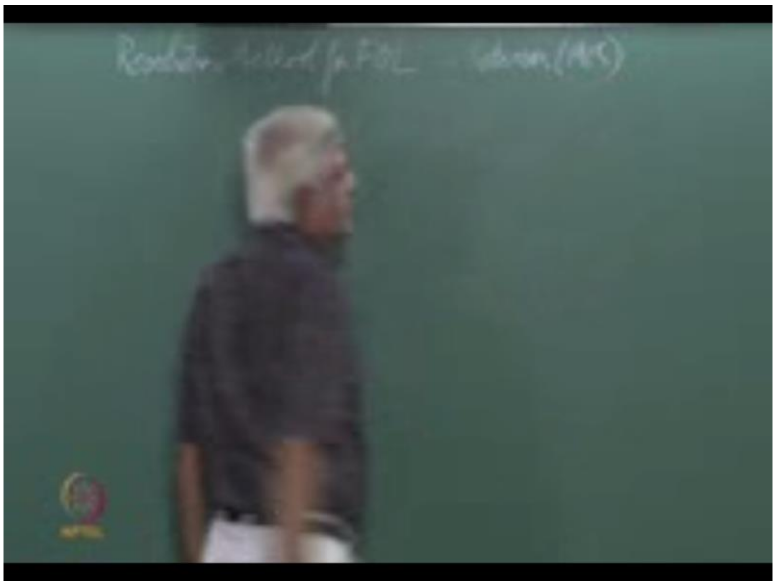
Artificial Intelligence
Prof. Deepak Khemani
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 48
Resolution for FOL

(Refer Slide Time: 00:15)

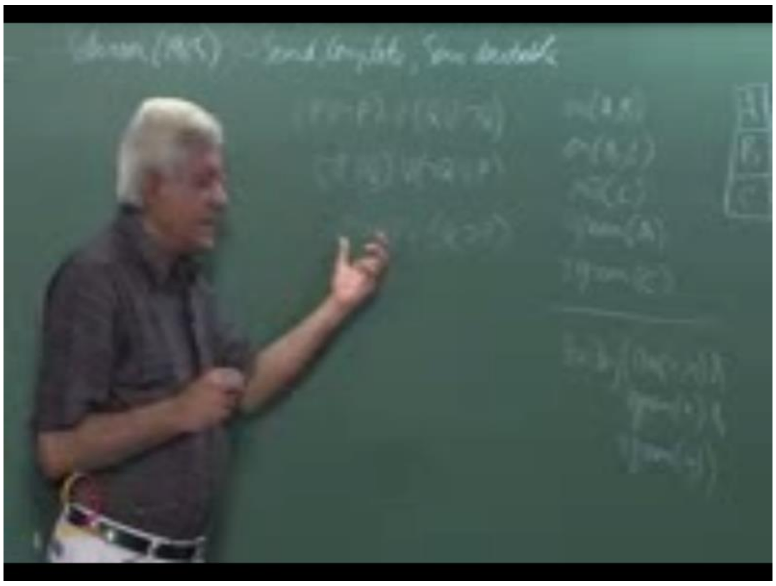


So, we meet for the last time semester and we want to look at the resolution method for F O L essentially. So, as my might have mentioned earlier it was the method which was introduced by Alan Robinson in 1965 and since then it has been used extensively in theorem proving. So, things like for example, the proof for last theorem had extensive support from programs which would based on resolution method eventually. And it is a complete method for a proving the theorem or proving something which uses only one rule of inference which is the resolution rule and it does not mean any other essentially. Now, talking about generally about logic as we said that logic is basically a system in which you define a language. And then define some rules of inference in that language define a meaning of sentence in that language and then talk about soundness and completeness and as you define more and more expressive languages. So, we have seen only one step we have moved from professional logic to personal logic in which we said that we can talk about variables and quantifies. Our variables it was shown by godel in his theorem which was called Godel’s completeness theorem that first of the logic is sound and complete essentially which means you can always device of a certain logic system which is sound and which is complete.

