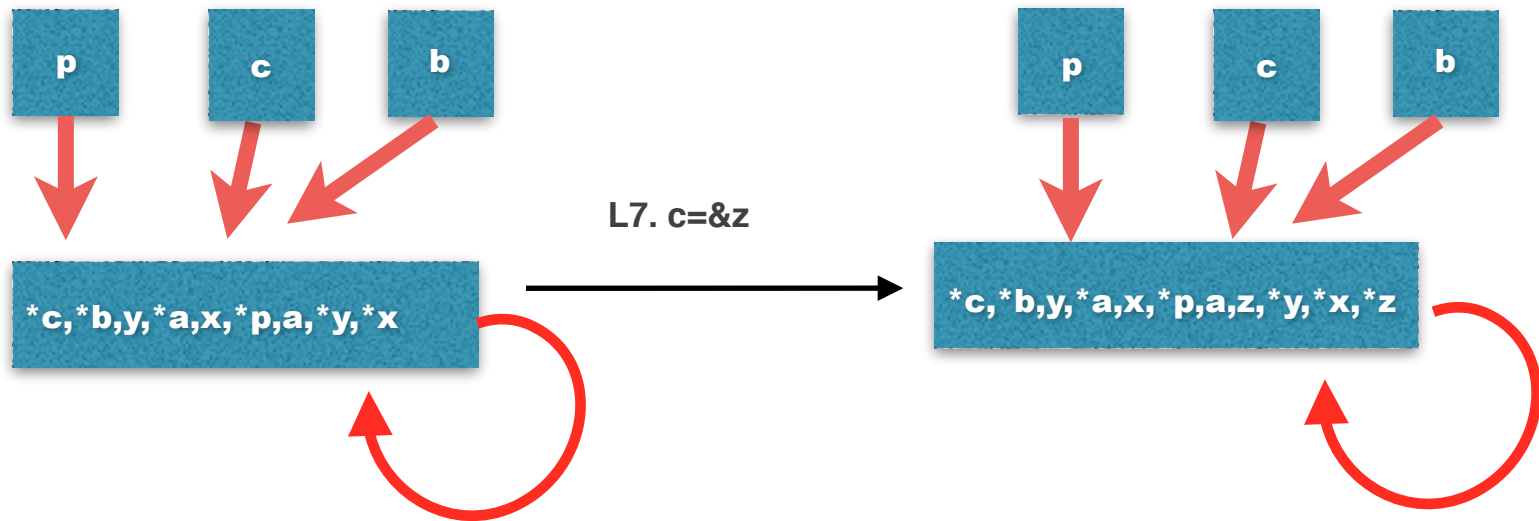
L4. $a = \&x$



L5. $*p = c$



L6. $c=p$



Question : Will we ever have to merge nodes of the form `**P`, `**C`, etc.,?

No. It will be needed only if we have statements of the form `*p=*c`. But in the normalized program, such statements will not be present. We can have either `p=*c` or `*p=c` but not `*p=*c` in the normalized program.

5.4 Comparison between Andersen's and Steensgaard's methods

	Andersen Analysis	Steensgaard's Analysis
Precision	High when compared to SA	Low when compared to AA
Runtime	$O(n^3)$ < n: number of variables in program>	Almost Linear time $O(m.\alpha(m))$
No. of Incoming edges to each nodes	$O(n)$, where n is number of pointer variables	$O(n)$, where n is number of pointer variables
No. of outgoing edges to each nodes	$O(n)$	at most one
Can cycle be present?	Yes	Yes - only in the form of self loops, that too in the leaf level nodes only

A few notes:

- What if we have a statement sequence like

```
Q=&P;  
P=Q;
```

Q will point to P. Points to sets of P and Q will be merged. Since the node after merging contains P, a self loop will be added. Finally we will remove P points to *P kind of information.

- If the analysis considers the system to be weakly typed, points to information has to be generated for all kinds of variables (including non-pointer variables).
- If the analysis considers the system to be strictly typed, points to information will be generated only for pointer variables which implies the calculated information will be more precise.
- For strictly typed languages, Steensgaard's analysis gives reasonably good information; in most cases, comparable to information generated by Andersen's Analysis.

- For strictly typed languages, levels in points-to graph indicate levels of dereferencing. This also indicates that pointers in different levels can never be aliases.
- Statements which don't generate any points-to constraints can be ignored if the analysis is path insensitive.
- `*dest=*src` will not change points to sets of `dest` and `src`. Such a statement can affect the points-to sets of the pointees of `dest` only.

