# Artificial Intelligence: First Order Logic: Forward Chaining

**Prof. Deepak Khemani**

**Department of Computer Science and Engineering**

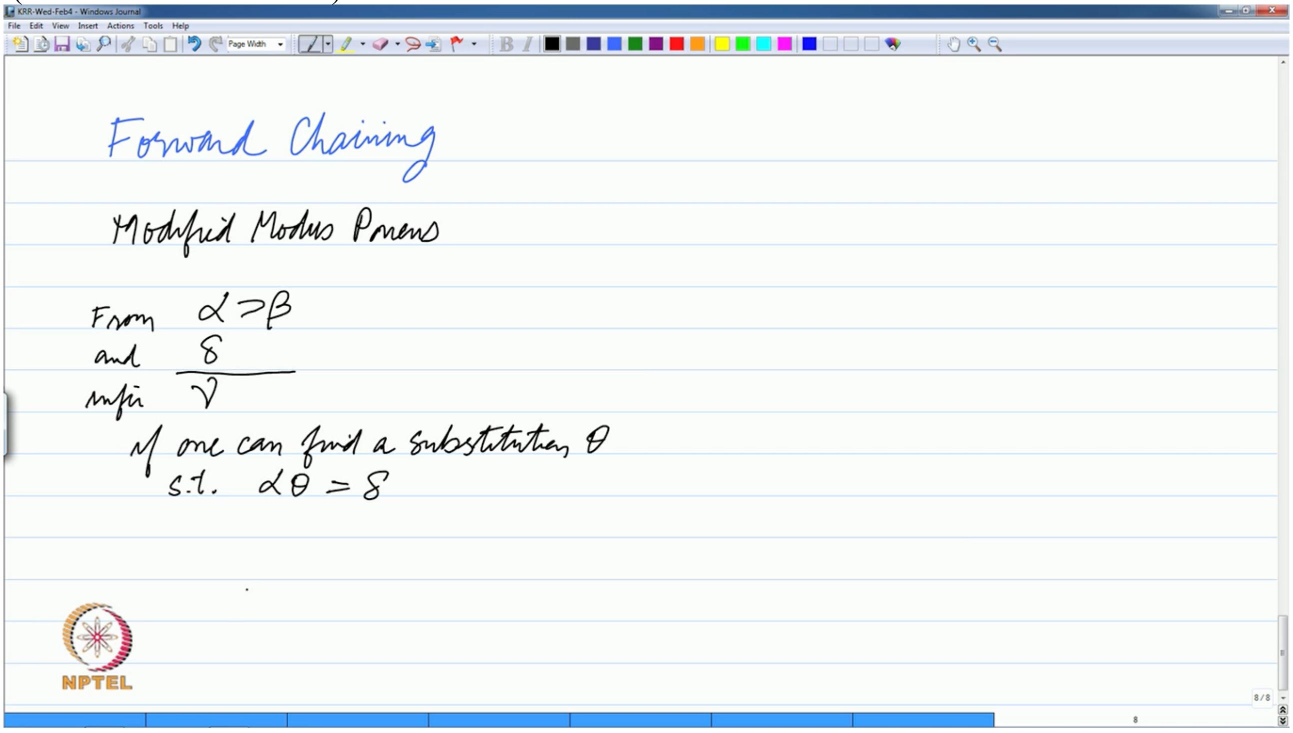
**Indian Institute of Technology, Madras**

**Module – 03**

**Lecture - 05**

Okay so we are in the process of generating proofs of first order logic and in particular we are looking at this process of forward chaining. We started by looking at this rule called modified modus ponens so let's begin with that. Let's define it formally. So what we are saying here is that if we are given a formula alpha implies beta which is an implicit quantifier form. For the moment we are working only with universally quantified statements and if you are given a formula delta, then you can infer a formula let's say gamma. So from this and this we infer gamma if one can find a substitution theta such that alpha theta is equal to delta. In this notation what we are saying is that we have applied the substitution theta to the formula alpha.