And by complete we mean anything which is entailed by a set of statements can be derived using that machinery that we have building essentially. It terms out that that F O L is sound complete, but it is only semi decidable. And what this means is that in particular it the way it applies to F O L is as follows that if you gave a true statement or statement which is entailed to the system. And ask the system to prove it then there exists a proof for that statement in that system essentially. And you can always devise a strategy and you can imagine the strategy something like breath first search that always you know find the nearest inference first and eventually. You will go further and further away and eventually you will find the proof. So, you can always devise a strategy to for finding a proof, but if you give a statement which is not true then the system in your halt. Because there is no proof and the system my never come out saying that there is no proof now in the case of proportional logic, because we have only dealing with a countably, countable set of propositions when usually it is finite.

We can always say that at least for finites set of propositions we can always say that there is no proof, because we try all combinations and eventually say that there is no way that this can this statement can be true. So, even if it is false you can come out and say false in the case of first of logic we cannot come out and say that yes statement is false. You can only keep trying essentially which falls down to saying that you are program can get in to an infinite loop essentially. And which is not of course, surprising that we have written all of us have written programs which get into infinite loop. Of course, most of our programs get it be infinite loop imperative languages, because we have written it a long a long loop or long exist criteria in logic programming or prolog. We do not control the flow of execution we simply stay it what is to be done. Of course, we do control it in the order in which we write statements, because there we observed prolog does depth first search essentially. Top to down and and left to right which means if you write statements in a wrong order it is possible we could get it into a infinite loop. But even if you write statements in the correct order if the statement that you trying to prove is not true it can still get into an infinite loop essentially.

(Refer Slide Time: 05:11)



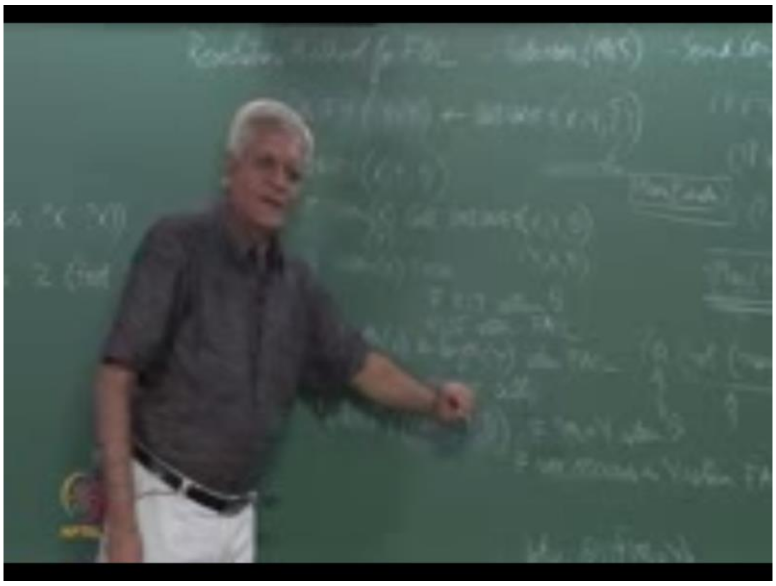
So, today, we will we will go back to this example that we saw and so on a b on b c. And let us on table c and being a supposing given to you this is a set of facts given to you. And you want to show that there exists an x there exists a y such that on x y and being x and not green y. This is a goal that you want to show to be true and that is the facts given to you now you can see that forward chaining backward chaining does not make sense here. Because in your in your data base or knowledge base there are no implication statements both forward chaining and backward chaining move set of allow you to move across simplification statement. So, if you have alpha implies beta then if you have alpha then you can say yes beta is there. Or if you have the goal beta you can say the goal alpha can be a goal, but they are no implication statements here given the set of facts and we are also show this here. So, you can imagine that forward chaining and backward chaining does not work at all essentially.

So, you was of course, convenience your self is that this statement is true and as human beings. We might use one technique which is call looking at different cases of proof by cases we could say take the case when x is a and y is b or look at b. So, there are 2 cases that either b is green or b is not green if b is green then x is equal to b and y is equal to c and b is on c and b is green and y is not b. If b is not green then you can say x is equal to a and y is equal to b and a is green and b is not green. So, whether b is green whether b is not green you can show that in either or the 2 cases this statement is true. And if you further say that this is the only possibility that one of them is true then you can argue that yes this statement must be true essentially. So, this sequence which has slipped in between if you argue that one only one of them is true is essentially a part of classical logic in classical logic every statement is either true or false and there is nothing in between.

