

**Compiler Design**  
**Prof. Y. N. Srikant**  
**Department of Computer Science and Automation**  
**Indian Institute of Science, Bangalore**

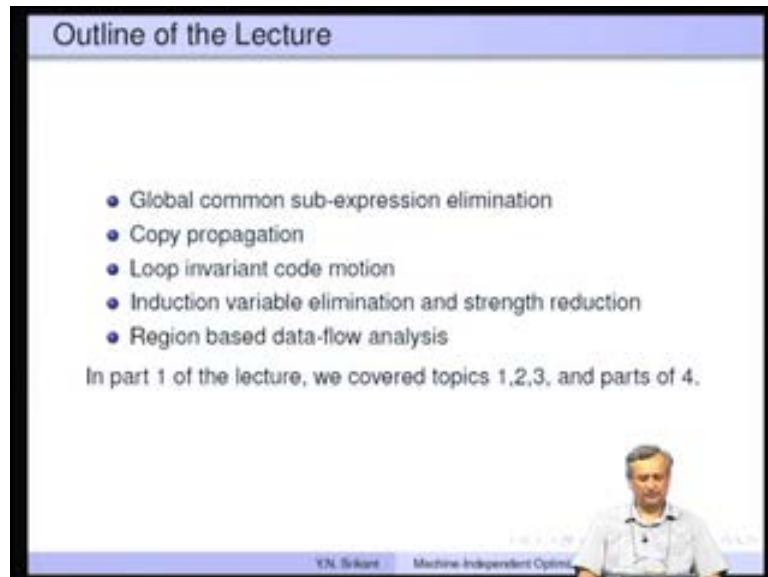
**Module No. # 10**

**Lecture No. # 26**

**Machine-Independent Optimizations - Part 3 and Data-flow Analysis: Theoretical Foundation**

Welcome to the lecture part 3 of Machine Independent Optimizations.

(Refer Slide Time: 00:24)



**Outline of the Lecture**

- Global common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Region based data-flow analysis

In part 1 of the lecture, we covered topics 1,2,3, and parts of 4.

Y.N. Srikant Machine-Independent Optimiz.

In the last lecture, we discussed induction variable elimination, strength reduction, etcetera. We also saw some parts of region based data-flow analysis. Today, we will continue with region based data-flow analysis and then move on to the theory of data-flow analysis.

(Refer Slide Time: 00:46)

**Region Based Data-flow Analysis**

- **Region:** A set of nodes  $N$  that includes a header, which dominates all other nodes in the region
- All edges between nodes in  $N$  are in the region, except (possibly) for some of those that enter the header
- All intervals are regions but there are regions that are not intervals
  - A region may omit some nodes that an interval would include or they may omit some edges back to the header
  - For example,  $I(7) = \{7, 8, 9, 10, 11\}$ , but  $\{8, 9, 10\}$  could be a region (see next slide)
- A region may have multiple exits
- We shall compute  $gen_{R,B}$  and  $kill_{R,B}$  of definitions generated and killed (resp.), along paths within the region  $R$ , from the header to the end of the block  $B$

YN Srikant Machine-Independent Optimizations

Again, a very quick revision; we discussed regions, which are groups of nodes. All edges between the nodes in  $N$  are in the region, except for some of the back edges, which enter the header, and the region may have multiple exits. So, the basic idea in region based analysis is to compute  $gen$  and  $kill$  for basic blocks in a certain region,  $gen_{R,B}$  and  $kill_{R,B}$ . In this, we consider the definitions generated and killed along the paths within the region  $R$ , from the header of the region to the basic block  $B$ .

(Refer Slide Time: 01:39)

**Region Based Data-flow Analysis (2)**

- These will be used to define a transfer function  $trans_{R,B}(S)$ , that tells for any set  $S$  of definitions, what subset of definitions reach the end of  $B$  by travelling along paths wholly within  $R$ , assuming that all and only the definitions in  $S$  reach the header of  $R$
- $trans_{R,B}(S) = gen_{R,B} \cup (S - kill_{R,B})$
- $trans_{U,B}(\cdot) = OUT[B] = gen_{U,B}$ , where  $U$  is the region consisting of the entire flow graph
- We need to provide a method to compute the transfer functions  $trans_{R,B}$ , for progressively larger regions defined by some  $(T_1 - T_2)$  transformation of a CFG
- Since  $OUT[B] = gen_{U,B}$ , we need to compute  $gen_{R,B}$  and  $kill_{R,B}$ , for each basic block, for progressively larger regions
- Interestingly, this approach does not compute

YN Srikant Machine-Independent Optimizations

**Central** to region based analysis is the computation of a transfer function called  $\text{trans } R, BS$ . This tells us how a set  $S$  of definitions or anything else are modified when we pass through various regions.  $\text{trans } R, BS$  is defined as  $\text{gen } R \text{ comma } B \text{ union } S \text{ minus kill } R \text{ comma } B$ . If the input to this trans function is  $\phi$ , then we get  $\text{gen } U \text{ comma } B$ , which is nothing, but  $\text{OUT } B$ . So, we basically want to compute  $\text{gen } U \text{ comma } B$ , where  $U$  is the region containing the entire flow graph.

(Refer Slide Time: 02:28)

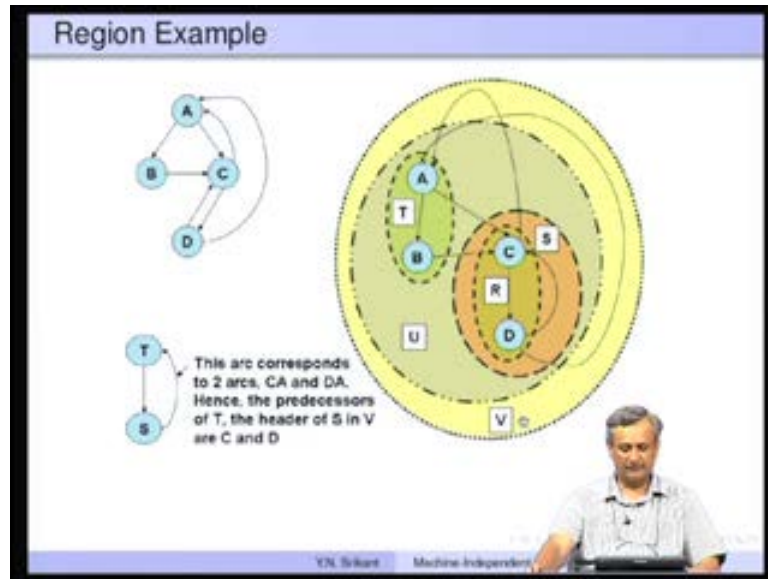
**Region Based Data-flow Analysis (3)**

- As we reduce a flow graph  $G$  by  $T_1$  and  $T_2$  transformations, at all times, the following conditions are true
  - 1 A node represents a region of  $G$
  - 2 An edge from  $a$  to  $b$  in a reduced graph represents a set of edges
  - 3 Each node and edge of  $G$  is represented by exactly one node or edge of the current graph
- Region based DFA can be compared to *syntax-directed translation*, with the structure being provided by the hierarchy of regions
- We consider data-flow analysis for *reaching definitions*
- It should be emphasized that all data-flow values which reach the header of a region will surely flow to all constituent regions and basic blocks, since all basic blocks are reachable from the header of the enclosing region.

YN Srikant Machine Independent

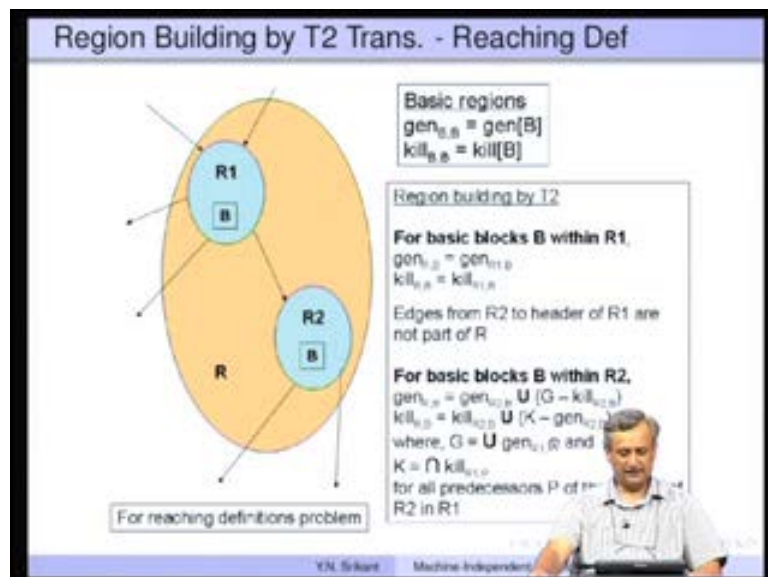
Now, how do we get regions? We applied  $T_1$   $T_2$  analysis, the transformations and as we go on, every time we apply some transformation, we get a bigger region. So, there is a hierarchy of regions, which is built once we start applying the transformations to the time when we end applying the transformations.

(Refer Slide Time: 02:56)



Here is a very simple example. This is a flow graph. We apply T 1 transformation on C and D, again T 2 transformation on C and D, T 2 transformation on A and B, and T 1 transformation on the region R containing C and D. This self-loop is removed and then we apply T 2 transformation between these two regions – T and S. Finally, we apply the T 1 transformation to remove this self-loop and we get the region V.

(Refer Slide Time: 03:33)



When you look at the reaching definitions and see how region building happens by T 2 transformation and how  $gen_{R, B}$  gets computed, we have two situations: one is

for T 2 and the other is for T 1. So, in the T 2 transformation, there is a region R1, another region R2, and we make a bigger region R. So, we are given the information about R1 and R2 and we want to compute the information about R.