(Refer Slide Time: 02:08)



Forward Chaining

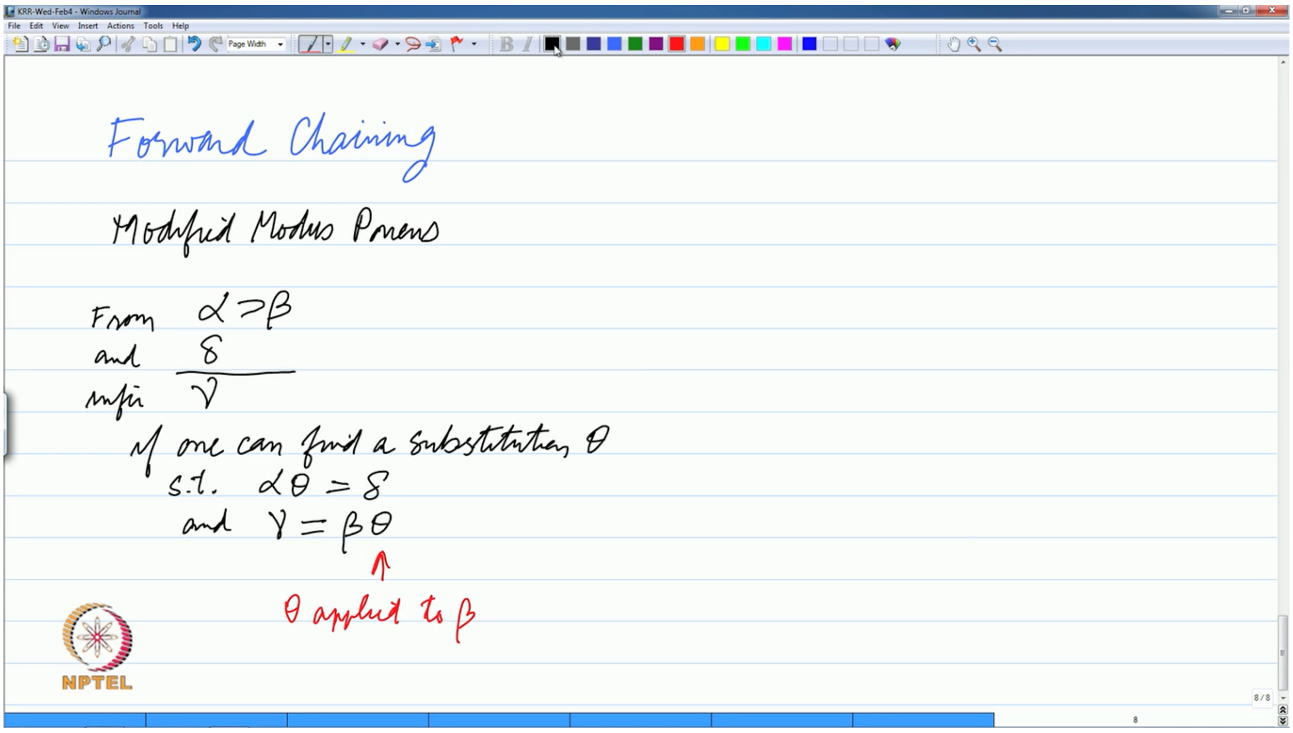
Modified Modus Ponens

From $\alpha \supset \beta$
and $\delta$
infer $\gamma$

if one can find a substitution $\theta$
s.t. $\alpha\theta = \delta$

So wherever the variables that are talked about in theta are present in alpha we have substituted them with whatever the substitution specifies essentially. So if we can find a substitution theta such that when we apply to alpha, the formula alpha theta becomes equal to this formula delta, then we are in the position to apply modus ponens and what do we infer? We infer a formula gamma which is obtained by applying the same substitution to beta. So this is theta applied to beta. In the first case it is theta applied to alpha

(Refer Slide Time: 03:00)