So, it is called law of excluded middle essentially which some people object to, because of statements like this, but I can say that p or not p is true or q or not q is true. So, this is a true statement why because of the law of excluded middle either p must true or not p must be true. So, in which case this this and this true and true and both are true essentially I can shuffle this and write it as not p or q or not q or p. And then which I can write as p implies q or q implies p. So, this is true for any 2 statements p and q whether had any language personal or professional or high order language. It does not matter and if you treat implication as implies in the causal sense then people have difficulty with such statement. Because just imagine that p stands for the earth is flat and q stands for the moon green then you are saying that there is casual connection between these 2 things that the earth is flat. And the moon is green that either the earth is flat implies that the moon is green or the fact that moon is green implies that the earth is flat essentially.

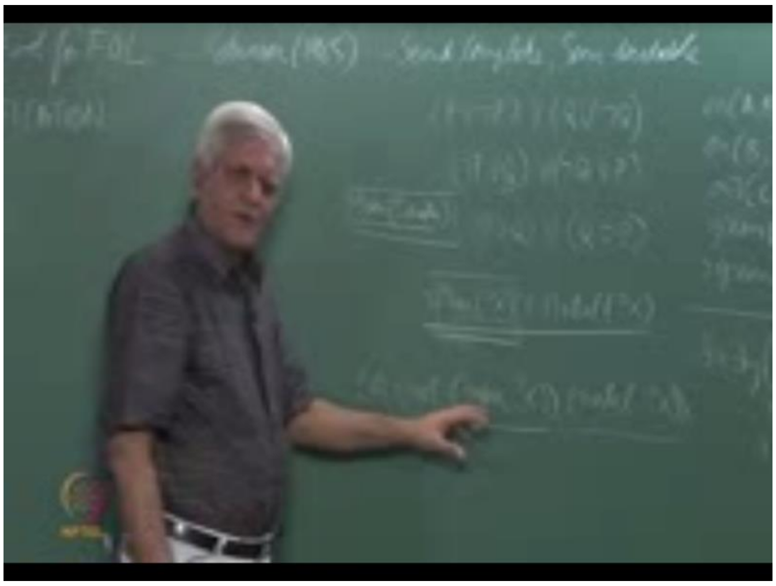
Now, obviously there is no casual connection between these 2 statements you take any 2 statements p and q. And this statements is always true essentially so which is I sometimes logicians tend to distinguish between logics with capture cause and relationship. And we will not go in to that but rather they would see that instead reading an implication here. You must read it like this when you say p implies q it is easier to read it as this without getting worked up about it says that either p is false or q is true. That is all you are saying essentially, but when we read this an implication we have the sense of being a causal relationship, which is not really the case essentially. Now, let us get back to the resolution method.

(Refer Slide Time: 10:52)



And before we do that we want to look at this algorithm called unification algorithm which is a very famous algorithm. What unification algorithm does is that it takes 2 patterns while use a more generic term then formulas and tries to unify them which means tries to find the substitution which should make them same essentially. This patterns are made up of 2 kinds things; one are one is constant and one is variable. And variable is something that you can substitute something for the other and constants are or atoms as some people call it cannot be substituted cannot be changed they have. So, for the sake of simplicity we will adopt slightly different notation.

(Refer Slide Time: 11:46)



Supposing I have this statement man x or, so let us say not man x or mortal x. Now, this statement which is in the mathematical language of logic has this particular notation that you have the predicate name. Or it could be the function name in some situations followed by brackets followed by the arguments and then predicates connected using logical connectives essentially. So, you have to distinguish between different kinds of things for the sake this unification algorithm that is assume that we have a uniform notation. And the in that notation what this will look like is a that I will use or here and the notation uses the question mark as a thing for variable. This just for the sake of understanding this algorithm easily you can always adopt the algorithm to this notation, but this is simply easier to use. Because it is a very uniform list like notation which those of you who are use list put be happier to look at essentially. So, I have moved a or sign outside. So, the outer most connective is first and then the inner connective which is not here so not man is written like this so not.