For basic blocks B within the region R1, there is no modification because it is not affected by anything in R2 and there are no arcs going back from R2 to R1 within the region R. So, gen of R comma B and kill of R comma B remain the same as gen of R1 comma B and kill of R1 comma B. However, for R2, there are inputs coming from R1 and where in R1 do they come from? They are actually the... If we take this particular arc (Refer Slide Time: 04:33), it is the number of arcs connecting to various basic blocks in R1 and those are the basic blocks from which input can arrive into R2. So, for this region, we define G as union of gen of R1 comma P, where all the predecessors P of the header of R2 in R1 are considered. Similarly, K is intersection of kill of R1 comma P with the same definition.

Now, gen of R comma B for basic blocks in R2 are gen of R2 comma B union G minus kill of R2 comma B. So, taking it into account all those information, which are coming from R1.

(Refer Slide Time: 05:21)

The slide is titled "Region Building by T1 Trans. - Reaching Def". It features a diagram on the left showing a large yellow oval labeled 'R' containing a smaller blue oval labeled 'R1'. Inside 'R1' is a small square labeled 'B'. Arrows indicate flow from outside 'R' into 'R1' and from 'R1' into 'R'. A text box on the right contains the following text:

Region building by T1

$$\text{gen}_{R,B} = \text{gen}_{R1,B} \cup (G - \text{kill}_{R1,B})$$

$$\text{kill}_{R,B} = \text{kill}_{R1,B}$$

where,  $G = \bigcup \text{gen}_{R1,P}$  for all predecessors P of the header of R1 in R

It is not necessary to compute  $\text{kill}_{R,B}$  as in the previous case (T2)

A definition gets killed going from the header to B if it is killed all acyclic paths, and hence edges incorporated into R will cause more definitions to be...

At the bottom of the slide, there is a small video inset of a man speaking, and the text "For reaching definitions problem" and "Y.N. Srikant Machine Independent".

Then, we presented the T1 transformation. This is very similar. Here, the kill does not get modified at all when we go from region R1 to R simply because we would have considered all the acyclic paths in R1 to the header of B in computing kill. So, the kill



information remains the same. kill of R comma B is kill of R1 comma B, but the gen information is computed in a way similar to that of T2 transformation.

We consider the G, which is the union of the gen of R1 comma P. So, P is the set of all predecessors of R1 in R. So, you take the header of R1 and then all the predecessors that gives you the set of predecessors.

(Refer Slide Time: 06:10)

**Region Based RD Analysis - An Example (1)**

Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A-W 1986

Y.N. Srikant Machine Independent

This is what we did for reaching definitions. Then, we saw how exactly it can be computed for this example.

(Refer Slide Time: 06:17)

**Region Based RD Analysis - An Example (2)**

Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A-W 1986


- Building region R from regions C and D by T2 transf.
- $gen_{R,C} = gen_{C,C} = 000$ ;  $kill_{R,C} = kill_{C,C} = 010$
- Header of R is D and pred. of D in C is C
- $G = gen_{C,C} = 000$  and  $K = kill_{C,C} = 010$
- $gen_{R,D} = gen_{D,D} \cup (G - kill_{D,D}) = 001 + (000 - 0) = 001$   
 $kill_{R,D} = kill_{D,D} \cup (K - gen_{D,D}) = 000 + (010 - 0) = 000$

Y.N. Srikant Machine Independent

For each step, when we built a region R, we said how to compute the gen and R sets.

(Refer Slide Time: 06:25)

### Region Based RD Analysis - An Example (3)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A. W. 1989


- Building region S from region R by T1 transformation
- The only predecessor of the header C, within S is D
- Therefore,  $G = gen_{R,D} = 001$
- $kill_{S,C} = kill_{R,C} = 010$ ;  $kill_{S,D} = kill_{R,D} = 010$
- $gen_{S,C} = gen_{R,C} \cup (G - kill_{R,C}) = 000 + (001 - 0) = 001$
- $gen_{S,D} = gen_{R,D} \cup (G - kill_{R,D}) = 001 + (001 - 0) = 001$

YN. Srikant    Machine-Independent

Then, we applied T1 transformation to build S. Again, we said how to compute gen and kill.

(Refer Slide Time: 06:31)

### Region Based RD Analysis - An Example (4)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", A. W. 1989

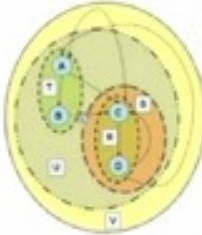
- Building region T from regions A and B by T2 transf.
- $gen_{T,A} = gen_{A,A} = 100$ ;  $kill_{T,A} = kill_{A,A} = 010$
- Header of B is B and pred. of B in A is A
- $G = gen_{A,A} = 100$  and  $K = kill_{A,A} = 010$
- $gen_{T,B} = gen_{B,B} \cup (G - kill_{B,B}) = 010 + (100 - 100) = 010$
- $kill_{T,B} = kill_{B,B} \cup (K - gen_{B,B}) = 101 + (010 - 010) = 101$

YN. Srikant    Machine-Independent

Then, the same for T.

(Refer Slide Time: 06:33)

### Region Based RD Analysis - An Example (5)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", July 1986

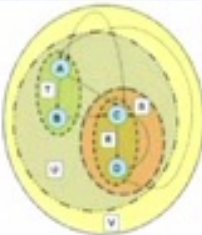
- Building region U from regions T and S by T2 transf.
- $gen_{U,A} = gen_{T,A} = 100$ ;  $kill_{U,A} = kill_{T,A} = 010$
- $gen_{U,B} = gen_{T,B} = 010$ ;  $kill_{U,B} = kill_{T,B} = 101$

Y.N. Srikant    Machine-Independent

Then, for U.

(Refer Slide Time: 06:35)

### Region Based RD Analysis - An Example (6)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", July 1986

- Building region U from regions T and S by T2 transf.
- Header of S is C and pred. of C in T are A and B
- $G = gen_{T,A} \cup gen_{T,B} = 110$  and  
 $K = kill_{T,A} \cap kill_{T,B} = 000$
- $gen_{U,C} = gen_{S,C} \cup (G - kill_{S,C}) = 001 + (110 - 010) = 101$   
 $kill_{U,C} = kill_{S,C} \cup (K - gen_{S,C}) = 010 + (000 - 000) = 010$   
 $gen_{U,D} = gen_{S,D} \cup (G - kill_{S,D}) = 001 + (110 - 010) = 101$   
 $kill_{U,D} = kill_{S,D} \cup (K - gen_{S,D}) = 010 + (000 - 001) = 010$

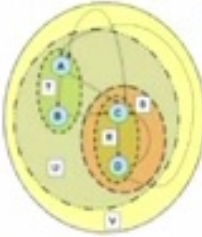
Y.N. Srikant    Machine-Independent

Then, for building a region U.



(Refer Slide Time: 06:39)

### Region Based RD Analysis - An Example (7)



Block	gen	kill
A	100	010
B	010	101
C	000	010
D	001	000

Adapted from "The Dragon Book", Aug 1986

- Building region V from region U by T1 transf.
- Header of U is A and pred. of A in U are C and D
- $G = gen_{U,C} \cup gen_{U,D} = 101$
- $gen_{V,C} = gen_{U,C} \cup (G - kill_{U,C}) = 101 + (101 - 010) = 101$   
 $gen_{V,D} = gen_{U,D} \cup (G - kill_{U,D}) = 101 + (101 - 000) = 101$   
 $kill_{V,C} = kill_{U,C} = 010; kill_{V,D} = kill_{U,D} = 010$

Y.N. Srikant    Machine-Independent

Finally, for the region V.

(Refer Slide Time: 06:44)


### Results from Iterative RD DFA for the same example

	gen	kill	OUT <sub>1</sub>	IN <sub>1</sub>	OUT <sub>2</sub>	IN <sub>2</sub>	OUT <sub>3</sub>	IN <sub>3</sub>	OUT <sub>4</sub>	IN <sub>4</sub>	RA
A	100	010	100	001	101	101	101	101	101	101	101
B	010	101	010	100	011	101	010	101	010	101	010
C	000	010	000	111	101	111	101	111	101	111	101
D	001	000	001	000	001	101	101	101	101	101	101

$$OUT[B] = gen[B] \cup (IN[B] - kill[B])$$

$$IN[B] = \bigcup_{P, \text{ a predecessor of } B} OUT[P]$$

$$IN[B] = \emptyset \text{ (initialization)}$$



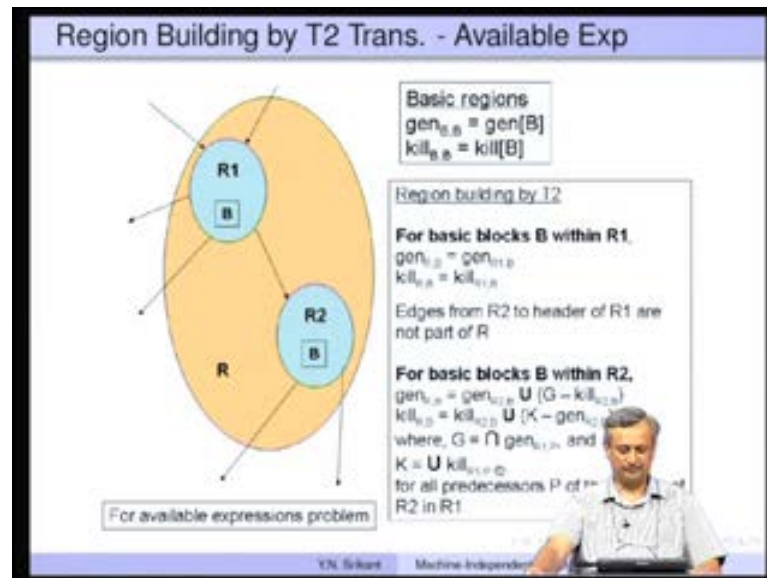
Reaching Definitions Problem

Y.N. Srikant    Machine-Independent

We still have not seen what happens to this analysis when we compare it to the iterative data-flow analysis. These are our well known equations of iterative data-flow analysis for reaching definitions –  $OUT\ B$  equal to  $gen\ B$  union  $IN\ B$  minus  $kill\ B$ ,  $IN\ B$  equal to union of  $OUT\ P$ , where  $P$  is a predecessor of  $B$ , and  $IN\ B$  equal to  $\phi$  as the matter of initialization.

