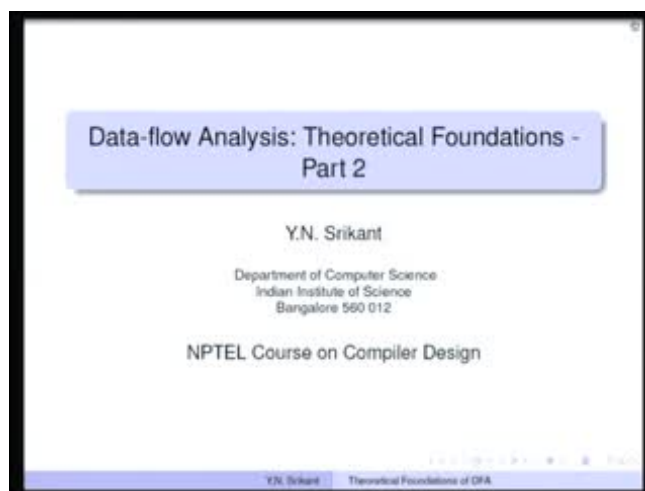**Compiler Design**
Prof. Y. N. Srikant
Department of Computer Science and Automation
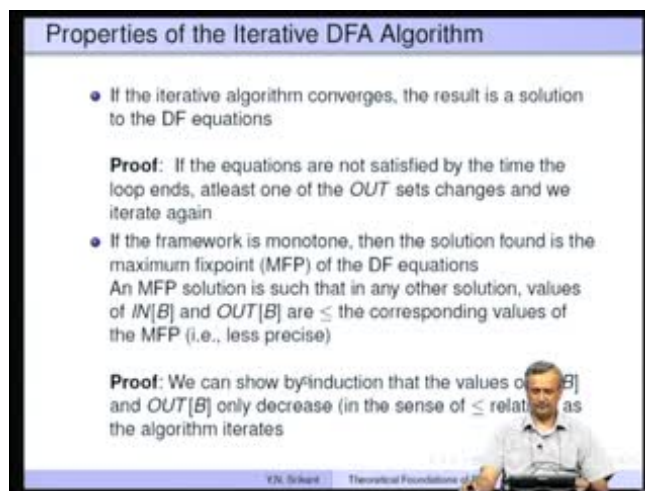**Indian Institute of Science, Bangalore**

Module No. # 11
Lecture No. # 28
Data-flow Analysis: Theoretical Foundation-Part 2 and
Partial Redundancy Elimination

(Refer Slide Time: 00:20)

Welcome to part two of the theoretical foundations of data flow analysis. This lecture looks at various theoretical aspects of data flow analysis; for example, what are the conditions under which the DFA algorithm delivers correct results and does it terminate? What exactly is the meaning of a solution and how far are we from the ideal solution etcetera.

(Refer Slide Time: 00:23)

Last time, we discussed the data flow framework which consists of direction - a domain of values, which is actually a semi lattice - a meet operator for the lattice and set of transfer functions. We started discussion on the properties of data flow algorithms, especially the iterative variety. The first property is if the iterative algorithm converges then the result is a solution to the data flow equations. This is true because, if the equations are not satisfied by the time, the loop ends an iteration, at least one of the OUT sets of the various basic blocks changes and therefore, we need to iterate once again. When all the outsets are stabilized and if it does not change again, then that would be a solution to the equation.

If the framework is monotone, then the solution found is the maximum fix point of the data flow equations. What is the maximum fix point? A maximum fix point solution is such that in any other solution the values of IN B and OUT B are actually less than or equal to the lattice theoretic sense corresponding values of the maximum fix point value; that is, that will be more conservative and less precise. In other words, if there is a reaching definitions problem, you may get a slightly bigger set in a solution which is not the MFP solution. That does not mean the MFP solution is the ideal solution.

We will actually discuss today, what exactly ideal solution is and so on. This property that we stated now can be proved by induction, the number of iterations of the algorithm and that is what we are going to induct upon.

(Refer Slide Time: 03:08)

Properties of the Iterative DFA Algorithm (2)

- If the semi-lattice of the framework is monotone and is of finite height, then the algorithm is guaranteed to converge

  Proof: Dataflow values decrease with each iteration
  Max no. of iterations = height of the lattice × no. of nodes in the flow graph

If the semi lattice of the frame work is monotone, if it is of finite height, then the algorithm is guaranteed to converge and it terminates definitely with a solution to the data flow equations. You see there are many nodes in the control flow graph that we are looking at; we have to actually scan every one of those nodes of the flow graph. Each flow graph possibly would change its values in each of its iteration. As we saw just now, the values only decreases, so from the top values it goes towards the bottom which is the most conservative part.

That is why the maximum number of iterations in the worst case would be the height of the lattice; that is, the maximum number of changes possible to the value multiplied by the number of nodes in the flow graph. This may not happen in practice, why? Well in practice, the number of iterations is actually 2 plus the depth of the flow graph, we have seen this in one of the previous lectures.

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 ∧ operator to these values to find their *glb*
- 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 ≤) 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

What exactly do we mean by the ideal dataflow solution? We will try to find an ideal solution by the following method and we find all possible execution paths from the starting node to the beginning of the basic block. Let us say, we are looking for IN values of basic block, then assuming forward flow we compute the dataflow value at the end of each path; that is, at the IN point of the basic block. For every one of these paths, we apply composition of the various transfer functions for the basic blocks involved. We get so many values, so we apply the meet operator to these values to find the greatest lower bound or the glb of these.

Since, we have considered every possible execution no other execution of the program can produce a smaller value than what we have just now computed. This ideal solution would be meeting of fp with an initial value V in it; it is the overall possible execution paths from the starting node to the beginning of B. Answers are greater than the sense of partial order, less than or equal to IDEAL are incorrect, so we would have left out some of the execution paths.

Any value smaller towards the bottom most point of the lattice; smaller than or equal to the ideal value is conservative - safe. We would have possibly included some infeasible paths also; that is, a condition possibly cannot be executed, but we have taken that branch and considered the path along that part as well, but the solution is safe. In other

words, you know the application will not deliver in correct results if we use this approximation.

(Refer Slide Time: 06:45)

Closer the value to its ideal, more precise it is. So, this is the exact solution, but this is not achievable in practice. Simply because, you just cannot find all the execution paths of a program, this is un decidable problem, we try to approximate this by including all paths in the flow graph itself. This is a graph theoretic path that we are talking about and not an execution path, this is called as meet over all paths solution MOP.

Here, we again take fp of V in it, but P is a path from start node to B, so we are considering all the graph theoretic paths from the start node to the beginning of the basic block B and not the execution paths. Because, we include the graph theoretic paths where some of them may not be executable and some of these paths may never be executed in any execution of the program; therefore, MOP solution tends to be a little more conservative than the ideal solution. That is why MOPB is less than or equal to IDEAL B since we consider a superset of the set of execution paths.