So, this whole thing from this bracket to this bracket is one expression or term if you we just call it a term or a list. So, from here to here is a list which is so the first one is always a connective the first element in the list is always a connective or a predicate name. In this case so here predicate name is differentiated and you know connective. So, the first thing is in the case it is a predicate name man here it is a connective here it is a connective. So, this has got this 2 arguments this whole thing is one thing and this whole thing. So, it is a list of 3 elements so everything is a list of some number of elements and the only thing we need worry about is let either something is a constant or an atom as the list people say or it is a variable essentially So, and the only the point up out that is an atom can only match an atom. So, if I am this with a if I want to match this with man, so what I am trying to do I am trying to do something like resolution. If you recall I have this clause and I have this clause. And I have something here I have and I have the negation of that thing here resolution's rule if you remember always first of all it works in the clause form that that you must express things in c n f like form. And then you have to look for something verses and it is negation of a positive literal and negation negative literal. And in some sense cancel it out essentially and from there you can derive mortal x essentially.

So, which means of course, I will have to match this man x with mortal x with man sonatas which is work the unification algorithm will allow it do it will tell me what values for x will make this true expressions. The same in one case we have just called it as list here essentially. So, to match this with this of course, the predicate name must be the same. So, we are just treating it as a atom here which means it can only match a another atom which is same a man not can only match not or can only match or it is only the variables which can match something else. So, let us assume that we have an arbitrary nested pattern which is express as a list of this kind. And we want to write a general algorithm which will find a unifier for any 2 patterns or a substitution was those 2 patterns which should make it a same this substitution is called a unifier. And that is algorithm that we are looking for unification algorithm. So, the algorithm is called let us say it is called unify I will just sketch it here and you get fill in the details.

So, let us say x y and what is does is it. So, will write in a prolog like fusion this part that is sub unify x y. So, like we do in very very often we write a program; we add a third parameter to make like simpler for us we added an empty list and call it sub unify sub unify what this is going to be is the substitution. So, obviously when we call it when we call we so x y are any search list arbitrary list we start of by supplying an empty substitution. And now we, so x y theta where theta is the substitution we are trying to build what is a substitution? Substitution is a collection variable value essentially it says x equal to x should be substituted by this y should be substituted by this and so on. We should make the 2 patterns same essentially and this algorithm goes to a series of cases. I will just list out the salient once here; obviously x and y are the same whether they have atom or whether they have a list then you do not need to do anything they already unified essentially. So, some of main cases are follows.

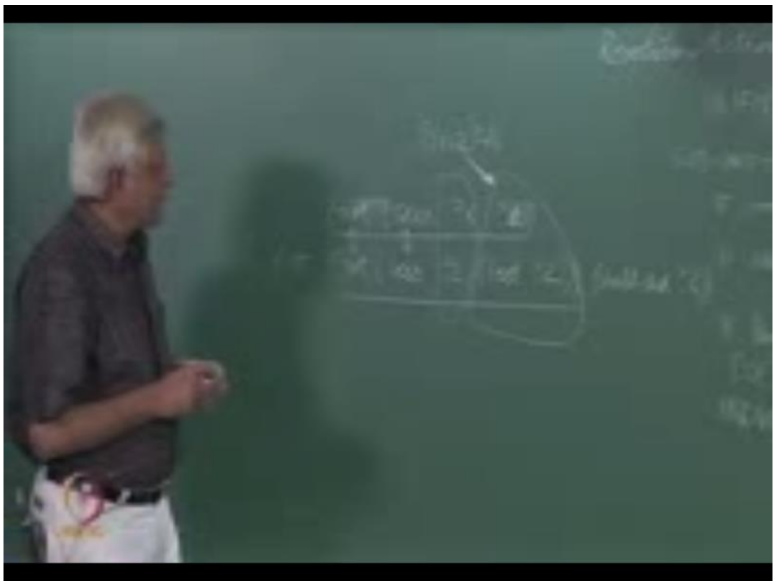
So, these are cases so I just write it as f So, let us understand this to mean that x is a variable then call another function called unify x y theta. So, if one of them is a variable then it is a candidate for being substituted by something else and that this function unify will do likewise if y is a variable you can also call var unify with y x theta somewhere. So, this order will not be the perfect order I am writing here if atom x then if x equal to why return theta else. So, if x is an atom if the argument that we are trying to unify. So, this going to be a program, so this x and y initially will be such bigger patterns but eventually when we build on into list at some point they will either a variable or an atom. So, if it is a variable then we just I call it call var unify with others argument if it is an atom we check whether it is matching or not essentially. So, if it is not a variable then if it is not an atom it must be a list then we say if length x not equal to length it is a list.