If we go through the OUT and IN computation in about four iterations, the set converges and does not change thereafter. Here are the OUT values, which we get finally. This is what we get (Refer Slide Time: 07:28) by region analysis and you can see that they are identical. So, whether we follow the region based analysis, or we follow the iterative data-flow analysis, the sets will be very similar; not much difference between these two.

(Refer Slide Time: 07:48)



Now, what happens if we want to tackle the available expressions problem? If we recall the available expressions problem and the iterative solution that we gave, we had this as a forward data-flow analysis problem and we used the confluence operator as intersection. Here, something similar happens. The picture seems to be exactly the same as the previous case; yes, it is indeed so. It is just that for basic blocks within R1, there is absolutely no change as before, but for R2, there are some changes.

Here, the gen and kill equations are identical:  $\text{gen of } R2 \text{ comma } B \text{ union } G \text{ minus kill of } R2 \text{ comma } B$  and  $\text{kill of } R \text{ comma } B$  is  $\text{kill of } R2 \text{ comma } B \text{ union } K \text{ minus gen of } R2 \text{ comma } B$ . In other words, whatever is generated within R2, that is taken, then whatever is coming from R1 is taken, and then whatever is killed by R2 is removed from that. So, that is exactly what we used to do before. So, that is what we do here as well. However, the computation of G is slightly different. Similarly, K is also different. So, **available expressions**, we wanted the expression to reach on all paths. Justifiably, this G (Refer

Slide Time: 09:20) is intersection gen of R1 comma P. For the reaching definitions, any one path was **ok**, but here we need the expression along all paths. So, it is an intersection.

K is a union instead of the intersection because if the expression is killed along any path, then it is completely killed. So, the G computation involves an intersection and K computation involves union.

(Refer Slide Time: 09:52)

**Region Building by T1 Trans. - Available Exp**

Region building by T1:

$$\text{gen}_{R_1, B} = \text{gen}_{R_1, B}$$

$$\text{kill}_{R_1, B} = \text{kill}_{R_1, B} \cup (K - \text{gen}_{R_1, B})$$

where,  $K = \bigcup \text{kill}_{P, B}$ , for all predecessors P of the header of R1 in R

It is not necessary to compute  $\text{gen}_{R_1, B}$  as in the previous case (T2).

An expression gets generated going from the header to B if it is generated along all acyclic paths, and hence back edges incorporated into R will not cause more expressions to be generated

For available expressions problem

YN Srikant Machine-Independent Optimizations

Similarly, in the case of T1 transformation, the gen part remains as gen of R1 comma B there is no change, whereas the kill changes. The reason is – for gen, we would have already considered all paths in the region R1. An expression gets generated going from the header to B if and only if it is generated along all acyclic paths, and hence back edges incorporated into R will not cause any more expressions to be generated. So, we need not consider this back edge when we compute the gen sets.

gen of R comma B is same as gen of R1 comma B, but in the previous reaching definitions example, this was not so. Whereas, kill of R comma B was the same as kill of R1 comma B and here, it is different – kill of R1 comma B union K minus gen of R1 comma B. So, looks like the gen and kill equations are kind of interchanged from the reaching definitions problem to the available expressions problem.

(Refer Slide Time: 11:00)

Results from Iterative AE DFA for the same example

	gen	kill	OUT <sub>1</sub>	IN <sub>1</sub>	OUT <sub>2</sub>	IN <sub>2</sub>	OUT <sub>3</sub>	IN <sub>3</sub>	OUT <sub>4</sub>	IN <sub>4</sub>	RA
A	100	010	100	101	101	000	100	000	100	000	100
B	010	101	010	100	010	101	010	100	010	100	010
C	000	010	101	000	101	000	000	000	000	000	000
D	001	000	111	101	101	101	101	000	001	000	001

$$OUT[B] = gen[B] \cup (IN[B] - kill[B])$$

$$IN[B] = \bigcap_{P, a \text{ predecessor of } B} OUT[P]$$

$$IN[B] = U \text{ (initialization)}$$

Available Expressions Problem

Like in the previous case, when we look at the available expressions problem in the iterative domain, we have the same kind of equations as before. The confluence operator here is intersection and in these, U, the universe of all expressions.

Again, it converges in about four iterations. The final values are here. In the final values, we obtained by region based analysis are here. We can see that they are identical. So, whether we follow region based analysis or iterative analysis, the answer probably will always be the same, except of course, when there is a little bit of node splitting and so on, which we will see very soon. Then, why do we want to apply region based analysis? In general, region based analysis is slightly faster than iterative analysis, but otherwise there is no huge advantage.

(Refer Slide Time: 12:06)

**Handling Irreducible Flow-Graphs**

- At some point of reduction in  $T_1 - T_2$  analysis, no further reduction is possible if the graph is irreducible
- At this point, we split nodes (regions are now nodes) and duplicate them as explained earlier
- We then continue our analysis
- If we wish to retain the original graph with no splitting, then after analyzing the split graph, we compute  $IN[B] = IN[B_1] \wedge IN[B_2] \wedge \dots \wedge IN[B_k]$ , where  $B_i, 1 \leq i \leq k$  are the siblings of the split node  $B$
- Splitting regions may be sometimes beneficial to optimizations since data-flow information may become more precise after splitting
  - For example, fewer definitions may reach each of duplicated blocks than that reach the original

YN Srikant Machine Independent

How do we handle irreducible flow-graphs? At some point of reduction in  $T_1 - T_2$  analysis, no further reduction is possible if the graph is irreducible. We will get stuck. At this point, we split nodes; we have seen examples of this already, regions are all nodes in this case and duplicate them as explained earlier. We saw how node splitting is done and then edges, which come into a particular node are all distributed to the various siblings that we get. We then continue our analysis as if nothing happened. Eventually, this splitting and continuation will take us to a limit flow-graph, which is a single node. Suppose we want to retain the original graph with no splitting; that is, we want to relate the gen sets and OUT sets to the original graph and not with the changed graph, we can simply compute IN of B in the original graph. B is in the original graph.

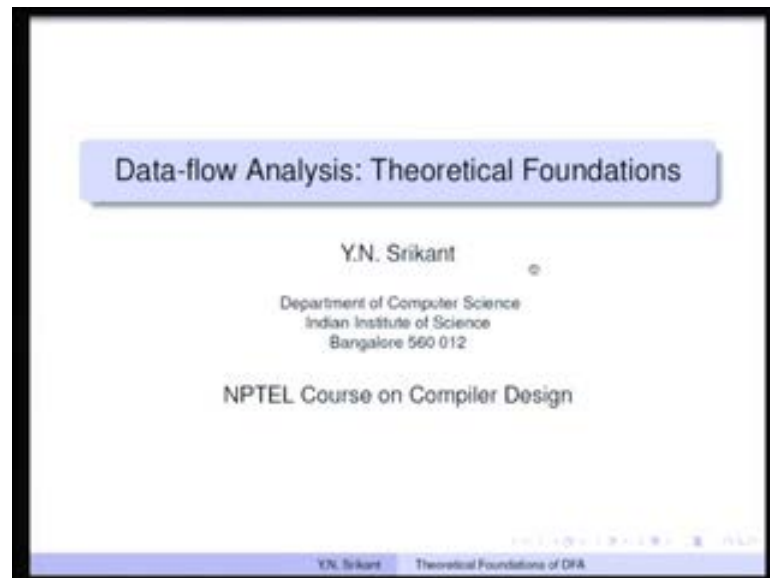
As  $IN$  of  $B_1$  meet  $IN$  of  $B_2$  meet  $IN$  of  $B_k$ , where  $B_1$  to  $B_k$  are all siblings of the split node  $B$ , this gives us a conservative estimate. In the reaching definitions problem, this will be a union in the available expression problem and this meet operator (Refer Slide Time: 13:34) will be an intersection.

Splitting regions sometimes may be beneficial to optimizations since data-flow information may become a little more precise after splitting. For example, fewer definitions may reach each of the duplicated blocks than that reach the original **node**. The reason is we do not have as many incoming edges to each node, which has been split. Exactly one incoming edge to each of these split nodes exist. Therefore, we can probably



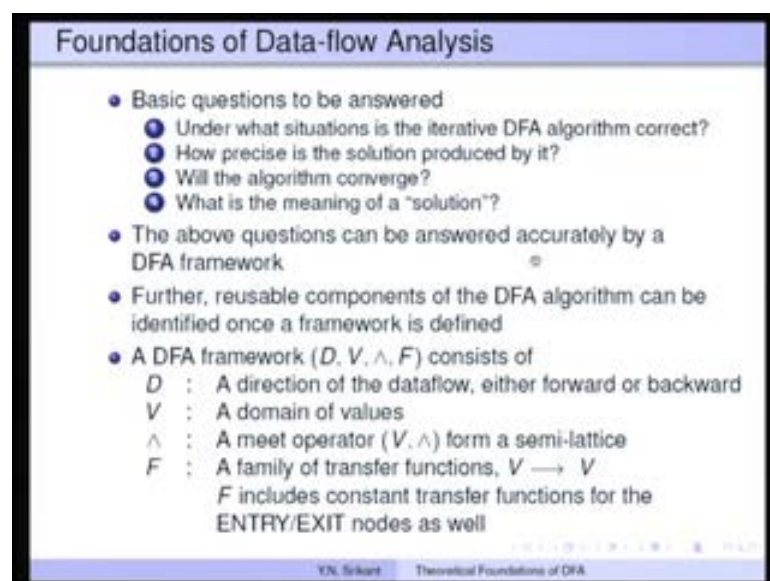
have a little more precise data-flow information when we split nodes, but then the penalty we pay is many nodes, explosion in the size of the control flow graph. Finally, the code generator will also be much more than in the previous case. That is the end of the lecture on region based analysis and optimizations. Now, we move on to the next lecture on the theory of data-flow analysis.