(Refer Slide Time: 07:46)

But even this is bit hard, finding all the paths in a flow graph may still be impossible. What happens if it has a cycle? Then there are infinite numbers of paths within the graph theoretic sense as well. So, you really cannot enumerate all the paths and do what we want. The iterative algorithm does not even try this, it visits all basic blocks not necessarily in execution order and it applies the meet operator at each join point in the flow graph. It does not try enumerating a path and then apply the meet operator for paths. Either the graph theoretic paths or the actual execution paths, it just applies it at every join point in the flow graph.

The solution obtained is the maximum fix point solution, so this is all the MFP solution that we discussed a little while ago. The iterative algorithm delivers this, so if the frame work is distributive then the MOP and MFP solutions will be identical. We will see that the constant propagation framework is not distributive and therefore, these two solutions do not become the same for that particular framework. Distributive property is very important for reaching definitions problem, the frame work is indeed distributive so the meet over all paths and the maximum fix point solutions will be identical. We can do very little about the ideal solution anyway; the best we can do is MOP. Here, we know that if it is distributive then MOP and MFP are identical and therefore, we should be happy with this result.

So otherwise, if we have just monotonicity like the constant propagation frame work then distributivity is false, MFP is smaller than the MOP solution and MOP is of course smaller than the ideal solution. But, whatever we get from the iterative algorithm is still safe; it may be conservative, but it is definitely safe.

(Refer Slide Time: 10:04)

(Refer Slide Time: 10:20)

So let us show that the MFP solution is indeed smaller than the MOP, rather is more conservative than the MOP solution. This is the flow graph that we will consider, so in several examples from now on start B_1 B_2, B_3 and B_4, there is a join point at B_3 from B_1
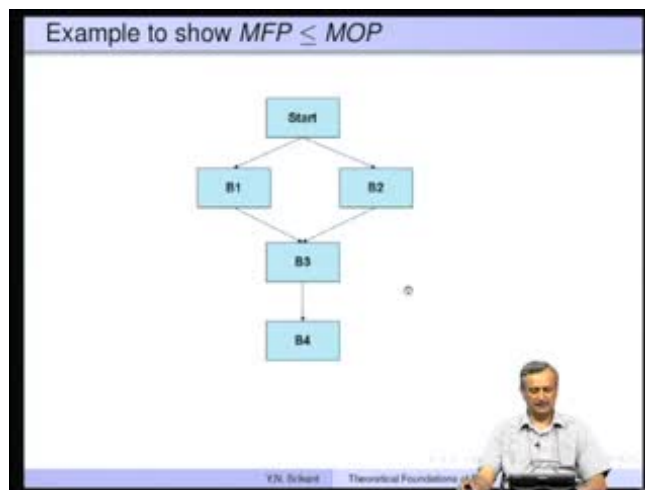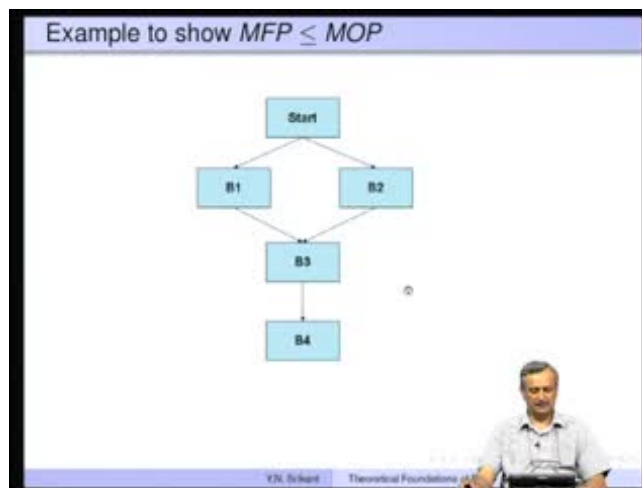
and B_2. There are two paths from start to B_4 so that is easy to see right start B_1, B_3, B_4 and start B_2, B_3, B_4.

(Refer Slide Time: 10:44)

(Refer Slide Time: 10:52)

Now, how do you compute the MOP over these flow graphs? It very easy, consider the transfer functions along this path; consider the transfer functions along this path as well, compose these transfer functions separately and then apply the join meet operator. So MOP of B_4 is f B_3 dot f B_1, so this is the first path; f B_3 dot f $_2$ this is the second path. Then we take the meet over these two and of course, the initial value of entire data flow

problem is V_{init}, so this is our MOP solution. In the iterative algorithm, we get the MFP solution, so to do that we must choose some order to visit the nodes of the flow graph. Let us say, we assume start B_1, B_2, B_3 and B_4, so that is start B_1, B_2, B_3 and B_4, so we do this, then this, then the join point and then B_4.

So IN of B_4 would be; so you do f B_1 of V in it, you do f B_2 of V in it, now we have a join point here, so we apply the meet operator on these two and then do the f B_3. This is the IN value for basic block B_4. Observe that this value MOP which is f B_3 dot f B_1 meet f B_3 dot f B_2, apply to V_{init}. This solution is f B_3 of f B_1 V_{init} meet f B_2 V_{init}; these two are different. You know they are not identical and these two will have equal value only if the framework is distributive. If the transfer functions over the lattice are distributive; they will satisfy the distributive property then this and this will be identical, they will deliver the same value.

So with just monotonicity IN of B_4, this value will always be less than or equal to MOP of B_4. This property is well known, this shows that the maximum fix point value will always be less than or equal to the meet over all paths value.

(Refer Slide Time: 13:28)
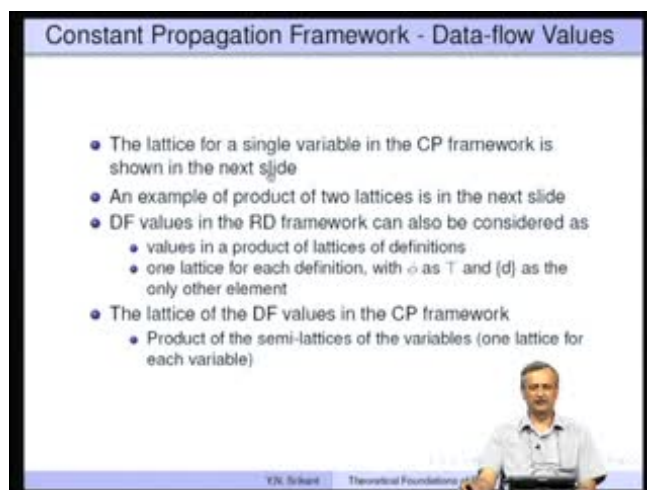
Let us move on to the constant propagation framework; what is the constant propagation problem? The constant propagation problem is to propagate the constant values of variables down in the flow graph. Whenever there is constant value assigned to a variable and it does not change, whenever the variable is used we will use the constant

value, some something like copy propagation applied to constants. Whenever there is an expression involving constants, evaluate that expression and the compiler can do this. There is no need to wait for the run time system to do this. We can now go on using the evaluated value of the expression - constant expression and do constant propagation further.