So, we can compute it is length return fail if the 2 list are of unequal length then you can never make them match else make recursive cause. So, make I just write it like this appropriate recursive calls what does it mean that? For example, if this is one of my argument this is x and this x has a list of 3 elements the first element is constant. Or the second element is this list which is this whole thing and the third element is this other list. So, if I have another list of 3 elements I can try to match that and then I will make recursive calls once this with this then another with this and another with this. So, I implemental build a substitution that leads with ask with the task of writing the var unify which is really building the substitution I might have skipped one of the small details does not matter x. So, when we make a call to var unify we know that x is a variable. The other thing would be a list could be a something else. So, let us say this is a variable and y and theta we know that the first thing is a variable essentially we have to do a few checks. So, essentially what we really want do is to say add var egual to y to my theta.

Add one more substitution, but before doing that I want to do. So, what I what I want do?

I want say return bete union var equal to y. I want to basically add one more substitution which is the call I am making, but is I may allowed to do that I have do a couple of checks first first is if var is already equal to y. Then you can just return theta you do not have to do anything else it could be the case. For example, I am comparing well these are not variables does not matter it could that they are same variables then you just return theta then if variable occurs return fail. So, if the variable happens to occur in y then you return fail. So, let me use an example to illustrate by this is needed and this example is from book by I can make them what which disturbs it in this notation.

(Refer Slide Time: 24:31)



So, the example is as follows naught c's x. So, let us say c is stands for the predicate and the meaning of sees is that the first argument can see the second argument. So, for example, I see you or things like that, and this is a universally quantified statement. It is saying there for all x let us assume that x is people x cannot see x which means one cannot see oneself essentially. So, let us assume that that is a true statement and I have a rule which I am let us say I am doing forward chaining. And the rule is as follows again I am writing it in this new notation where instead of writing the implication sign I will write an if here then the antecedent and then the consequent. So, instead of saying this implies the consequent I am writing it in this list like notation which may them what the use which is quite nice notation to use easy to process so this rule says.

So, it always worry about the number of brackets in this set of thing. So, what does this tool say? It is a universally quantified statement how would read it in English. So, feet is a function remember that in first of the logic the argument to predicate can only be terms. And terms are either variables or art or constant or functions feet of z is function. So, let us say feet was stands for z's feet. So, this is saying that is anyone cannot see their feet they should diet essentially this may be a true statement. Of course, know the question is shall should this rule and this these 2 rules can we apply forward chaining here to that everyone should diet not given these facts right. So, let us say how that particular statement that we are talking about if variable occurs y then return failure comes to our. So, we are trying to unify this with this remember the antecedent should match.

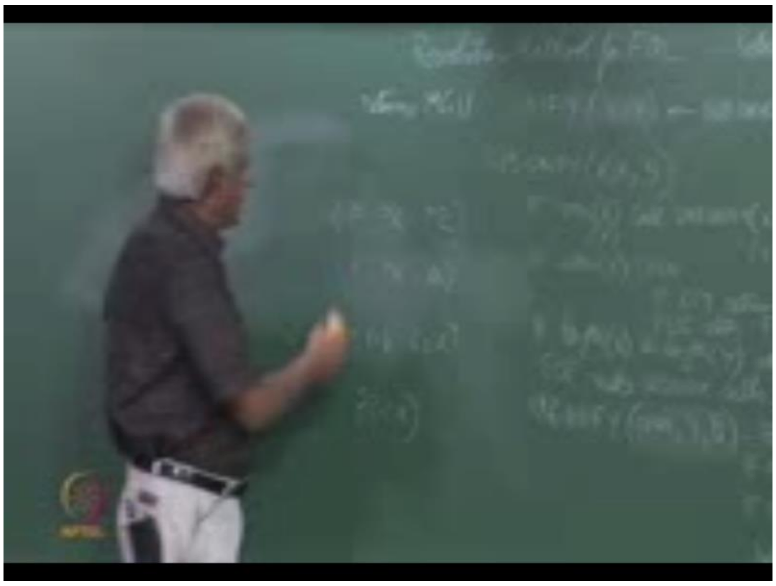
Now, we will make recursive cause first you see that this is a list of 2 arguments then we will make 2 recursive cause one with the first argument. Then one with the second argument then the first call this is an atom and this matches this. So, that is fine so the first recursive call will work and it will not change theta at all the second recursive call has list of 3 elements and this also as a list of 3 elements. So, that is fine first we will make a call with sees and sees here in the first call this case will come if if this is an atom this is same atom then do nothing in the second call we will do this. So, our algorithm will say x is a variable. So, I will call var unify with x and x and z and then in the last statement which I will reached there I will say add x is equal to z I am not writing the question mark here to theta. So, this will go into theta so this is if you want to may these 2 patterns same substitute for x the value z. And you will have theta now you have x is already been put as z here. So, let so whether you do this here or whether you do this theta both ways it works in fact the others other situation is if var as the value in theta while you let us call it z in theta.