5.4.1 Unifying Steensgaard's and Andersen's Methods:

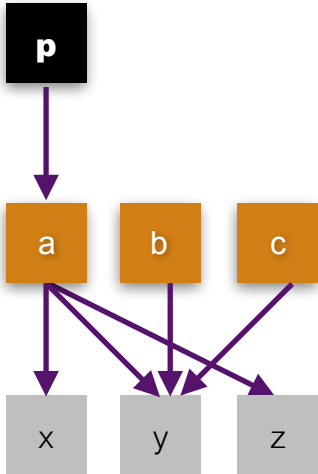
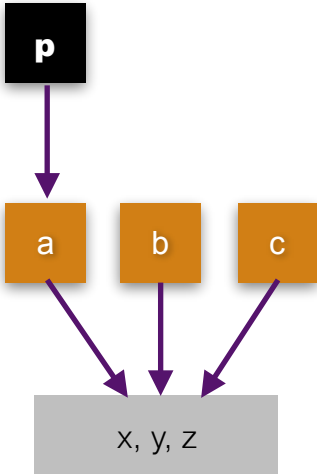
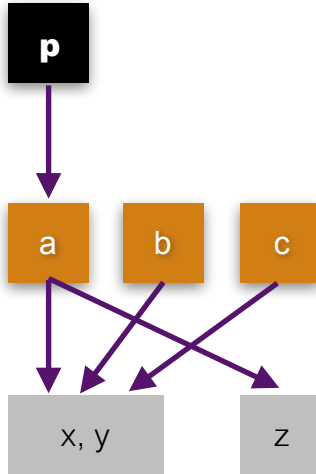
Both these methods can be unified and put into a single framework where the number of outgoing edges from a node can be a parameter. If the parameter is set to 1, it results in Steensgaard's solution, and if it can be any arbitrary number (bounded by n), then we get Andersen's solution.

- Is it possible to get a solution with better precision compared to Steensgaard's, but taking less time compared to Andersen's to compute the points-to information?
 - Yes! Steensgaard's method insists that every node in the points-to graph can have at most one outgoing edge. Andersen's method allows an arbitrary number of outgoing edges to each node in the points-to graph. If we modify the constraint on the number of outgoing edges to something in between (i.e., we allow more than one, but less than a fixed number of outgoing edges to each node), we may possibly get points-to information which is more precise than the information computed by Steensgaard's, in less time compared to the time taken by Andersen's. Some nodes which are merged in Steensgaard's analysis may not be considered for merging in the new method. Depending upon which nodes are considered for merging, the computed points-to information may be same as computed by Steensgaard's method, or Andersen's method or something in between.

Example:- Consider the set of statements below.

```
a=&x;
b=&y;
p=&a;
c=b;
*p=c;
a=&z;
```

The table below compares the points-to information computed by the unified model in three scenarios by varying the number of outgoing edges allowed. In the third scenario, instead of merging x and y, if the nodes x and z are merged, the information computed will be same as Andersen's solution.

Andersen's (Any number of outgoing edges per node)	Steensgaard's (At most one outgoing edge per node)	In between (At most two outgoing edges per node)
$a \rightarrow \{x, y, z\}$ $b \rightarrow \{y\}$ $c \rightarrow \{y\}$ $p \rightarrow \{a\}$	$a \rightarrow \{x, y, z\}$ $b \rightarrow \{x, y, z\}$ $c \rightarrow \{x, y, z\}$ $p \rightarrow \{a\}$	$a \rightarrow \{x, y, z\}$ $b \rightarrow \{x, y\}$ $c \rightarrow \{x, y\}$ $p \rightarrow \{a\}$
		

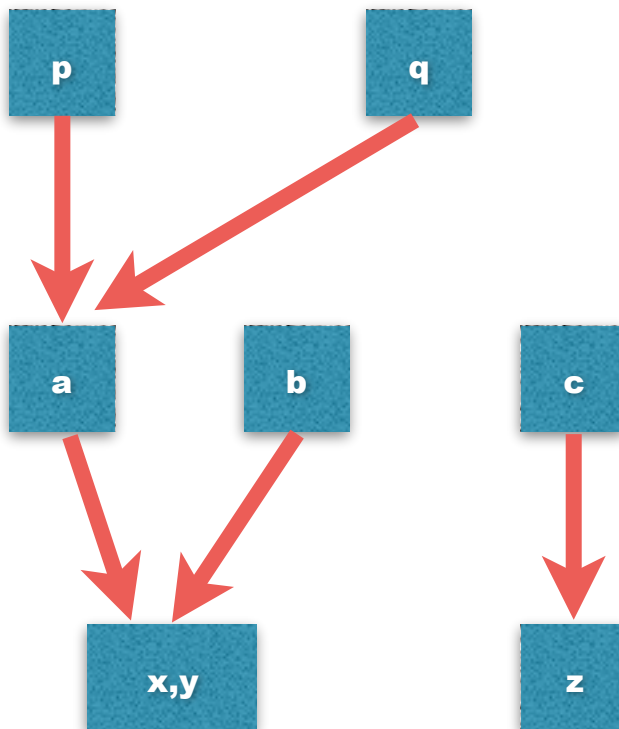
- Another way to unify Steensgaard's and Andersen's methods is based on the transitivity of aliasing relation among pointers.