(Refer Slide Time: 14:44)
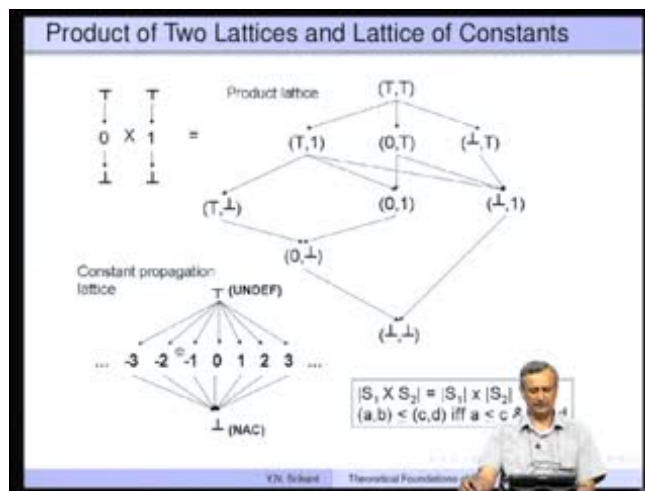
What is the lattice for the constant propagation framework? Let us first consider a single variable. Here is the constant propagation framework of a single variable, you have a top; you have a bottom and then the constant values. The top says the variable is undefined, the bottom says it is definitely not a constant. In between the variable can take any of these values; they are in comparable, so in fact the combination or evaluation of expressions is defined over and above this lattice as a transfer function and not as a part of the lattice as such.

This is the abstraction that we are going to look at. This entire middle will be termed as c - a constant. This top is another abstract value; bottom is one more abstract value, so we have three abstract values for the constant propagation framework. One is the top or UNDEF, then the constant infinite number of values and then the bottom which is not a constant. Here, I should point out that this lattice is indeed infinite; you know the number of values possible for the constants is definitely infinite, not necessarily in the practical way, but in theoretical way. The height of the lattice is finite; you know this is just 2.
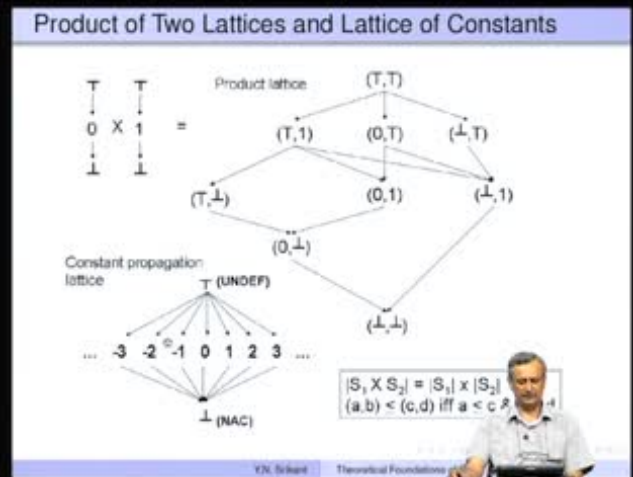
(Refer Slide Time: 16:10)

(Refer Slide Time: 16:25)

We also need to understand what exactly the product of two lattices, because the constant propagation framework requires the product of lattices. Let us take very simple lattices of two of them; top 0 bottom and top 1 bottom, we want to do cross product of these two lattices. We are going to consider all possible pairs of values, of course from one to the other in a cross product, top top, top 1, top bottom. You know 0 top, 0 1 and 0 bottom; bottom top, bottom 1, bottom bottom, so all these values are here.

Now, when can we say that this value is smaller than this value or this value is smaller than this value? In general, we can say that a coma b, so this is a pair in the product lattice, c coma d is another pair in the product lattice. So a coma b is less than or equal to c coma d, if and only if the first two components are less than or equal to; a less than or equal to c. Second two components are again related by less than or equal to; b less than or equal to d.

First component belongs to the first lattice and second component belongs to the second lattice. The number of elements in the lattice will be number of elements in S_1 multiplied by the number of elements in the other one, so 3 into 3; 9 elements here.

Let us verify the inequality in this; it is built upon the inequality from this single lattice. You know 0 less than or equal to top, bottom less than or equal to 0. Let us take the 0 bottom and 0 1, so 0 less than equal to 0 is not a problem and bottom less than or equal to 1 is also not a problem. This is the way; for example, you know we cannot have any connection between top coma bottom and 0 coma top, simply because top is not less than or equal to 0. Even though bottom is not equal to top, top is not less than or equal to 0, you can see that. This does not show any transitive relations as usual in the case of a lattice diagram, it shows only the immediate relationships; transitivity is to be inferred from the relationships and inferred from the diagrams.
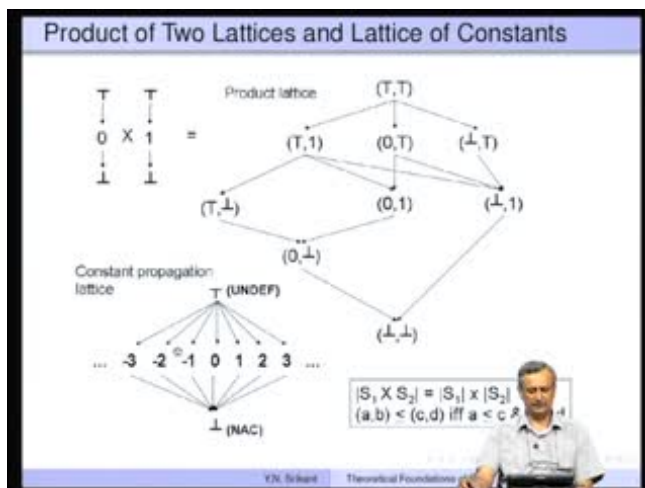
(Refer Slide Time: 19:04)

(Refer Slide Time: 20:13)

Data flow values in the reaching definitions frame work can also be considered as values in a product of lattices of definitions. So, we had considered the reaching definitions lattice, in which the subsets of the definitions are the elements of the lattice. Another way of looking at the problem is you have one lattice for each definition, with phi as top and d as the only other element, no other bottom; two element lattice. Now take a product of the lattices of each definition, if there are ten definitions, then you have a ten fold product. So finally, you will have a product lattice for all the reaching definitions rather all the definitions in the program. You can use exactly the same type of argument that we gave here to construct that lattice. Finally, argue about the maximum fix point solution and so on.