Then if variable already has the value in theta then call sub unify with that value essentially.So, in this example we already have a value z in theta x equal to z. So, if we are not change this x that clause will and we will call with x n. So, we so that recursive call that sub unify call in this line here would be with the value of z with z and feet of z essentially. So, let us assume that we have already made this z which is this case where this is happening now you are trying to unify this x which is a variable with feet of. So, this is become z now, so let me put an arrow here and show that this is become z.

Because we have substituted x with z now, we have making a call of a variable z with a list which is feet of z which is the y.

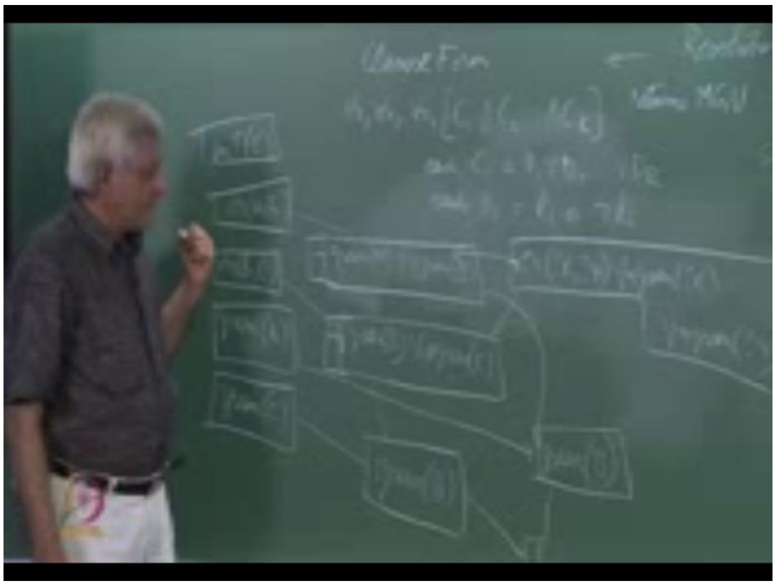
So, this x in this statement here sorry this is well in this statement and this is y and this statement says that if var occurs in y then return fail essentially. So, in this example z occurs in this pattern or list. So, the algorithm should return failure in that we cannot unify this essentially which is good for us because otherwise all of us would have have to diet. So, this does not apply and this particular clause here is meant to catch exactly these kind of a things essentially. So, you can never unify z with feet of z you know if you substitute feet of z for z then you will have to substitute feet of z for this z also then the z inside then the z inside then the z inside. So, it know does not make sense. So, this unification algorithm.

(Refer Slide Time: 32:20)



And what it returns is the theta which is the most general unifier which can may the 2 patterns same. So, for example, if I have a statement p x z and another statement p x let us say constant a. Then you can see that I can have one unifier which is x is equal to b and z equal to a that is a unifier I am not saying that this algorithm will find this that a unifier. So, that is one unifier and there is another unifier which is simply z equal to a this unifier more general then this unifier, Because it does less amount of substitution I mean anything that this does this also does, but there is something which this does which this does not do. So, this is called more general then this go in to details we will accept that there is a partial order of unifiers. And there is something called the most general unifier which is called m g u and this algorithm essentially returns the most general unifier. So, again without going into details we just accept the fact that it is desirable to find the most general unifier. And the reason for that is that you can make the most general inferences from which you can always derive most specific inferences using the universal rule essentially. So, I am we do not have to go in to details over that, but it basically the it is son ate algorithm which is really popular and theorem proving and we use it all the time. So, let us address this problem let us see an how the resolution method solve this problem.