(Refer Slide Time: 14:41)



Welcome to the lecture on Theoretical Foundations of Data-flow Analysis.

(Refer Slide Time: 14:50)



Why do we actually explore the theoretical foundations? These are the questions that we are going to answer in this lecture. Under what situations is the iterative data-flow analysis correct? So far in the data-flow analysis, we just studied the equations, how to implement them in the form of a while loop, with a change variable, and so on and so forth, but we never thought about the question of – is the answer correct, is it always correct, or is it correct only a few times? Under what situations is the iterative data-flow analysis correct? This is the question we never answered. We will try to provide answer to this question.

How precise is the solution provided by the iterative algorithm? How close are we to the ideal solution? This is also something that we have never understood. Let us try understanding this issue as well. Will the algorithm converge? How do we know that we are going to stop? May be the iterations will go on and on. There is a theory, which ensures that the algorithms indeed converge and we are going to study that theory now.

What is the meaning of a solution? What do we mean by a solution, are there other solutions possible, etcetera need to be answered. So, we are going to define a data-flow analysis framework formally, a little more mathematically than we did before. Then, the properties of this framework will help us in understanding these questions and answers.

Further, by actually proposing a theoretical framework for data-flow analysis, reusable components of the data-flow analysis algorithm can be identified once we define such a framework. Which of these are the components? Etcetera; we will see very soon.

What is a data-flow analysis framework? It has a direction of dataflow,  $D$ . There is a domain of values,  $V$ . There is a meet operator; that is, the confluence operator that we have been talking about. There is a family of transfer functions. What are the formal aspects of these? The direction of course is just forward or backward. We are not studying any other type of direction here. The domain of values,  $V$  along with the meet operator, they actually form a semi-lattice. So, we are going to study what is a semi-lattice and how is it applicable to our problem. The family of transfer functions,  $V$  to  $V$ ; that is, the domain and range are  $V$  to  $V$ ;  **$V$  and range is also  $V$** . The family,  $F$  includes constant transfer functions for ENTRY and EXIT nodes. It includes identity transfer functions, etcetera. So, we are going to explore these possible topics very soon.

(Refer Slide Time: 18:43)

**Semi-Lattice**

- A semi-lattice is a set  $V$  and a binary operator  $\wedge$ , such that the following properties hold
  - 1.  $V$  is closed under  $\wedge$
  - 2.  $\wedge$  is idempotent ( $x \wedge x = x$ ), commutative ( $x \wedge y = y \wedge x$ ), and associative ( $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ )
  - 3. It has a top element,  $\top$ , such that  $\forall x \in V, \top \wedge x = x$
  - 4. It may have a bottom element,  $\perp$ , such that  $\forall x \in V, \perp \wedge x = \perp$
- The operator  $\wedge$  defines a partial order  $\leq$  on  $V$ , such that  $x \leq y$  iff  $x \wedge y = x$
- Any two elements  $x$  and  $y$  in a semi-lattice have a greatest lower bound (glb),  $g$ , such that  $g = x \wedge y$ ,  $g \leq x$ ,  $g \leq y$ , and if  $z \leq x$ , and  $z \leq y$ , then  $z \leq g$

YN Srikant Theoretical Foundations of CS

What exactly is a semi-lattice? A semi-lattice is a mathematical structure, an algebraic structure. It consists of a set,  $V$  and a binary operator, meet. The inverted  $V$  is called as meet. So, that is the one we are talking about.

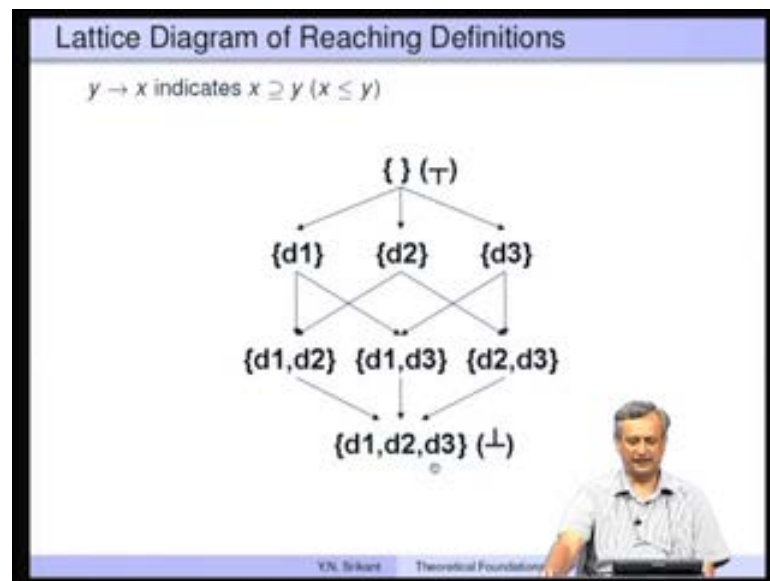
Why is it a binary operator? Because it takes two operands. What are the properties to be satisfied by a set and a binary operator in order to become a semi-lattice? The set  $V$  is closed under the meet operation. In other words, whenever we say  $V_1$  meet  $V_2$  and we get an element  $V_3$ , if  $V_1$  and  $V_2$  are in  $V$ , then  $V_3$  must also be in  $V$ . So, this a well-known closure operation and closure property. This must be satisfied for the set  $V$  and the binary operator meet.

Meet is idempotent; that is,  $x$  meet  $x$  is equal to  $x$ . So, nothing happens if you apply the meet on yourself. Meet is also commutative. So,  $x$  meet  $y$  is same as  $y$  meet  $x$ . Further, it is associative. So,  $x$  meet  $y$  meet  $z$ . So, we are associating  $y$  and  $z$  first and then with  $x$ . The answer you get will be the same as when you associate  $x$  and  $y$  first and then with  $z$ . So, idempotency, commutativity, and associativity are all true for this set and the binary operator meet. The semi-lattice also has a set  $V$ , rather has a special element called a top element. This is denoted as this  $T$  (Refer Slide Time: 20:46) and read as top. So, this is the highest element in some sense.

How do you formally state it? For all  $x$  in  $V$ , top meet  $x$  is always  $x$ . So, the meet operation with the top always gives you the same element. This is the definition of the

top element. There can only be one top element. So, there cannot be more than one in a semi-lattice. It may have a bottom element. It is not essential that the bottom element be present. So, this is the inverted T (Refer Slide Time: 21:26)  $(\perp)$  it has bottom. How do you define the bottom element? For all  $x$  in  $V$ , bottom meet  $x$  is bottom. So, in this case, top meet  $x$  came down to  $x$ , whereas bottom meet  $x$  always comes to bottom. So, this is the lowest element. That is why it is called a bottom.

(Refer Slide Time: 21:48)



Here is an example – we are going to show a semi-lattice in this form of a diagram called the Hasse diagram.  $d1 \ d2 \ d3$ ,  $d1 \ d2$ ,  $d1 \ d3$ ,  $d2 \ d3$ ,  $d1 \ d2 \ d3$  – so, there are many sets. Here is an empty set and then singleton element sets, 2 element sets, and finally, the 3 element sets. These are actually nothing, but the power set of this set  $d1 \ d2 \ d3$ ; this entire thing; the set of all subsets of  $d1 \ d2 \ d3$ . So, this empty set is the top element and  $d1 \ d2 \ d3$ , the entire set is the bottom element.

(Refer Slide Time: 22:39)

**Semi-Lattice**

- A semi-lattice is a set  $V$  and a binary operator  $\wedge$ , such that the following properties hold
  - 1.  $V$  is closed under  $\wedge$
  - 2.  $\wedge$  is idempotent ( $x \wedge x = x$ ), commutative ( $x \wedge y = y \wedge x$ ), and associative ( $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ )
  - 3. It has a top element,  $\top$ , such that  $\forall x \in V, \top \wedge x = x$
  - 4. It may have a bottom element,  $\perp$ , such that  $\forall x \in V, \perp \wedge x = \perp$
- The operator  $\wedge$  defines a partial order  $\leq$  on  $V$ , such that  $x \leq y$  iff  $x \wedge y = x$
- Any two elements  $x$  and  $y$  in a semi-lattice have a greatest lower bound (glb),  $g$ , such that  $g = x \wedge y$ ,  $g \leq x$ ,  $g \leq y$ , and if  $z \leq x$ , and  $z \leq y$ , then  $z \leq g$

YN Srikant Theoretical Foundations

The meet operator also defines a partial order less than or equal to on this set  $V$ . What is the property of this partial order?  $x$  less than or equal to  $y$  if and only if  $x$  meet  $y$  equal to  $x$ . In other words, you want to actually call something less than something else. That is why it is called as a partial order and it is defined using the meet operator  $x$  less than or equal to  $y$  if and only if  $x$  meet  $y$  equal to  $x$ .