(Refer Slide Time: 20:21)

Observe that the number of elements is going to be the same, so whether we use the product lattice or the subsets of all sets of n elements, it will have exactly the same number of elements. If there are n definitions, each one has two elements, so 2 into 2 into n times is 2 to the power n anyway, so we still have the same number of elements in the reaching definition lattice considered as a product as well.

It is possible to consider many others also as product of lattices. The lattice of the dataflow as used in the constant propagation framework is the product of semi lattice of the variables. One for each variable exactly is like in the RD scheme that I mentioned now.

(Refer Slide Time: 21:20)

We already saw this in a product lattice a_1, b_1 is less than or equal to a_2, b_2 if and only if a_1 is less than a_2 and b_1 is less than b_2 in the appropriate lattices. Now, we associate a map with each variable, so the map actually uses the abstract value as in the lattice. I showed you the abstract value in this case; the bottom and this constant value or rather the top and this constant value, or the bottom.

So m v is the abstract value as in the lattice of the variable v in a map, so if variable v is undefined then the m v gives us UNDEF or top. If the variable value is a constant, then it gives you the actual constant value; if the variable value is not at all a constant - certainly not a constant, then it gives us the bottom or NAC value. So, this is what the m v map for each variable gives us.

Each element in the product lattice is a similar but have larger map; now this map for the product lattice is defined for all variables, so again the product lattice would also have a top bottom and the middle. m v is the abstract value of the variable, therefore in the product lattice if you are going to have m less than or equal to m prime, then for all variables v we should have m v less than or equal to m prime v; that is what we need to have. So, you have to have this map, which this tells you take m v if it is top or bottom, it is trivially satisfied, but if it is somewhere in the middle, then m v must be less than or equal to m prime v. Otherwise, we do not have any connection between these values or in the meet sense; m meet m prime is m double prime, if m double prime v is m v meet

m prime v, for all variables v. So here, it was just for that variable and here, we have for m for single variable, but here, m v must be less than or equal to m prime v for all the variables. The bigger map here is defined for all variables and that is why we are justified in saying that m v less than or equal to m prime v.

(Refer Slide Time: 24:28)
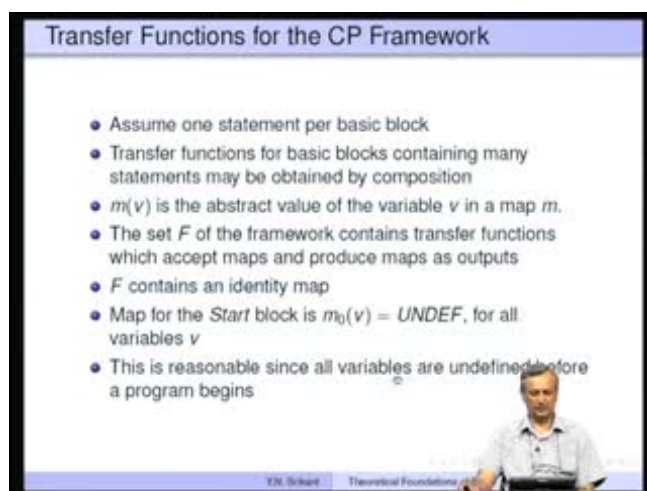
Let us look at the transfer functions, so far we looked at the meet operator in the constant propagation framework. This is the meet operator definition; m meet m prime. Again remember, we have a product lattice of the individual variables. Now let us look at the transfer functions for the constant propagation framework.

Assume that you have one statement per basic block. Transfer functions for basic blocks containing many statements obtained by composition; this is a very simple thing which we have already seen. So m v is abstract value of the variable in a map m, so this is the other assumption.

The set F of the framework contains transfer functions, which accepts maps and produce maps as outputs. The transfer functions here, they take a map and give you another map. That is the way they are, because the elements of the product lattice are maps, so the transfer functions also must work on maps.

So F contains an identity map, this is a requirement of the frame work itself. Map for the start block is m_0 v equal to UNDEF for all variables. This is reasonable because nothing is really defined before the program begins.

(Refer Slide Time: 25:47)

Transfer Functions for the CP Framework

- Let f_s be the transfer function of the statement s
- If $m' = f_s(m)$, then f_s is defined as follows
  - If s is not an assignment, f_s is the identity function
  - If s is an assignment to a variable x, then $m'(v) = m(v)$, for all $v \neq x$, provided, one of the following conditions holds
    - (a) If the RHS of s is a constant c, then $m'(x) = c$
    - (b) If the RHS is of the form $y + z$, then

    $$m'(x) = m(y) + m(z), \text{ if } m(y) \text{ and } m(z) \text{ are constants}$$
    $$= NAC, \text{ if either } m(y) \text{ or } m(z) \text{ is NAC}$$
    $$= UNDEF, \text{ otherwise}$$
    - (c) If the RHS is any other expression, then $m'(x) =$

Let us say f_s is the transfer function of this statement, let us see how to define f_s. So m prime is f_s applied on m, it takes a map and gives you another map. How is f_s defined? If s is not an assignment, then f_s is an identity function so very simple; it is not an assignment at all, then f_s is just a simple identity function.

If s is an assignment to a variable x, then m prime v is equal to m v for all v not equal to x, provided one of the following conditions also holds. What are the various possibilities now? We are looking at assignments, RHS of s is a constant, so we have x equal to c. Therefore, we must modify the map and then say m prime x now becomes c, it is a constant. For this variable the map will return constant value c, whatever is the constant value.

If the RHS is an expression; y plus z, so what is the possibility? Then you have three cases possible, it is either m y plus m z, if both m y and m z are constants. Apply m y, you may get y is a variable, z is a variable. m y will give you some value, m z will give you some other value, if these two are constants then you just add the two constant values and that is what happens to be the m prime x. The map m prime for x will return that particular constant value from now on, so this is constant folding.

Either m y or m z is not a constant value, which is already proved then m prime x will also become NAC. If one of them is not a constant, you cannot have y plus z as a constant anyway, in all other cases it is UNDEF. This is the map for the variable x in the new m prime, for all other variables the map remains the same as before, whatever the value is retained, but for x the map has now changed. If the RHS is any other expression then m prime x is NAC, this just covers all other possibilities which we have not been intelligent enough to think off.

(Refer Slide Time: 28:38)

This is regarding the transfer function for the framework; now let us start looking at the properties of the transfer function. This you know, these transfer functions actually satisfy the monotonicity property. Here is the single lattice, rather lattice for a single variable. Now, let us consider m y, m z an m prime x ,the way we have just now defined it. m y let us say is UNDEF, then if m z is UNDEF, m prime x is also UNDEF. If m z is c a constant, m prime x is still UNDEF, because m y is UNDEF.