(Refer Slide Time: 34:55)



So, to convert to solve a problem with resolution method you have to convert it in to clause form. And a clause form is a form which looks like follow as follows at there is some number of universal quantifiers x 1 all x 2 all x n. Then there is a set of clauses c one and c 2 and c k such a form such a form of a formula is clause form when each c I is d one or d 2 or d r and each d I is equal to l I or negation of l I. So, of course, they inside part you will recognize as a conjunctive normal form a set of clauses which are joined by an and each clause is basically some something. And each of those things is either a literal which means an atomic statement or the negation of an atomic statement. So, you have push the negation side all the way inside and you have removed or thrown way existential quantifiers likely in our example we do not really have existential quantifiers. Or at least we will see in a moment that we do not have, but we discussed earlier how to handle existential quantifiers by using the solemn functions. And the solemn constants that can be done and then re arranging any formula into c n f is something that I am sure you have studied how to do that.

And then you move the universal quantifiers outside one they are together outside you can just threw it away use quantifier form which is what we had doing here essentially. So, what is it, so if you recall the deduction theorem that we had talked about earlier it said that to show this follows on this you are equivalently showing that this and this entails this. And if you want use the resolution method you will recall that to use the resolution method you must take the conclusion and take it is negation and add it as a clause to your system essentially. So, this is already in clause form all we need do is to convert this into take it is negation. So, what it is negation of that? The negation of that put on negation sign outside here I will have to push the negation sign inside, because I have to convert it into this clause form. So, this will become for all x for all y not one x y or not green x or green y. So, once this negation go inside then it go inside the and sign convert it to an or remember that we have to push to the inner most place. So, this will become not on this will become not green this negation negation cancel then this will become green y.

So, let me write this here on x y or green x or not green y. So, this is one clause express it in the implicit quantifier from and those are other things given to us which is on a b and on b c. So, let us forget on table c that is not useful for us I mean we can write it, but it does not help us green a and that is say it is on table c there any way we do not really need that. So, we has this clauses and we want to what is it we want to show that can we derive the null clause from this. And we are going to use the unification algorithm along the way our example is so simple. That we do not have to really use a very complicated, because it is very simple you can match it. So, let us just try this form this and this I substitute x equal to a and y is equal to b I get green a or not green b. So, I am not stated the resolution step or the rule for first of the logic, but you can see that it is very similar to modified you do you apply the substitution and in the resolving you have substitution already applied essentially.

So, because I am saying x equal to a and y is equal to b then this becomes a this becomes b and I get this essentially then from this. And this I can get similarly green b or not green c is that correct this should be negation here right and these should not be negation and this should be negation y. So, all 3 are long way so negation on x y negation green x yes seen that thing there and green y essentially what we get here is not green a and green b and not green b or green c. So, is that correct know and then from this and this I can get not green b and from this one and this one if you can keep track of arrows I can get green b and from this and this. So, just I repeat from on a b and that negated goals. So, remember this is a negated goal that we have added we get not green a or green b then from on b c and the negated goal we get not green b or green c, but here we have side not green c.

So, when we resolve this with this we get not green b only this remains when we resolve this with this this is green a and this not green a. So, you get green b and then we not green b or green b and from that we get the null clause essentially. So, you can see that there is simple proof using the resolution method and if look at a proof carefully you can see that it is trying to in some sense say at the same time that if this formula is to be unsatisfiable. If this whole set of formulas which means only this formula, because this is accepted to be true; this is the premises given to us. If this is to be false then it entails same time b must green and b must not be green and of course, is a contradiction. So, as we discussed earlier the resolution method is like a proof for contradiction. So, if you remember when we talked about forward chaining or backward chaining. There is no way you can move from this set of data to this conclusion essentially the conclusion holds that one the exists a block on another block.

So, that the block top is green that the block below is not green it is it is not even initially clear, but it is true. But we cannot derive it using first forward chaining or back ward chaining, but in resolution method there is a very simple small proof for doing that. So, in fact, this procedure by Alan Robinson was a big in logical reasoning automatic theorem proving. And nowadays, automatic theorem proving is an, in many different place for many different applications essentially. And the heart of this is resolution method essentially which is a sound and complete method for first of the logic essentially. So, I think we should stop here with this we will end this course I must say I enjoyed teaching the class. And I hope some of you at least enjoyed the course I think so.