(Refer Slide Time: 23:28)

**Lattice Diagram of Reaching Definitions**

$y \rightarrow x$  indicates  $x \supseteq y$  ( $x \leq y$ )

```
graph TD
    T["{} (τ)"] --> D1["{d1}"]
    T --> D2["{d2}"]
    T --> D3["{d3}"]
    D1 --> D12["{d1,d2}"]
    D1 --> D13["{d1,d3}"]
    D2 --> D12
    D2 --> D23["{d2,d3}"]
    D3 --> D13
    D3 --> D23
    D12 --> B["{d1,d2,d3} (⊥)"]
    D13 --> B
    D23 --> B
```

YN Srikant Theoretical Foundations

In this case, for example,  $x$  less than or equal to  $y$  is indicated by the arrow direction  $y$  to  $x$ . This is  $y$  and this is  $x$ . So,  $x$  less than or equal to  $y$ . So, the low elements are less than



or equal to the top element **in that sense**. This less than or equal to is the superset operator. So, this single element, singleton  $d_1$  is a superset of the empty set, the  $d_1 d_2 d_3$ , a full set is a superset of  $d_1 d_2$ , etcetera.

(Refer Slide Time: 24:04)

**Semi-Lattice**

- A semi-lattice is a set  $V$  and a binary operator  $\wedge$ , such that the following properties hold
  1.  $V$  is closed under  $\wedge$
  2.  $\wedge$  is idempotent ( $x \wedge x = x$ ), commutative ( $x \wedge y = y \wedge x$ ), and associative ( $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ )
  3. It has a top element,  $T$ , such that  $\forall x \in V, T \wedge x = x$
  4. It may have a bottom element,  $\perp$ , such that  $\forall x \in V, \perp \wedge x = \perp$
- The operator  $\wedge$  defines a partial order  $\leq$  on  $V$ , such that  $x \leq y$  iff  $x \wedge y = x$
- Any two elements  $x$  and  $y$  in a semi-lattice have a greatest lower bound (g/lb),  $g$ , such that  $g = x \wedge y, g \leq x, g \leq y$ , and if  $z \leq x$ , and  $z \leq y$ , then  $z \leq g$

VN. Srikant Theoretical Foundations of DFA

Any two elements  $x$  and  $y$  in a semi-lattice have a greatest lower bound  $g$  l b, what is the definition of  $g$ ? such that  $g$  equal to  $x$  meet  $y$ ,  $g$  less than or equal to  $x$ ,  $g$  less than or equal to  $y$ . So,  $g$  is lower than either  $x$  or  $y$  in that sense in the partial order. If there is any other element  $z$ , which is less than or equal to  $x$  and it is less than or equal to  $y$ , then  $z$  is also less than or equal to  $g$ . So, in some sense, the greatest lower bound is not the lowest element in the set. So, you really cannot have... If you have any other  $z$ , which is less than or equal to  $x$  and less than or equal to  $y$ , then it is also less than or equal to  $g$ . So, this is the largest element, which is smaller than either  $x$  or  $y$ ; both  $x$  and  $y$ .

(Refer Slide Time: 25:09)

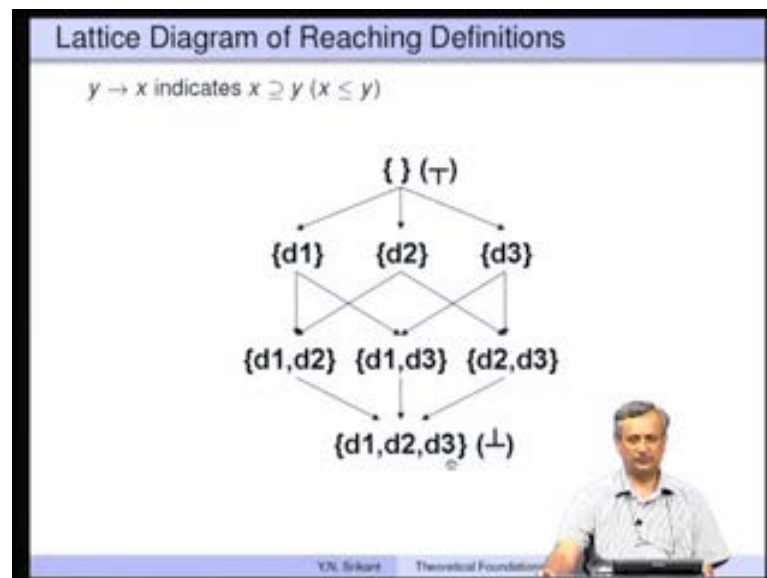
**Semi-Lattice of Reaching Definitions**

- 3 definitions,  $\{d1, d2, d3\}$
- $V$  is the set of all subsets of  $\{d1, d2, d3\}$
- $\wedge$  is  $\cup$
- The diagram (next slide) shows the partial order relation induced by  $\wedge$  (i.e.,  $\cup$ )
- Partial order relation is  $\supseteq$
- An arrow,  $y \rightarrow x$  indicates  $x \supseteq y$  ( $x \leq y$ )
- Each set in the diagram is a data-flow value
- Transitivity is implied in the diagram ( $a \rightarrow b$  &  $b \rightarrow c$  implies  $a \rightarrow c$ )
- An ascending chain:  $(x_1 < x_2 < \dots < x_n)$
- Height of a semi-lattice: largest number of ' $<$ ' relations in any ascending chain
- Semi-lattices in our DF frameworks will be of finite height

V.N. Srikant Theoretical Foundations of DFA

Let us look at the semi-lattice of reaching definitions. This is what I showed you here (Refer Slide Time: 25:16). Let us formally discuss it now. There are 3 definitions –  $d1$ ,  $d2$ , and  $d3$ .  $V$  is the set of all the subsets of  $d1$   $d2$   $d3$ .

(Refer Slide Time: 25:31)



I already showed you. There are three elements. So, 2 to power 3 is 8. We have 8 elements in  $V$ . The meet operator is the union operator. If you look at  $d1$   $d2$  and take the meet operation -  $d1$  meet  $d2$ , then we get the union of the two sets  $d1$  and  $d2$ . Similarly,

$d1$  and  $d3$  will give you  $d1 \ d3$ , the meet of  $d1 \ d2$  and  $d1 \ d3$  will give you  $d1 \ d2 \ d3$ , and so on and so forth.

The diagram shows the partial order relation induced by the meet operator. Here, this is the diagram we are referring to (Refer Slide Time: 26:17). We do not want to show the transitive relationship explicitly in this diagram. This is called as a Hasse diagram. The transitivity is implicit. For example,  $d1$  is a superset of  $\phi$  and  $d1 \ d3$  is a superset of  $d1$ . So,  $d1 \ d3$  is obviously a superset of  $\phi$ , but this should be inferred by the two arcs and there is no explicit arc between  $\phi$  and  $d1 \ d3$ . So, transitive relationships are not explicit in this diagram; otherwise, the diagram becomes too clumsy to understand and read.

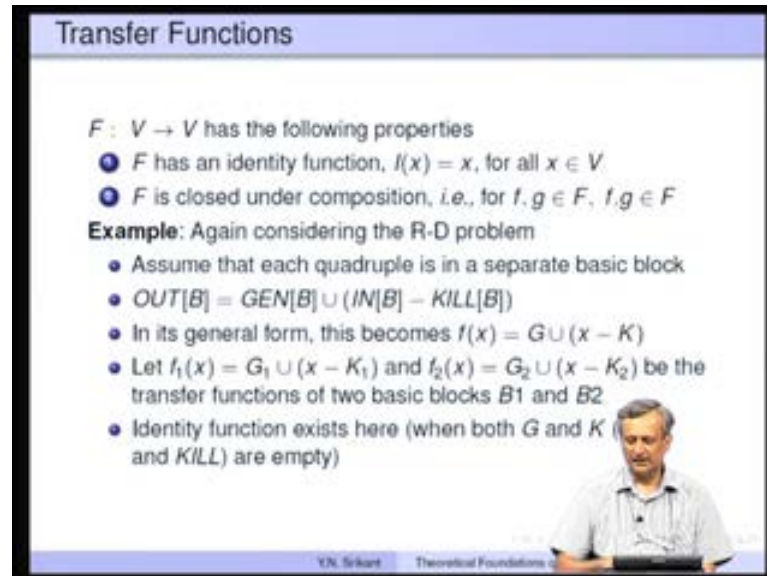
An arrow,  $y$  to  $x$  indicates  $x$  superset  $y$  or  $x$  less than or equal to  $y$ . This is something I already mentioned;  $x$  superset  $y$ . Each set in the diagram is a data-flow value. Now, we are talking. We have already seen the iterative data-flow analysis algorithm. So, at every point, we compute the IN and OUT, they are sets of data-flow values. That set is what we are talking about. So, the data-flow value is a set. So, reaching definitions is a semi-lattice and it forms a data-flow framework as we shall soon see.

Transitivity is implied in the diagram. I mentioned this already. What is an ascending chain? Instead of 'less than' or equal to, we just use less than. So, if we use the self-element also, then equal to is necessary; otherwise, we use distinct element, then less than is sufficient. So,  $d1$  less than or equal to  $\phi$ ,  $d1 \ d3$  less than or equal to  $d1$ , but since these are all different elements (Refer Slide Time: 28:22), we can get rid of the equal to and say -  $d1$  less than  $\phi$ ,  $d1 \ d3$  less than  $d1$ ,  $d1 \ d2 \ d3$  less than  $d1 \ d3$ . So, this is an ascending chain. This entire thing is an ascending chain (Refer Slide Time: 28:37).

The height of a semi-lattice is the largest number of less than relations in any ascending chain. If you look at this (Refer Slide Time: 28:45), 1 2 3, there are three edges we traverse to the top and there are three heights of this particular semi-lattice.

Semi-lattices in our dataflow framework will always be of finite height. We are not going to deal with infinite lattices, infinite height lattices rather. There may be number of elements as we will see soon.