## Forward Chaining

Modified Modus Ponens

From $\alpha \supset \beta$

and $\delta$

infer $\gamma$

if one can find a substitution $\theta$

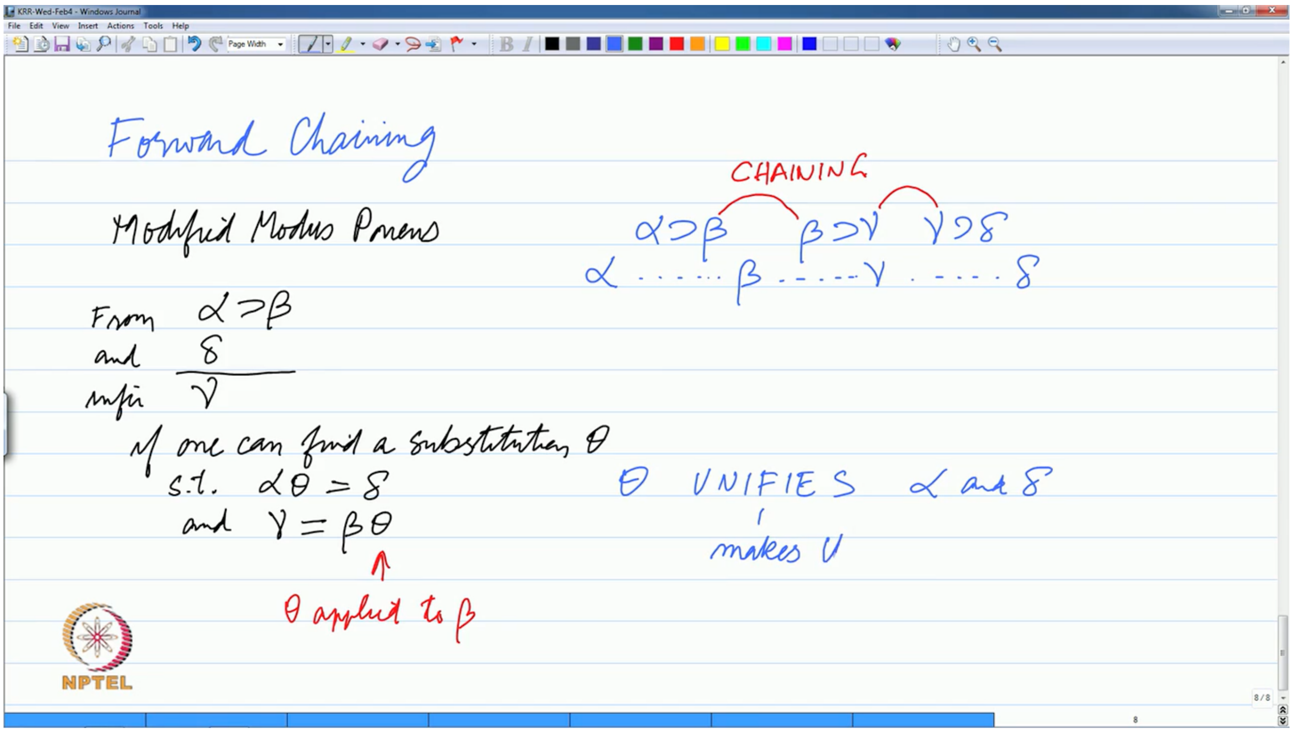
s.t. $\alpha\theta = \delta$

and $\gamma = \beta\theta$

$\theta$ applied to $\beta$

so this is the modified modus ponens rule. It actually combines two steps of first order logic, one is the step of universal instantiation and the second is the step of modus ponens. It does everything in one step but the thing is you have to find this theta on the way. Why is it called chaining? Because let’s say you are given a fact alpha and you have a rule which let’s say is in the same form alpha implies beta so now you can infer from this beta by applying modus ponens. And if you have a rule of the form beta implies gamma then you can infer from this gamma. If you have a rule of the form let’s say gamma implies delta, then you can infer delta essentially. So what are we doing we are in some sense changing rules together and since we doing it in the forward direction that we are moving from what is given to us, what the facts are and moving towards the conclusions, this process is called forward chaining. And we will start looking at algorithms of forward chaining. This theta unifies in this example alpha and delta so if you apply in this case of course delta may not have had the variable in the example that we were looking at but it is also possible that delta may also have had the variables. Theta is a substitution which unifies two or more formulae, in this case there are two formulae. And when we say unifies we say make identical.