Observe that if one of them is NAC, m z is NAC which is not a constant, then m prime x becomes not a constant. Similarly, when m y is a constant if m z is UNDEF, we remain at UNDEF if m z is a constant, then we have the value c_1 plus c_2 and if m z is NAC, then we have NAC. So far for NAC it does not matter, what we remain at NAC. This is the definition we have just now provided. What should we observe here, it must be noted that the transfer function always produce a lower or same level value in the CP lattice

whenever there is a change in inputs. Let us say one of the values is UNDEF and the other value changes from UNDEF to c_2 to NAC, let us see how m prime changes.

It was UNDEF at this point, then for c_2 it was still at UNDEF and for NAC it moved down all the way to NAC. It went from the top to the bottom as the input z change from UNDEF to NAC. When it was m y c_1 it starts with UNDEF, then it goes to c_1 plus c_2 which is again NAC constant value and then it goes to NAC, with NAC it always remains at NAC, m prime x always remains at NAC. In other words, whenever we have changed the input, the change in m prime x has always been from the top to the bottom, it has never changed from bottom to the top. It was not as if it was a constant and then it became UNDEF or it was not a constant then it became constant, this has never happened. It has not gone up and then come down etcetera; it always travel from the top towards the bottom.

This is precisely what we require for monotonicity, whenever there is a change in input in one direction, the change in the output should also be in the same direction. That is something that we want, if this is satisfied and therefore, the CP framework is monotone.

(Refer Slide Time: 32:02)
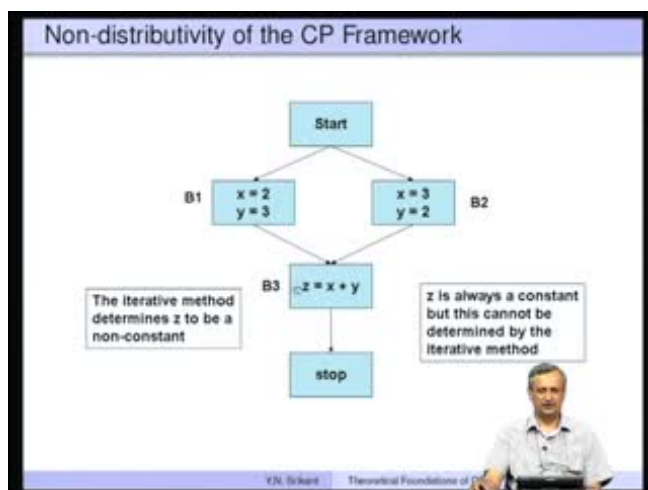
We are going to prove that the CP framework is not distributive. This is a very important thing to show, because probably this is one of the most famous examples which show that distributivity is important. Our problem such as reaching definitions and so on, the

frameworks are indeed distributive, so there is no problem about that we have seen this in the last lecture.

The implication is, since the CP framework fails to be distributive and it is only monotone, our MFP solution will not be as good as the MOP solution, it will be slightly inferior. What exactly do we mean by this is; this example will show that.

We have x equal to 2, y equal to 3; and x equal to 3, y equal to 2. Here, we have z equal to x plus y, so if you add up the values of x plus y at this point, we know that both sides gives you exactly 5. So 2 plus 3 is 5, 3 plus 2 is also 5. So the value of z irrespective of the path taken, either through B_1 or through B_2 during execution is actually going to be 5. The problem is our data flow analysis cannot determine this, because the value of x here and the value of x here are different, x equal to 2 and x equal to 3. The value of x at this point is not a constant, it is either 2 or 3, but it is not the same, it is not a single value. The same is true for y, y is 3 or 2, it is not a constant at this point and therefore we deem that z is also not a constant, so how do we show this formula.
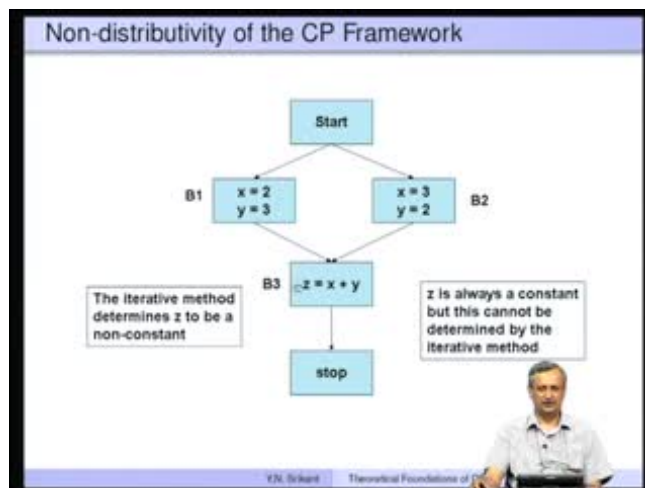
(Refer Slide Time: 34:05)

(Refer Slide Time: 34:47)

Let us look at f_1, f_2, f_3 as transfer functions of the three basic blocks B_1, B_2, B_3. Here is m, it is either m x, m y, m z and then m_0. So this is the initial map, it is UNDEF for all variables. You apply f_1 m_0, so what we want to show is f_3; you apply the meet operation at the join point and not after taking the path that is what we mean. So, f_1 m_0 meet f_2 m_0 and then f_3, so this is the MFP there, whereas f_3 of f_1 m_0 meet f_3 of f_2 m_0, so this is the MOP there.

Let us evaluate for this example; for these transfer functions and see what happens. Apply f_1 of m_0, m of x is 2, m of y is 3 and m z is still UNDEF, because we just want to apply for x and y. Remember, here we have only x and y we do not have z. Similarly for f_2 m_0 we do not have z here. So we get x as 3 and y as 2 and this remains as UNDEF.

Now take that meet, so f z remains UNDEF, but 2 and 3 give you not a constant, 3 and 2 also give you not a constant, because they are not equal, it always gives you the glp as not a constant. We go downwards then apply f_3, well the damage is already done and you already have an NAC. So applying any transfer function to NAC will not produce anything but NAC, so z, now get NAC from our definition.

So now on f_1, this is the MFP, you know it is part of it. Now let us do the MOP part, take f_1 m_0 and then apply f_3 on it. So we get 5. See this, so this is z. x and y do not change, only this. We are taking one path; by calculating z we get 5, take the other path and calculate z, so along this path z is a constant, so we get 5. Then take the meet of these

two 5, you get 5 itself for these 2 NAC. The MOP solution gave us the value 5, constant value for z, whereas the MFP solution gave us NAC not a constant as the solution.