(Refer Slide Time: 29:10)



**Transfer Functions**

$F : V \rightarrow V$  has the following properties

- $F$  has an identity function,  $f(x) = x$ , for all  $x \in V$
- $F$  is closed under composition, i.e., for  $f, g \in F$ ,  $f \circ g \in F$

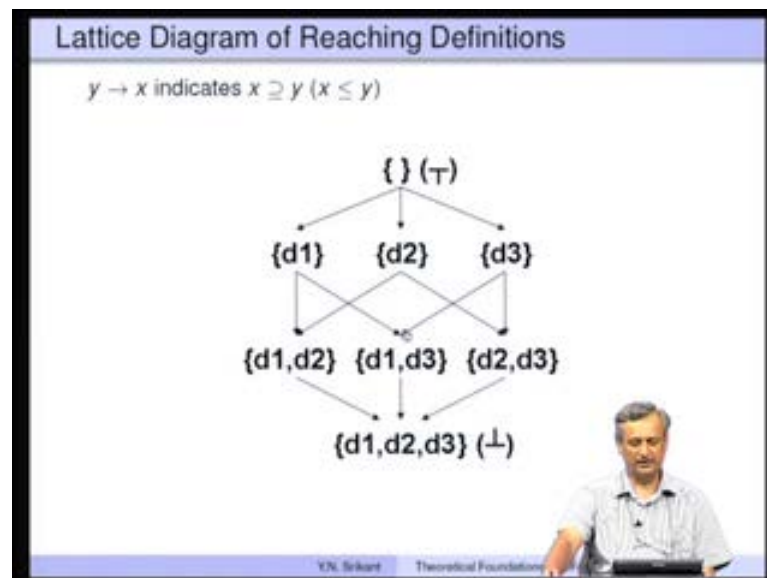
**Example:** Again considering the R-D problem

- Assume that each quadruple is in a separate basic block
- $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$
- In its general form, this becomes  $f(x) = G \cup (x - K)$
- Let  $f_1(x) = G_1 \cup (x - K_1)$  and  $f_2(x) = G_2 \cup (x - K_2)$  be the transfer functions of two basic blocks  $B_1$  and  $B_2$
- Identity function exists here (when both  $G$  and  $K$  and  $KILL$  are empty)

V.N. Srikant Theoretical Foundations

We discussed the semi-lattice. That was the set  $V$  and the meet operator that we talked about.

(Refer Slide Time: 29:24)



In the previous diagram, union is the meet operator and that is the confluence operator of our iterative data-flow analysis algorithm as well. That is why, whenever we take the meet, we take the union in the iterative data-flow analysis.

(Refer Slide Time: 29:41)

**Transfer Functions**

$F : V \rightarrow V$  has the following properties

- $F$  has an identity function,  $f(x) = x$ , for all  $x \in V$
- $F$  is closed under composition, i.e., for  $f, g \in F$ ,  $f \circ g \in F$

**Example:** Again considering the R-D problem

- Assume that each quadruple is in a separate basic block
- $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$
- In its general form, this becomes  $f(x) = G \cup (x - K)$
- Let  $f_1(x) = G_1 \cup (x - K_1)$  and  $f_2(x) = G_2 \cup (x - K_2)$  be the transfer functions of two basic blocks  $B_1$  and  $B_2$
- Identity function exists here (when both  $G$  and  $K$  ( $GEN$  and  $KILL$ ) are empty)

Y.N. Srikant Theoretical Foundations of DFA

The dataflow analysis framework also had a transfer function set  $F$ . What are the properties of the set  $F$ .  $F$  has an identity function  $I$   $x$  equal to  $x$  for all  $x$  in  $V$ . So, it does not do anything, takes an element and returns the same element.

This is a very important property.  $F$  is closed under composition; that is, if  $f$  and  $g$  are in the set  $F$ , then  $f \circ g$ , which is the composition of  $f$  and  $g$  is also in the set  $F$ . So, this is the one which enables us to take one quadruple at a time, locate the transfer function for that quadruple, then compose the two transfer functions, and get the transfer function for two quadruples. So, if we do this for all the quadruples in the basic block, we get the transfer function for the entire basic block. So, this is a very important property.

Let us again consider the reaching definitions problem. Assume that each quadruple is in a separate basic block. Then, we have this equation  $OUT\ B$  equal to  $GEN\ B$  union  $IN\ B$  minus  $KILL\ B$ . If we generalize this and call  $OUT\ B$  as  $f\ x$ , then  $IN\ B$  is actually the input that we are going to use; that is, we are talking about the  $x$ .  $B$  is not the parameter,  $IN\ B$  is the parameter and it is modified in this function. So,  $f\ x$  equal to  $G$  union  $x$  minus  $K$ , where  $G$  is  $GEN\ B$  and  $K$  is  $KILL\ B$ . This is the general form of the equation.

Other function - if we would consider two such functions, transfer functions, one for each basic block, which consists of only one quadruple in this case, we have  $f_1\ x$  is  $G_1$  union  $x$  minus  $K_1$  and  $f_2\ x$  is  $G_2$  union  $x$  minus  $K_2$ . So, these are the two transfer functions of two basic blocks:  $B_1$  and  $B_2$



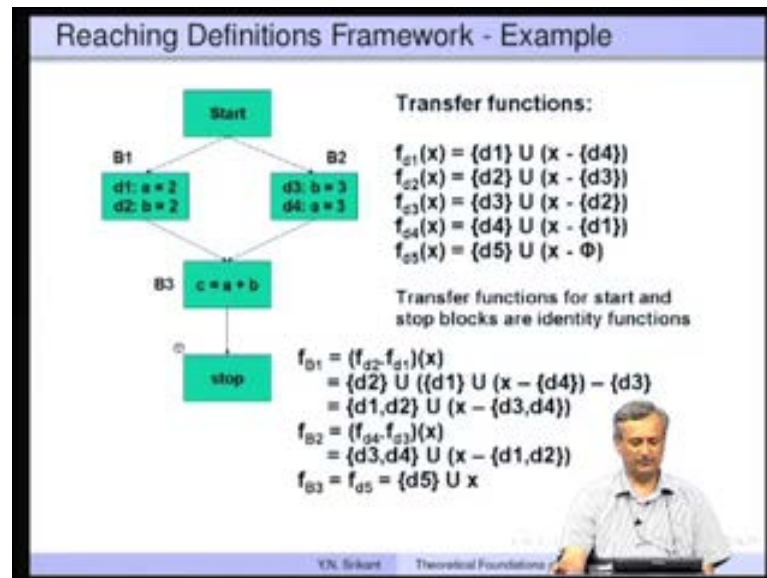
There is an identity function in the set. So, we easily verify, make  $G$  and  $K$ ; that is,  $G \cup \phi$  and  $K \cup \phi$ . In that case, we get  $f(x) = x$ ,  $G$  is  $\phi$ , and  $K$  is  $\phi$ . So, identity function already exist. So, if you consider this as the general form of transfer functions in the reaching definitions case,  $G$  and  $K$  are of course constants. So, this includes an identity function as well.

(Refer Slide Time: 32:50)

Suppose control flows from  $B1$  to  $B2$ ; there is an arc from  $B1$  to  $B2$  in the control flow-graph, then whatever comes to basic block  $B2$  is the output of the basic block  $B1$ . So, we look at  $f_2 \circ f_1$  of  $x$ . So,  $f_1$  processes the input  $x$  and output is given to  $f_2$  as input. That is why,  $f_2 \circ f_1$  of  $x$ . When we expand this and substitute, we get  $G_2 \cup ((G_1 \cup (x - K_1)) - K_2)$ ; this is our  $x$ , which is  $f_1(x) = G_1 \cup (x - K_1)$  and then minus  $K_2$  as usual for the  $f_2$ . Just rewrite the right hand side. So, you can write  $G_2 \cup (G_1 - K_2) \cup (x - (K_1 \cup K_2))$ . So, we can simply say  $K$  is  $K_1 \cup K_2$ ; this part (Refer Slide Time: 34:01).  $G$  is  $G_2 \cup (G_1 - K_2)$ ; that is very easy to see. So,  $G_1 - K_2$ . They are constants again;  $K$  and  $G$  are constants computed from  $K_1, G_1, K_2, G_2$ .

If you do this, then again is of the form  $f(x) = G \cup (x - K)$ . So, this is  $G$  and this is  $K$  (Refer Slide Time: 34:26). So, this entire  $f_2 \circ f_1$  of  $x$  is of the same form as the original equation.  $f_2 \circ f_1$  of  $x$  is nothing, but the composition of  $f_2$  and  $f_1$ . So,  $f_2 \circ f_1$ . Therefore, composition property is true. Further, reaching definitions example; the transfer functions of the reaching definitions example are composable.

(Refer Slide Time: 34:53)



Let us take an example instead of talking about just mathematical symbols. Here is a simple flow-graph B1 B2 B3 and then start and stop as well. There are two definitions in B1, two definitions in B2, and then one in B3. This is d5; that is missed out.

What are the transfer functions for the various statements? We are looking at one statement at a time. That is why, we have said f of d1, f of d2, f d3, f d4, and f d5 instead of basic block. So, the definitions generated by d1 are d1 union x minus d4. Why? d1 is defining a. So, it kills all statements, which are defining a. So, d4 is definition for a; that will be killed by d1. That is why, the kill function is d4 and kill constant is d4 k.

Similarly, f d2 is d2 union x minus d3. So, d3 is a definition of B. f d3 is d3 union x minus d2; d2 is a definition of B and similarly, for d4, d4 union x minus d1, whereas d1 is a definition of a. Interestingly, for d5, we have f d5 equal to d5; of course, union x minus phi because there is no definition of c elsewhere in the control flow-graph.