Nature of aliasing relation between pointers as computed by Steensgaard's analysis:

- For a statement $P=Q$, the points-to sets of P and Q are merged to a single set (Let's indicate this new set by X). After $P=Q$, if there is another statement $Q = R$ in the program, the points-to set of R and the set X get merged. This

leads to P and R becoming aliases. i.e., aliasing relation among pointers is made transitive by the unification of points-to sets in Steensgaard's analysis.

- Since aliasing relation is also reflexive and symmetric, Steensgaard's method makes it an equivalence relation. i.e., pointers are partitioned into equivalence classes. Pointers having the same set of pointees fall into one class. The points-to information shown in the figure below form the equivalence classes $\{p,q\}$, $\{a,b\}$, $\{c\}$, $\{x,y\}$ and $\{z\}$.



5.4.2 Precision level of Andersen's analysis:

Even though Andersen's analysis computes more precise information compared to Steensgaard's analysis, it is not the most precise flow insensitive information that can be computed. This can be demonstrated by showing that a particular points-to fact which is computed by Andersen's analysis can not hold with any ordering of the statements in the given program. The most precise flow insensitive analysis should generate only those points-to facts which can hold with respect to some ordering of the given statements at runtime.

Realizable Facts:

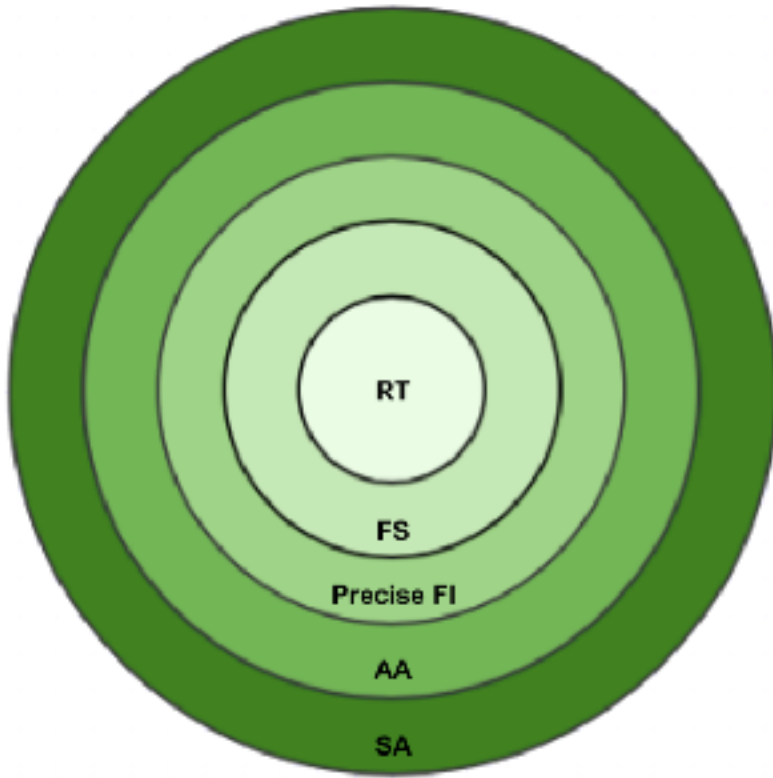
A points-to fact is realizable if there exists a sequence of statements which if executed in that order will satisfy the points-to fact. Such a sequence of statements is called a realizability sequence of the given points-to fact. If we allow the sequence to have any ordering of the statements in the given program, we are essentially doing a flow insensitive computation.

Lets consider an example. Figure shown below is a set of statements and the points-to information computed by Andersen's analysis.

Statements	Andersen's points-to
a = &c b = &a c = &b b = a *b = c d = *a	a -> {b, c} b -> {a, b, c} c -> {b} d -> {a, b, c}

- The realizability sequence for $b \rightarrow c$ is : $a = \&c, b = a$.
- The realizability sequence for $a \rightarrow b$ is : $c = \&b, b = \&a, *b = c$.
- The realizability sequence for $d \rightarrow a$ is : $b = \&a, c = \&b, *b = c, d = *a$.
- We can see that no realizability sequence exists for $d \rightarrow c$. This implies that the most precise flow insensitive analysis will not compute the points-to fact $d \rightarrow c$. But this points-to fact is computed by Andersen's analysis.

Note : Points-to is a liveness property. i.e., if a points to b in at least one execution path, a sound analysis will definitely compute the information $a \rightarrow b$. If the most precise flow insensitive analysis (which simulates all possible execution paths) does not compute the information $a \rightarrow b$, then we can be sure that a can not point to b at run time. If another analysis computes this information, we say this analysis computes a superset of the information computed by the most precise flow insensitive analysis. Hence we conclude that Andersen's analysis is not equivalent to the most precise flow insensitive analysis. The diagram below depicts the relationship among the amount of points-to information computed by various analyses.



RT : Run Time
FS : Flow Sensitive
FI : Flow Insensitive
AA : Andersen's Analysis
SA : Steensgaard's Analysis