This shows that the two values are different and therefore, the constant propagation framework is indeed non distributive. This is the end of this particular lecture on theoretical foundations, now we will continue this in the next lecture on partial redundancy elimination.

(Refer Slide Time: 37:58)
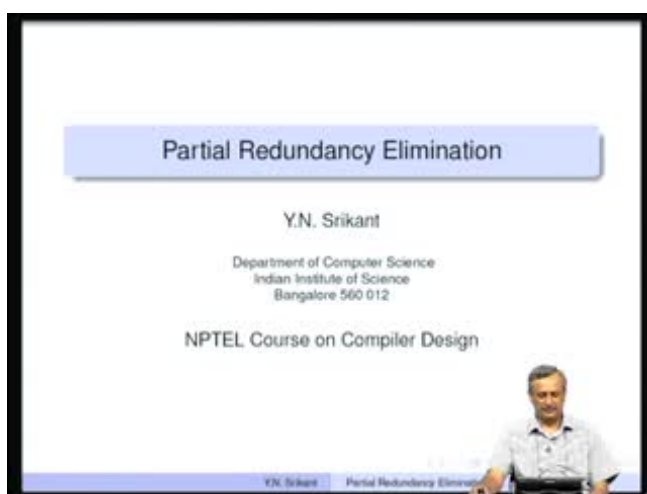
(Refer Slide Time: 38:07)
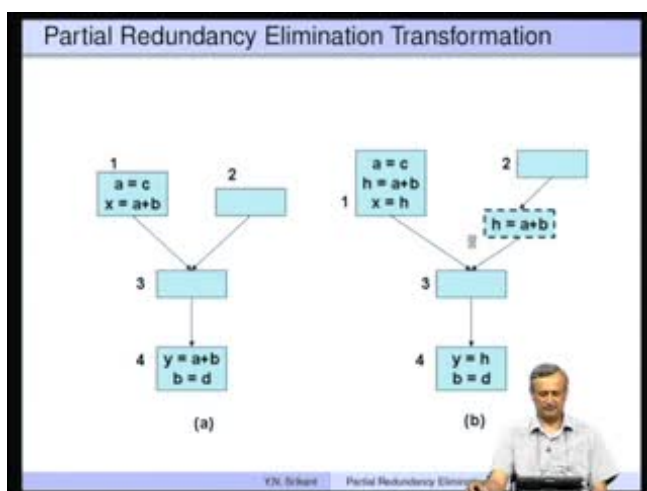
Welcome to the lecture on Partial Redundancy Elimination, so this is a very important optimization and therefore, a complete lecture is being devoted to this particular optimization. We saw an example of what exactly is PRE in the lecture on introduction to optimizations, so I am producing the same example here to recapitulate what the transformation is.

We have a very small flow graph in which this basic block has x equal to a plus b and then the basic block number 4 also has y equal to a plus b, but the basic block number 2 and basic block number 3 do not have any definitions rather than the evaluations of a plus b. what has happened is in this particular expression, if we follow this path it will compute twice, whereas if we follow this particular path it is being computed once. So can we eliminate computing this particular expression twice; we can do so, but how? We introduce the computation of a plus b on this particular path, now the this a plus b is available along this path and also along this path, so we have simply applied global common sub expression elimination; h equal to a plus b, x equal to h, h equal to a plus b and y equal to h.

So the basic idea is if an expression is available along one or more path but not along all paths, then it is said to be partially available. Such partially available expressions are turned into fully available expressions by introducing some extra computations. Then, we apply GCSC and eliminate the common sub expression. Now you can observe that a plus b is computed exactly once along this path and also along this path. The disadvantage is that you have introduced extra code so there is a bit of code blob, but the advantage of applying partial redundancy elimination overwhelms the bad effects and therefore, we always go ahead with PRE in a very standard compiler.

(Refer Slide Time: 40:53)

Let us begin with some definitions; what is a partially redundant computation? A computation is performed twice or may be more number of times in a certain path, so this is a partially redundant computation. So what is partial redundancy elimination? It involves insertions and deletions of computations to ensure that no partially redundant computations exist, so i just showed you the example of this now.

(Refer Slide Time: 41:45)

An important aspect of PRE algorithm that we have is safety, so we should not introduce any computations of new values on any path in the program. For example, in this path,

suppose there was no computation of a plus b along some other path. Let us say this we are not supposed to introduce a new computation along this path, so if it has one, fine; if it does not have one then we should not introduce it; so that is the basic idea. The number of time it is evaluated is immaterial, here it is 2 and it is 1 that is ok, but 0 and more than 0 really matter.

(Refer Slide Time: 42:14)
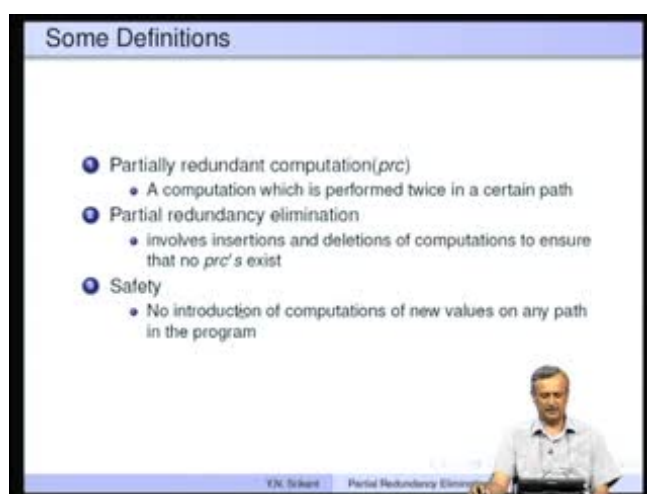
So safety is a very important feature of our algorithm, we do not want to introduce computations of new values on any path in the program, why? If we do this may be the semantics of the program can be retained by the compiler, there is no problem about that but, when we run the program it may lead to some run time errors because of the limitations of CPUs on the overflow of values and underflow of values, things of that kind. So, we do not want any such exceptional situations to occur by introducing extra computations.
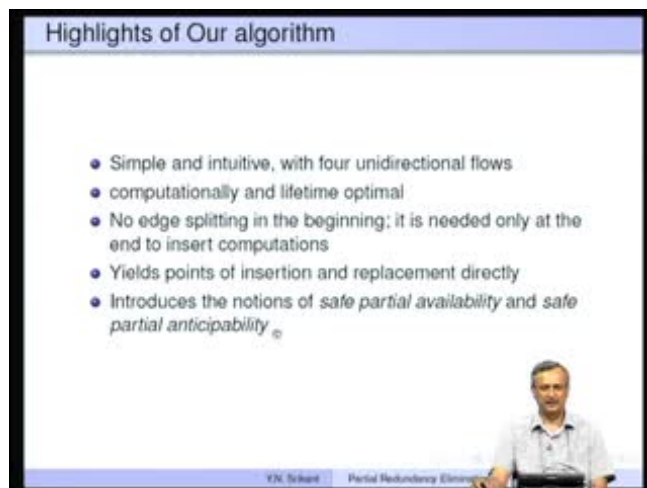
(Refer Slide Time: 42:56)
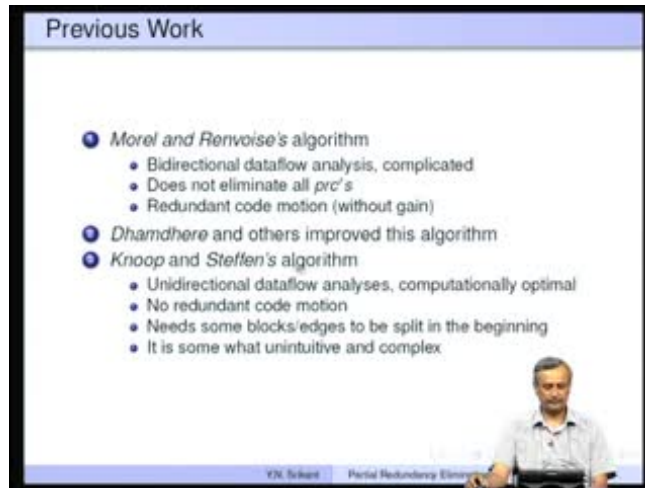
There has been plenty of previous work, the seminal research papers of Morel and Renvoise. This was a bidirectional data flow analysis, in other words we have seen forward and backward analysis, both these is involved in the algorithm, then it becomes a bidirectional data flow analysis and therefore, the problem is complicated. It does not eliminate all the partial redundant computations, some of them still remain. Redundant code motion without gain happens, so the code gets moved but there is no gain at all.