The transfer functions for start and stop blocks are just identity; they do not do anything. Now, let us compute the transfer functions of the basic blocks and show composition. f d2.f d1 x is d2 union d1 union x minus d4 minus d3. So, this is the f d1 part and the outer part is f d2 part (Refer Slide Time: 36:59). So, we get d1 d2 union x minus d3 d4. Intuitively, this is what we get. It generates d1 and d2 and kills all others with a and b. So, that is d3 d4. Similarly, f B2 is d3 d4 union x minus d1 d2. So, that is exactly the

other way.  $f \text{ B3 is } f \text{ d5}$ , which is  $\text{d5 union } x$ ; doing nothing else generates  $\text{d5}$  and then whatever is coming inside passes it out.

(Refer Slide Time: 37:35)

**Monotone Frameworks**

- A DF framework  $(D, F, V, \wedge)$  is monotone, if  $\forall x, y \in V, f \in F, x \leq y \Rightarrow f(x) \leq f(y)$ , OR  $f(x \wedge y) \leq f(x) \wedge f(y)$
- The reaching definitions lattice is monotone
- **Proof:**  $\wedge$  is  $\cup$ . Therefore, we need to prove that  $f(x \cup y) \supseteq f(x) \cup f(y)$   
 $f(x \cup y) = G \cup (x \cup y - K)$   
 $f(x) \cup f(y) = (G \cup (x - K)) \cup (G \cup (y - K))$   
 $= G \cup (x - K) \cup (y - K)$   
 $= G \cup (x \cup y - K) = f(x \cup y)$   
 Therefore, the Reaching Definitions framework is monotone

YN Srikant Theoretical Foundations of

What exactly is the monotonicity of the data-flow framework? This is a very important property and let see what it means. A data-flow framework  $D, F, V$ , meet is monotone, if for all  $x, y$  in  $V$  and  $f$  in the capital  $F$ ; that is, the  $f$  is a member of transfer functions family,  $x$  less than or equal to  $y$  implies  $f(x)$  less than or equal to  $f(y)$ . In other words, if  $x$  is smaller than  $y$ ,  $f(x)$  is smaller than  $f(y)$  in the lattice theoretic sense.

The other possible definition for monotonicity is  $f$  of  $x$  meet  $y$  is  $f(x)$  meet  $f(y)$ . Both definitions are identical because the less than or equal to is defined using the meet operator. It will be easy to show that these two are identical. Now, we claim that the reaching definitions lattice is monotone. Let see how.

The meet operator is union here in the reaching definitions framework lattice. What we need to prove is that  $f$  of  $x$  union  $y$ ; that is, here (Refer Slide Time: 39:02),  $f$  of  $x$  meet  $y$  is  $f$  of  $x$  union  $y$ . This less than or equal to is superset. So, we replace that with super set symbol.  $f(x) \text{ meet } f(y)$  is  $f(x) \text{ union } f(y)$ . Now, let us expand the  $f(x) \text{ union } f(y)$  part by substituting the transfer function. So,  $G \text{ union } x \text{ union } y \text{ minus } K$ . So,  $x \text{ minus } K$  is the original form. So, instead of  $x$ , you have  $x \text{ union } y$ . So,  $x \text{ union } y \text{ minus } K$ .

What about  $f(x \cup y)$ , the right hand side?  $f(x)$  is  $G \cup x \text{ minus } K$ ,  $f(y)$  is  $G \cup y \text{ minus } K$ , and therefore, we have  $G \cup x \text{ minus } K \cup y \text{ minus } K$ . Finally, we have  $G \cup x \cup y \text{ minus } K$ . Observe that this is same as  $f$  of  $x \cup y$ . So, the superset obviously is weaker than equal to. We have proved **that**  $f$  of  $x \cup y$  equal to  $f(x) \cup f(y)$ . So, superset is obviously is trivially true. This tells us that the reaching definitions framework is indeed a monotone framework. So, the monotonicity becomes very important for us when we say that the iterative analysis is going to terminate.

(Refer Slide Time: 40:46)

**Distributive Frameworks**

- A DF framework is distributive, if  $\forall x, y \in V, f \in F, f(x \wedge y) = f(x) \wedge f(y)$
- Distributivity  $\Rightarrow$  monotonicity, but not vice-versa

**proof:** If  $a = b$ ,  $a \wedge b = a$ , so,  $a \leq b$  (by definition of  $\leq$ )  
 From the definition of distributivity, we know that  $f(x \wedge y) = f(x) \wedge f(y)$   
 Substituting  $f(x \wedge y)$  for  $a$  and  $f(x) \wedge f(y)$  for  $b$ , in  $a \leq b$ , we get  $f(x \wedge y) \leq f(x) \wedge f(y)$ , which is the requirement of monotonicity

- The reaching definitions lattice is distributive

**Proof:** We have already proved during the proof of monotonicity of the RD framework, that  $f(x \cup y) = f(x) \cup f(y)$ . This proves distributivity

YN. Srikant Theoretical Foundations of

The next important property of the framework is the distributivity. A data-flow framework is said to be distributive, if for all  $x, y$  in  $V$  and little  $f$  in the transfer functions at  $F$ , we have  $x \text{ meet } y$  equal to  $f(x \text{ meet } y)$ . Recall that (Refer Slide Time: 41:14) we have already proved  $f$  of  $x \cup y$  equal to  $f(x) \cup f(y)$ . That is what is need here. So, trivially, we have already proved that the reaching definition lattice is distributive. We have already done that.

However, let us prove the other interesting result - if a framework or lattice is distributive, then it is indeed monotone, but if it is just monotone, then it is not necessarily distributive. Let us say - if equal to  $b$ , then  $a \text{ meet } b$  is  $a$ . This is obvious. So,  $a$  and  $b$  are the same.  $a \text{ meet } b$  equal to  $a$  or  $a \text{ meet } b$  equal to  $b$ ; both are the same. Let us say  $a \text{ meet } b$  equal to  $a$ . So, by the definition of the partial order relation less than or

equal to, if  $a$  meet  $b$  equal to  $a$ , then  $a$  less than or equal to  $b$ . So, this is the definition of this less than or equal to.

Now, apply the definition of distributivity. We know that we must have  $f$  of  $x$  meet  $y$  equal to  $f$   $x$  meet  $f$   $y$ . These two are now going to be substituted for  $a$  and  $b$  because  $a$  equal to  $b$  is needed and we have  $f$   $x$  meet  $y$  equal to  $f$   $x$  meet  $f$   $y$ . So, this is  $a$  and this is  $b$ . So, we substitute in this particular relation  $a$  less than or equal to  $b$ .

We get  $f$  of  $x$  meet  $y$  less than or equal to  $f$   $x$  meet  $f$   $y$ . This is exactly the definition of monotonicity. So, distributivity indeed implies monotonicity, but monotonicity does not imply distributivity trivially because this is less than or equal to, whereas equal to is a stronger relation. So, it is easy to construct a counter example to show that a lattice, which is monotone need not be distributive. We will see one very soon in the constant propagation framework later. So, we have proved that the reaching definitions lattice is also distributive. It is monotone and it is distributive.

(Refer Slide Time: 43:40)

The slide displays the following pseudocode and formulas:

```
{OUT[B1] = vinit;  
for each block B ≠ B1 do OUT[B] = ⊤;  
while (changes to any OUT occur) do  
  for each block B ≠ B1 do {  
    IN[B] = ⋀P a predecessor of B OUT[P];  
    OUT[B] = fB(IN[B]);  
  }  
}
```

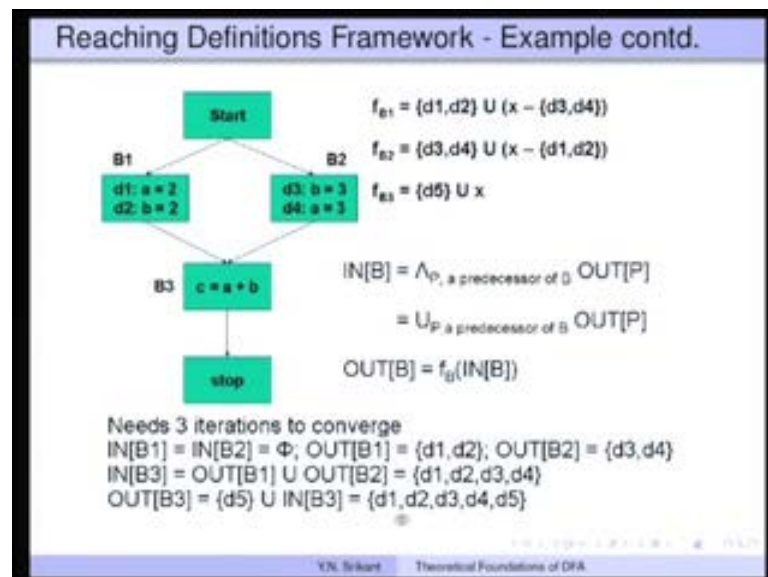
At the bottom of the slide, there is a small video inset showing a man speaking, and a footer that reads "V.N. Subramanian Theoretical Foundations of Compiler Design".

Now, let us take a look at the iterative algorithm for data-flow analysis. Assuming that there is a forward flow,  $OUT$  of  $B1$  is some initial value. For each basic block  $B$  not equal to  $B1$ , we have  $OUT$   $B$  equal to  $\top$ . Remember - the lattice of reaching definitions  $\top$  is  $\phi$ ; nothing. So, this is DFA algorithm for forward flow. So, we do not use any reaching definition confluence operator here, **but** we use the generic meet operator here. That is why the generic top value has been used; not  $\phi$ .

The change part remains while changes to any OUT occur do for each basic block B not equal to B1 do. So, we go on iterating until the OUT or IN sets are changing, or rather do not change any more. So, IN B is meet of OUT P, where P is the predecessor of B. So, we had union here in the reaching definitions case and intersection in the available expressions case. So, the generic operator is meet.