(Refer Slide Time: 5:33)



## Forward Chaining

Modified Modus Ponens

From $\alpha \supset \beta$
and $\delta$
infer $\gamma$

if one can find a substitution $\theta$
s.t. $\alpha\theta = \delta$
and $\gamma = \beta\theta$

$\theta$ applied to $\beta$

CHAINING

$\alpha \supset \beta \quad \beta \supset \gamma \quad \gamma \supset \delta$

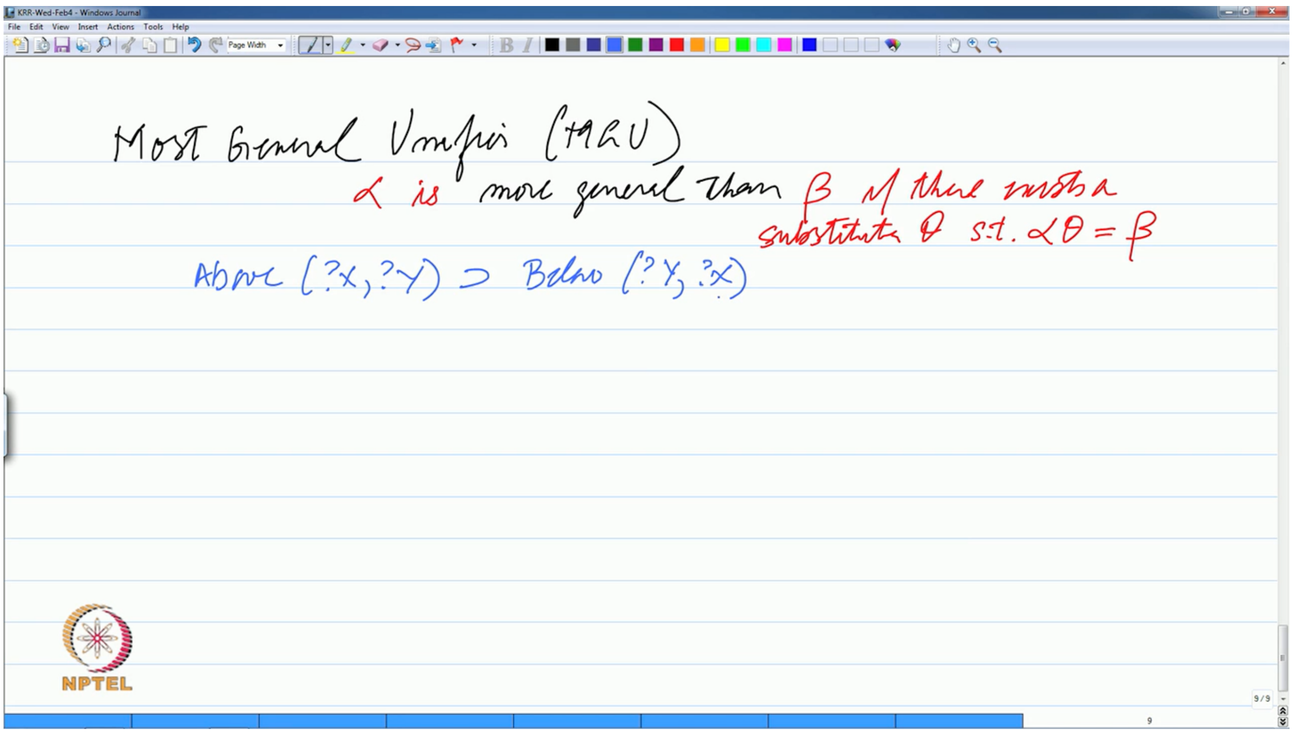
$\alpha \ldots\ldots \beta \ldots\ldots \gamma \ldots\ldots \delta$

$\theta$ UNIFIES $\alpha$ and $\delta$

makes

And the algorithm which we look at today is the algorithm which will find this theta essentially. So theta is called a unifier or a substitution. Basically a unifier and a substitution are the same things essentially. So let's first talk of what are we going to look for in this algorithm. We look for something which is called the most general unifier often known as MGU. So let's understand first what do we mean by most general unifier. There is a more general than relation between formulae. We say that alpha is more general than beta if there exists a substitution or a unifier theta such that you apply theta to alpha and you get beta. So if you are taking a formula which has a universally quantified variable and you are substituting something for that variable, you get the new formula beta. Now obviously this alpha is more general than beta because it is talking about a universally quantified variable. But in the example we were looking at we are substituting Man x with Man Socrates so obviously Man x is more general than Man Socrates. And this is the formal definition of more general than essentially.