Dhamdhere and others improved this algorithm, but then the optimal version was delivered by Knoop and Steffen. They transformed the problem to unidirectional data flow analysis problem, so there was no bidirectional analysis required, no redundant code motion happened. Computationally it was optimal, nothing better could be done, but it needs some blocks and edges to be split right in the beginning, this is a disadvantage, it is somewhat unintuitive and very complex to understand.

(Refer Slide Time: 44:20)

Highlights of Our algorithm

- Simple and intuitive, with four unidirectional flows
- computationally and lifetime optimal
- No edge splitting in the beginning; it is needed only at the end to insert computations
- Yields points of insertion and replacement directly
- Introduces the notions of *safe partial availability* and *safe partial anticipability*

The highlights of our algorithm are, it is simple, intuitive and it has the same four unidirectional flows as in the case of Steffens algorithm. Computationally and lifetime optimal: again, as in the case of Knoop and Steffens algorithm. Here is the basic advantage - no edge splitting in the beginning; it is needed only at the time of insertion of the computation that is at the end of the algorithm itself.

The second advantage is, it yields points of insertion and replacement directly, so there is no need to compute this separately, this is a part of the algorithm. It introduces the notions of safe partial availability and safe partial anticipability, which need not exist in the previous works.

(Refer Slide Time: 45:12)

Highlights of Our algorithm

- Every safe partially redundant computation offers scope for redundancy elimination
- Any safe partially redundant computation at a point can be made totally redundant by insertion of new computations at proper points
- Computation of any expression that is totally redundant can be replaced by a copy rule
- After the transformation, no expression is recomputed at a point if its value is *available* (not partially) from previous computations

What we basically do is; every safe partially redundant computation, we have not yet defined this term. It offers scope for redundancy elimination, so we turn every safe partially redundant computation at a point to a totally redundant computation, by introducing a new computation at appropriate points and by breaking certain edges.

The computation of any expression is totally redundant can now be replaced by a copy rule. So, this is as in the case of global common sub expression elimination. After the transformation no expression is recomputed at a point if its value is available from the previous computations, all these are highlights of our algorithm.

What are the properties that we require at a point? These are the basic things that we require. We already know available expression of dataflow analysis, the availability of expressions. In all our discussion on PRE we always referred to properties of expressions, all the properties are defined with respect to individual expressions, for each expression there is going to be a set of properties; for example, availability, partial availability, anticipability, partial anticipability and safety, etcetera. How do you define the availability of an expression? The expression is computed along all paths reaching p from the start node with no changes to operands, so this is availability. It is as usual, available at a point implies this anyway.

The basic difference here is we are going to have a bit vector, which shows availability of various expressions in various basic blocks. Rather for a single expression, in various basic blocks we are going to have availability shown as a bit vector. So, rather than one bit for each expression there is going to be one bit for each basic block and a bit vector for each expressions. Partial availability means, computed along at least one path to the point p, so we have seen an example of this already, we will see more very soon

(Refer Slide Time: 47:47)
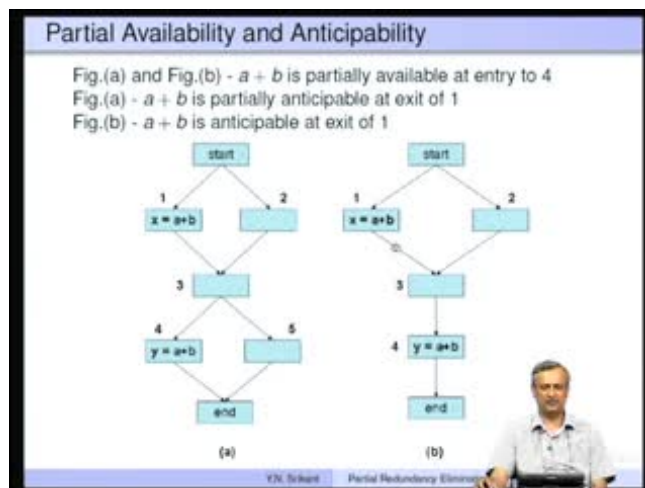
Here is an example; take this figure, the figure a and b; figure a plus b is partially available at entry to 4. Take 4 here, so entry to 4; suppose we traverse of this path a plus b is computed here and therefore it is available ,but if we follow this path a plus b is not computed along this path and it is not available, so we say a plus b is partially available at this point.