OUT B – now, you apply the transfer function f B on IN B. Remember - it was G union x minus KILL; x minus K. So, that is the transfer function we are applying on the set IN of B.

(Refer Slide Time: 45:29)



Let us take the same example and continue it. We have the transfer functions from our previous example f B1 equal to d1 comma d2 union x minus d3 comma d4. Similarly, f B2 and f B3.

Here are the two equations. This iterative algorithm requires 3 iterations to converge. So, in every iteration, we use the depth first search order. So, we start here (Refer Slide Time: 46:06), go here go here, we go here, here, then here, then this, and then this. So, when we come to this, both these results are available to us for the meet operation; otherwise, we need to make one more iteration. So, if we do that, apply these things - IN of B1 equal to IN of B2 equal to phi; that is, well known. Nothing is coming from the start block. OUT of B1 is d1 comma d2; that is very simple to see. So, these are the only two definitions, which come out. That is what we get. We get that by f B of IN B. IN B is



phi. So, if we apply  $f$  of  $B_1$ , then we get  $x$  minus phi. IN  $B$  is phi. So, this becomes phi minus  $d_3 d_4$  nothing. So, that is phi again. So, this part is  $d_1$  comma  $d_2$ ; that is what is mentioned here (Refer Slide Time: 47:05).

It is very similar. OUT  $B_2$  is similar. IN  $B_2$  is phi. So, we apply  $x$  as phi here (Refer Slide Time: 47:13). So, we get  $d_3 d_4$ . What about IN  $B_3$ ? We need to take the meet of these two (Refer Slide Time: 47:20); that is the union. So, OUT  $B_1$  union OUT  $B_2$ , which is  $d_1 d_2 d_3 d_4$ . These two are  $d_1 d_2$  and this is  $d_3 d_4$ . So, we get  $d_1 d_2 d_3 d_4$ .

What about OUT of  $B_3$ ? We now apply this transfer function  $d_5$  union  $x$ . That is  $d_5$  union IN of  $B_3$ ; that is  $x$ , the parameter coming in. So, we get  $d_1 d_2 d_3 d_4 d_5$ . This is intuitively also correct because whatever comes here, whatever comes here (Refer Slide Time: 47:52) and what is generated here will all reach this point. So, at this point, all the 5 definitions reach. That is why, the reaching definitions set contains all these here.

(Refer Slide Time: 48:05)

**Properties of the Iterative DFA Algorithm**

- If the iterative algorithm converges, the result is a solution to the DF equations

**Proof:** If the equations are not satisfied by the time the loop ends, atleast one of the *OUT* sets changes and we iterate again

- If the framework is monotone, then the solution found is the maximum fixpoint (MFP) of the DF equations

An MFP solution is such that in any other solution, values of  $IN[B]$  and  $OUT[B]$  are  $\leq$  the corresponding values of the MFP

**Proof:** We can show by induction that the values of  $IN[B]$  and  $OUT[B]$  only decrease (in the sense of  $\leq$  relation) as the algorithm iterates

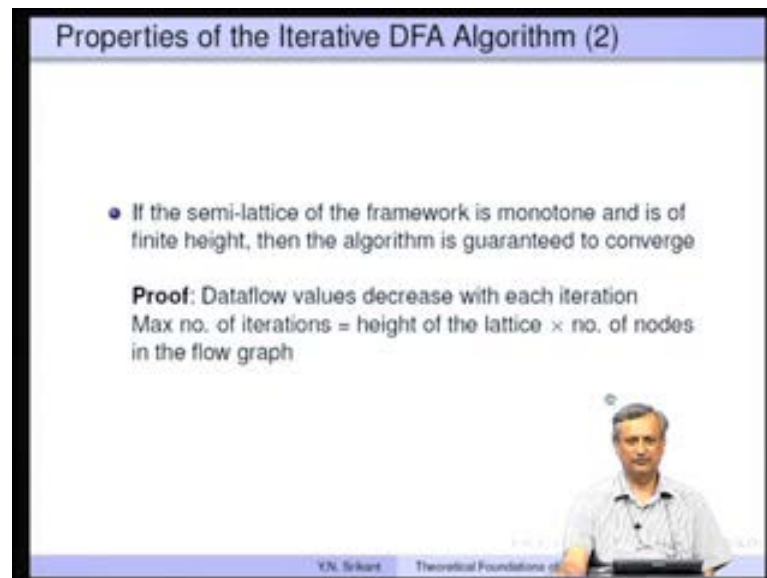
YN. Srikant Theoretical Foundations

What are the properties of the iterative data-flow analysis algorithm? We have defined many terms now - monotonicity, distributivity, composition of transfer functions, and so on and so forth. Let see where some of these actually fit. If the iterative algorithm converges, the result is a solution to the data-flow equations. We are not saying it converges; we will see that a little later.

If it converges, then the result is a solution to the data-flow equations. This is obvious because we iterate see - if the equations are not satisfied by the time the loop ends, at least one of the OUT sets change and then we iterate again. So, if they were not satisfied, then we would have iterated again and again. So, the constraints that we have set should have been satisfied. Only then, the OUT sets or IN sets do not change from one iteration to another and that is the time we stop it. If it does not change once, it does not change again and again in any of the other iterations as well.

If the framework is monotone, then the solution found is the maximum fix point of the data-flow equations. So, maximum fix point is a property such that in any other solution, values of IN B and OUT B are less than or equal to the corresponding values of the MFP. This is the maximum solution. So, any other solution is actually going to be just weaker. This property can be proved by induction. We can show by the induction that the values of IN B and OUT B only decrease in the sense of the less than or equal to relation. In other words, we start from the top of the lattice; that is, phi and then we actually move downwards. So, when we move downwards, we get weaker and weaker solutions in this less than or equal to sense.

(Refer Slide Time: 50:20)



The slide is titled "Properties of the Iterative DFA Algorithm (2)". It contains a bullet point: "• If the semi-lattice of the framework is monotone and is of finite height, then the algorithm is guaranteed to converge". Below this, it states "Proof: Dataflow values decrease with each iteration" and "Max no. of iterations = height of the lattice × no. of nodes in the flow graph". In the bottom right corner, there is a small video inset showing a man speaking. At the very bottom, there is a footer with the text "Y.N. Srikant Theoretical Foundations of...".

Now, if the framework is monotone and is of finite height, then the algorithm is guaranteed to converge. This is obvious almost because we initialize the values to phi and then with each one of the iterations, we can only move downwards. How many steps

can we go down? That is nothing, but the height of the lattice. So, the data-flow values decrease with each iteration; that is, we go down the lattice.

The maximum number of iterations is the height of the lattice into the number of nodes in the flow graph. We are not assuming that the **depth first search** numbering is used here. So, in the worst case, is just in upper bound height of the lattice into the number of nodes in the flow graph is the maximum number of iterations possible. You cannot really do anything more than that because for each one of these nodes, the stabilization occurs; let us say in one iteration. To get this stabilization, we need to pass through the entire lattice. So, from phi to the last one. So, that is the height of the lattice. For each node, we have to do it as many times. So this is the product.

(Refer Slide Time: 51:37)

**Meaning of the Ideal Data-flow Solution**

- Find all possible execution paths from the start node to the beginning of  $B$
- (Assuming forward flow) Compute the data-flow value at the end of each path (using composition of transfer functions) and apply the  $\wedge$  operator to these values to find their  $gib$
- No execution of the program can produce a smaller value for that program point

$$IDEAL[B] = \bigwedge_{P, \text{ a possible execution path from start node to } B} f_P(V_{init})$$

- Answers greater (in the sense of  $\leq$ ) than IDEAL are incorrect (one or more execution paths have been ignored)
- Any value smaller than or equal to IDEAL is conservative, i.e., safe (one or more infeasible paths have been included)
- Closer the value to IDEAL, more precise it is

YN Srikant Theoretical Foundations of DFA

Now, we want to look at the meaning of an ideal data-flow solution. So far, when we computed data-flow values and solved the equations in the iterative way, we always said the values are conservative, they are not the best that we can get, but practical situations forces to accept these conservative values and so on.

That makes it interested to see, what exactly is an ideal solution. To find the ideal solution is very simple. Find all the possible execution paths from the start node to the beginning of the basic block  $B$ . So, now, we are really looking at the execution paths. Some conditions may be true and some may be false during execution. So, we are looking at the path during execution.

Assuming forward flow, compute the data-flow value at the end of each path using composition of transfer functions and apply the meet operator to these values to find their greatest lower bound, glb. You have many paths coming to that basic block. Let us say - the input of the basic block. Now, at the input for each path, apply the composition of the various transfer functions of the basic blocks that occur along the path and find the value. Now, there are many paths. So, you take the glb.

IDEAL B is the meet of  $f P V$  init. This is the initial value. Now, what is P? P is a possible execution path from start node to B. So, no execution of the program can produce a smaller value for that program point. That is, the input of B in this case because we have considered all the execution paths. So, there is nothing else that can be possible. If we had considered one of the execution paths, then may be the other one would have produced a smaller value, but now we have looked at all execution paths and then found the values taken their meet. Any answer, which is greater in the sense of less than or equal to this partial order than this IDEAL value are incorrect; one or more execution paths have been ignored in this case. So, may be the meet did not take us low downwards sufficiently.

Any value smaller than or equal to IDEAL is conservative; that is, safe. So, one or more infeasible paths have been included in this case. So, this is what happens when we do iterative analysis. Closer the value is to IDEAL, more precise it is. This is the best that we can get and we still have not seen where we stand in the case of the iterative analysis.

We will understand the meaning of the values produced by iterative analysis, which are practical algorithms, in the next lecture. Thank you.