So let's look at an example to understand what are we talking about most general unifiers and why are we interested in them essentially. So let us say that we are given two facts, the first fact is that
(Refer Slide Time: 09:15)

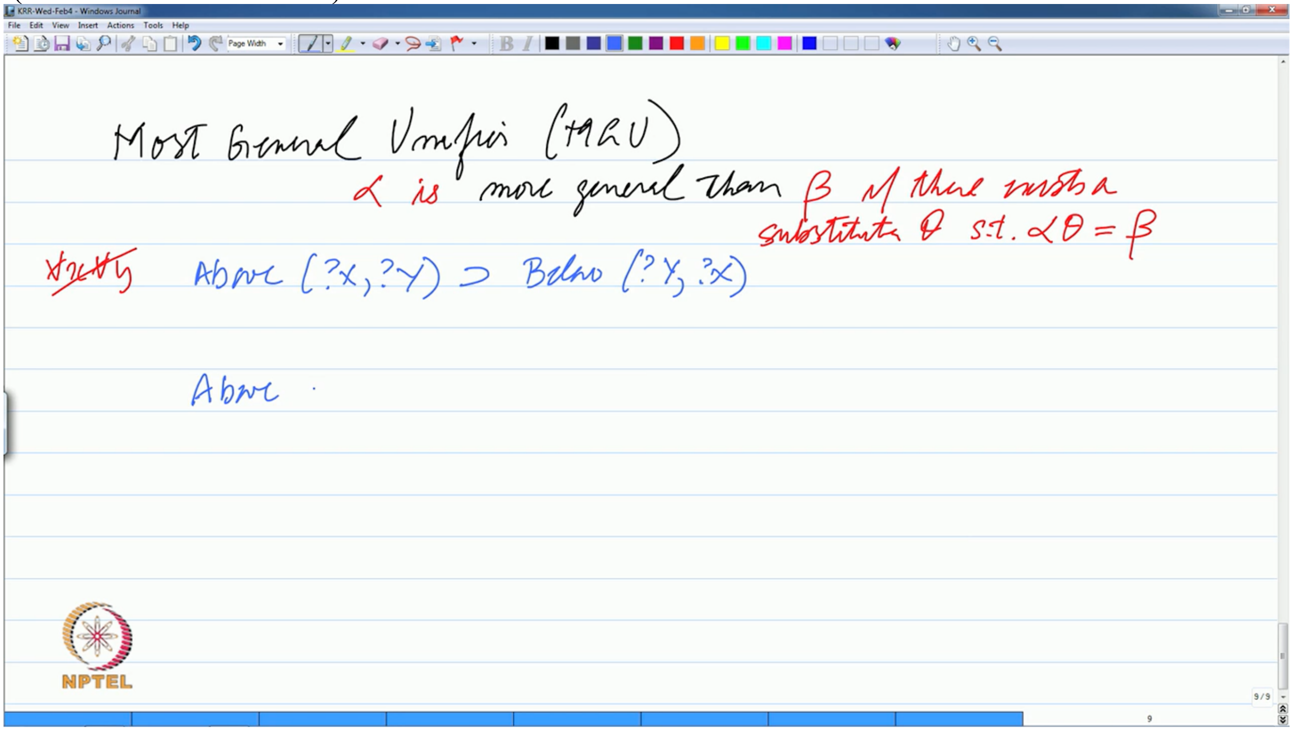

Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ if there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

Above (?x, ?y) ⊃ Below (?y, ?x)

So the first thing of course you should get used to is to read this statement as a universally quantified statement. Both x and y are universally quantified variables which means implicitly there is for all x for all y sitting out there. But since we are using the implicit quantifier form we are not writing that actually. But you should read this as for all x for all y and Above and Below are two variables which between us we will understand them to be the fact that Above x y means x is above y and Below y x means y is below x. So this statement is essentially saying that for all x and for all y if x is above y, it means y is below x.