(Refer Slide Time: 48:23)
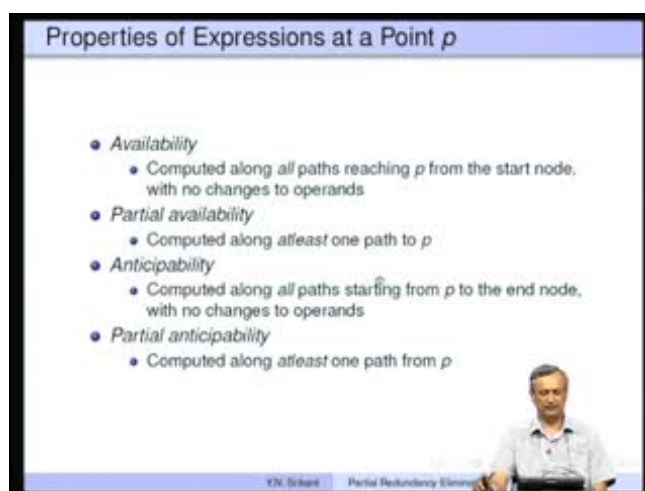
The same is true here, if we follow this path a plus b is available, but if we follow this path it is not, so it is partially available. Anticipability computed along all paths starting

form p to the end node with no changes to operands. There is a computation of the expression coming from p to the end node that is anticipability, partial anticipability is computed along at least one path from. So partial availability was computed along at least one path to p and partial anticipability is computed along at least one path form p to the end node.

(Refer Slide Time: 49:01)
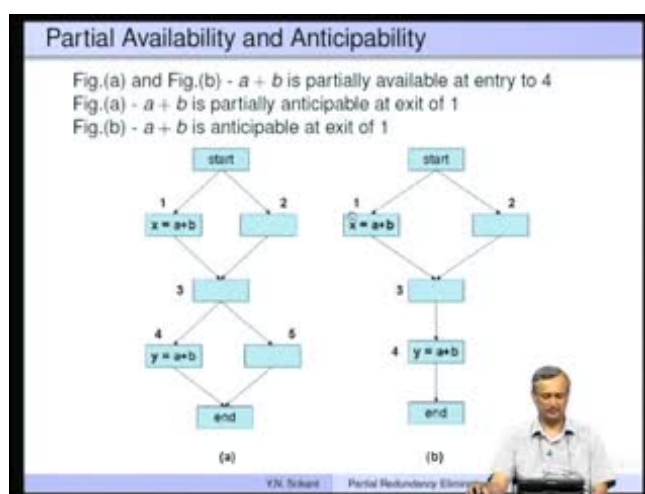
In figure a, a plus b is partially anticipable at the exit of 1. At the exit of 1, why is it partially anticipable? If we follow this path a plus b is indeed computed, so it is anticipable. If we follow this path, a plus b is not computed and therefore, it is not anticipable. So in total, from here to this at this point we say partially anticipable.

Figure b, a plus b is anticipable at the exit of 1, so this point we have only one path to take to the end node, so y is equal to a plus b computes a plus b and therefore, a plus b is anticipable at this point, it is also anticipable at this point, this point and so on. Similarly, at this point again, a plus b at the exit of 2, a plus b is indeed partially anticipable. If you take the entry to one, a plus b is anticipable, because a plus b is directly computed at this point, it does not matter which path we take.

What is safety? It is actually defined as either available or anticipable. In other words, the expression value should arrive from the top or the expression should be computed later; one of these must happen. What it indicates is there is going to be a computation of the expression on a path passing through a point, if that point is safe. That is why safe points are very important for us; it ensures us that no new computations will be introduced.

Safe partial availability; all points on the path of availability from the last computation of the expression to p are safe; in other words, we are looking at the last computation of the expression and then the point p, from that last computation to p the points must be safe. For example, in figure b, a plus b is safe partially available at entry to 4, so here is the entry to 4, here is the last computation of a plus B. We are looking at this path, the dark circle implies that all these points are safe, what does safety mean? It is either available or anticipable. Here at this point, it is available, because a plus b is computed right here, at this point availability is not true, because a plus b is not available from this path, but it is anticipable, so anticipability is true. Therefore, the point is safe, this is also safe, because anticipability is true even though availability is not true and this point is also safe because anticipability is true. All these points are safe from the last computation of a plus b to this point; therefore this particular expression is safe, partially available entry to 4.
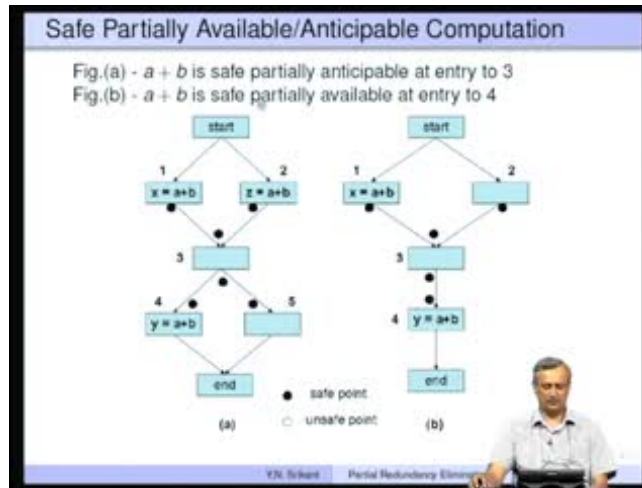
(Refer Slide Time: 52:14)

What is safe partial anticipability? All points on the path of anticipability from p to the first computation of the expression are safe, so a plus b is safe partially anticipable at entry to 3. Look at entry to three, so these are the various points which are all safe. Why is this point safe? Because of availability, again in this path the expression a plus b is available at this path; at this point expression a plus b is available and so on and so forth. At entry to 3, we have availability true, even though anticipability is not true, this is a safe point and anticipability is partial. From here to here it is anticipable, but along this path it is not anticipable, so partially anticipable but safe; so safe partially anticipable.

Comment [s1]: number
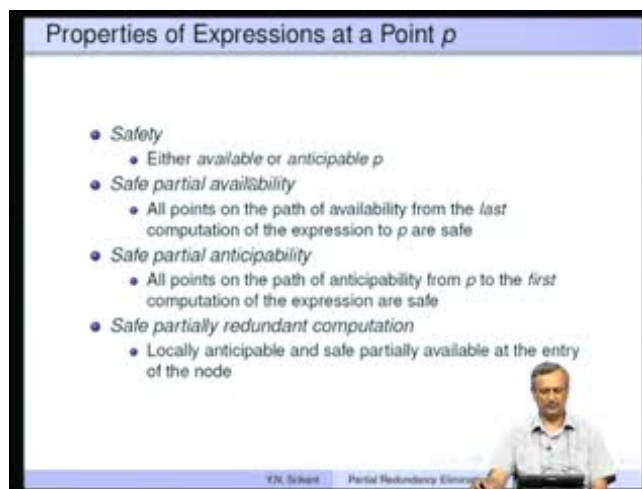
Safe partially redundant computation, is locally anticipable and safe partially available at the entry of the node. Let us understand this, so here it is not safe partially available at entry to 4, so this is an example to show that. We will discuss this particular example in detail in the next lecture with more explanation, thank you.