(Refer Slide Time: 10:16)


Most General Unifier (MGU)

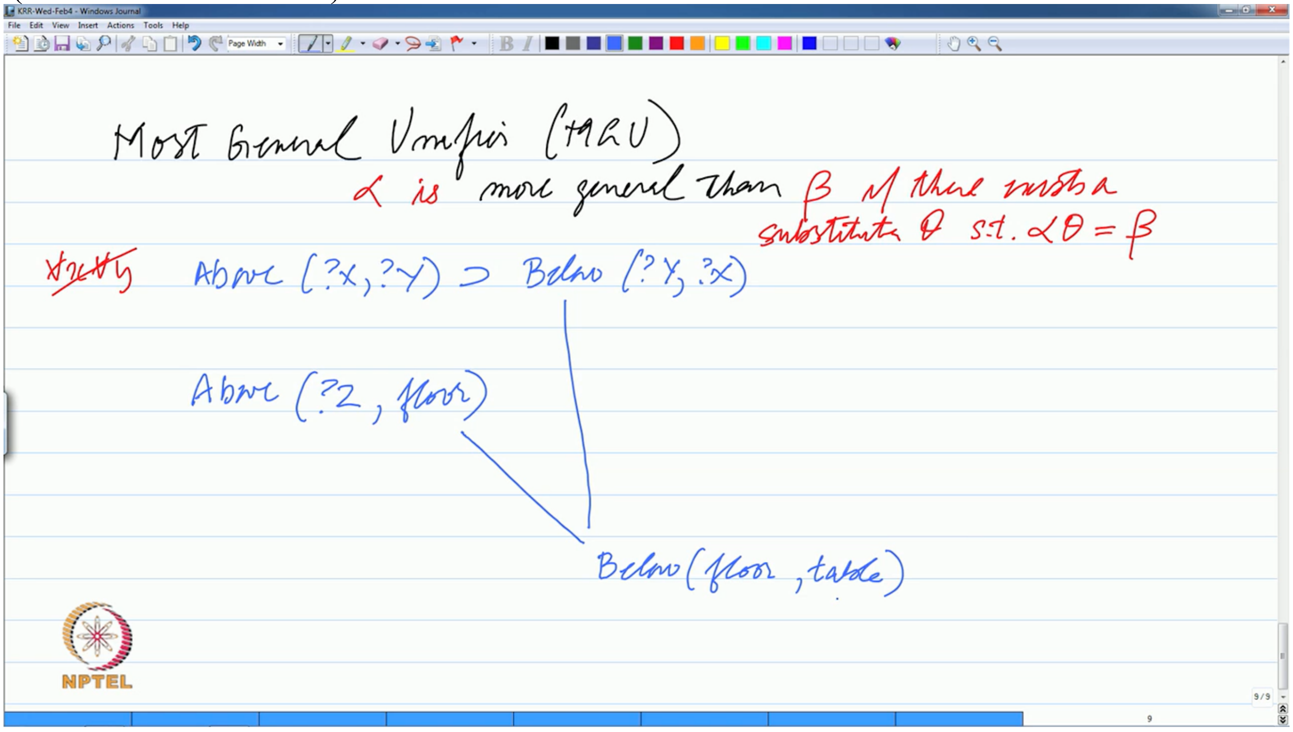
$\alpha$ is more general than $\beta$ if there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

$\forall x \forall y$ Above (?x, ?y) ⊃ Below (?y, ?x)

Above

And the other statement that is given to us is let us say Above z floor. Again this is a universally quantified statement, it is saying that everything is above the floor essentially. Now notice that we we write floor we didn't put a question mark so it's not a variable, whereas with z we out a question mark so it's a variable. So that's a convention that we will follow essentially. Now you can see that here is a case where we can apply modus ponens and we can make an inference that Below floor table.

(Refer Slide Time: 11:16)



Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ iff there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

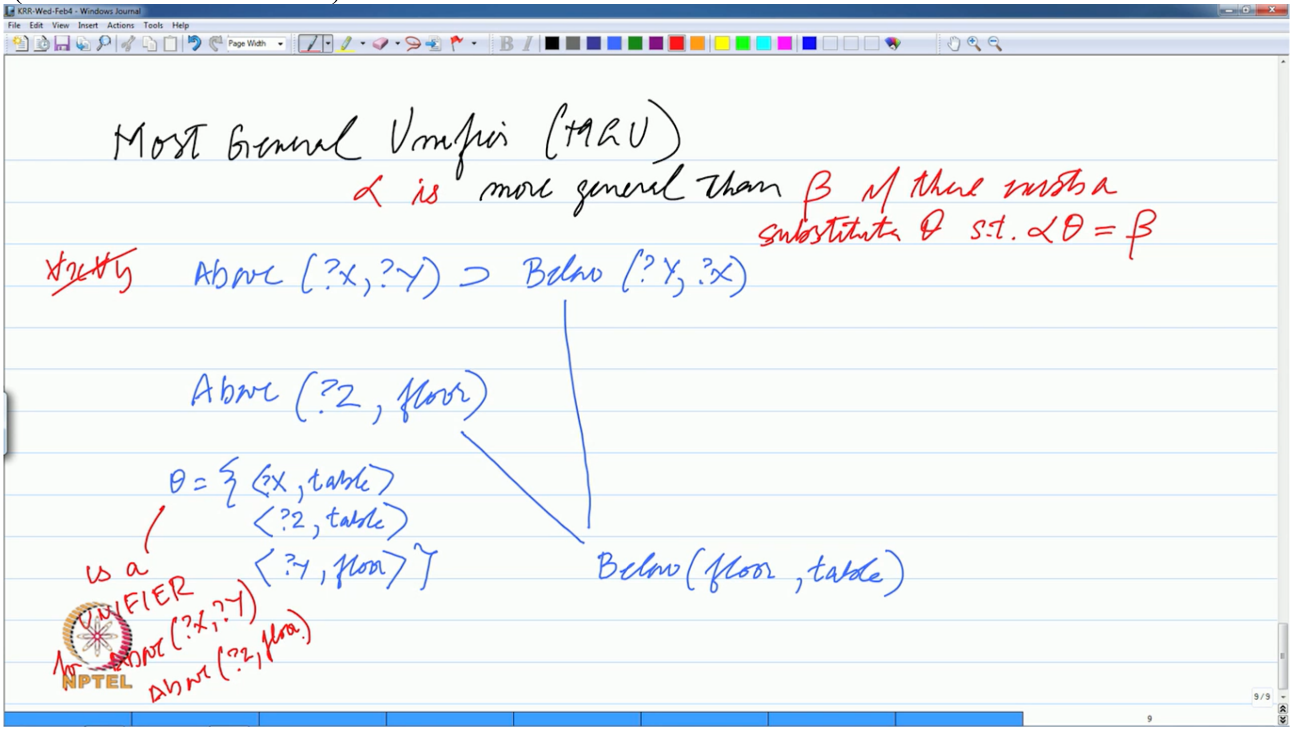
$\forall x \forall y$ Above(?x, ?y) $\supset$ Below(?y, ?x)

Above(?z, floor)

Below(floor, table)

So we can infer, so given these two statements we can infer that the floor is below the table. How do we infer that? We use a substitution theta which says that you must substitute, I will not write a question mark here because it is clear that it's a variable or let's just put it for the sake of consistency.    I can substitute a constant table for x, I am allowed to do that because universal instantiation says you can substitute anything. Likewise, I will substitute for z the word table and I will substitute for y floor. So observe that this substitution is a unifier for these two formulae.

(Refer Slide Time: 12:58)



Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ iff there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

$\forall x \forall y$ Above(?x, ?y) $\supset$ Below(?y, ?x)

Above(?z, floor)

$\theta = \{ \langle ?x, table \rangle \ \langle ?z, table \rangle \ \langle ?y, floor \rangle \}$

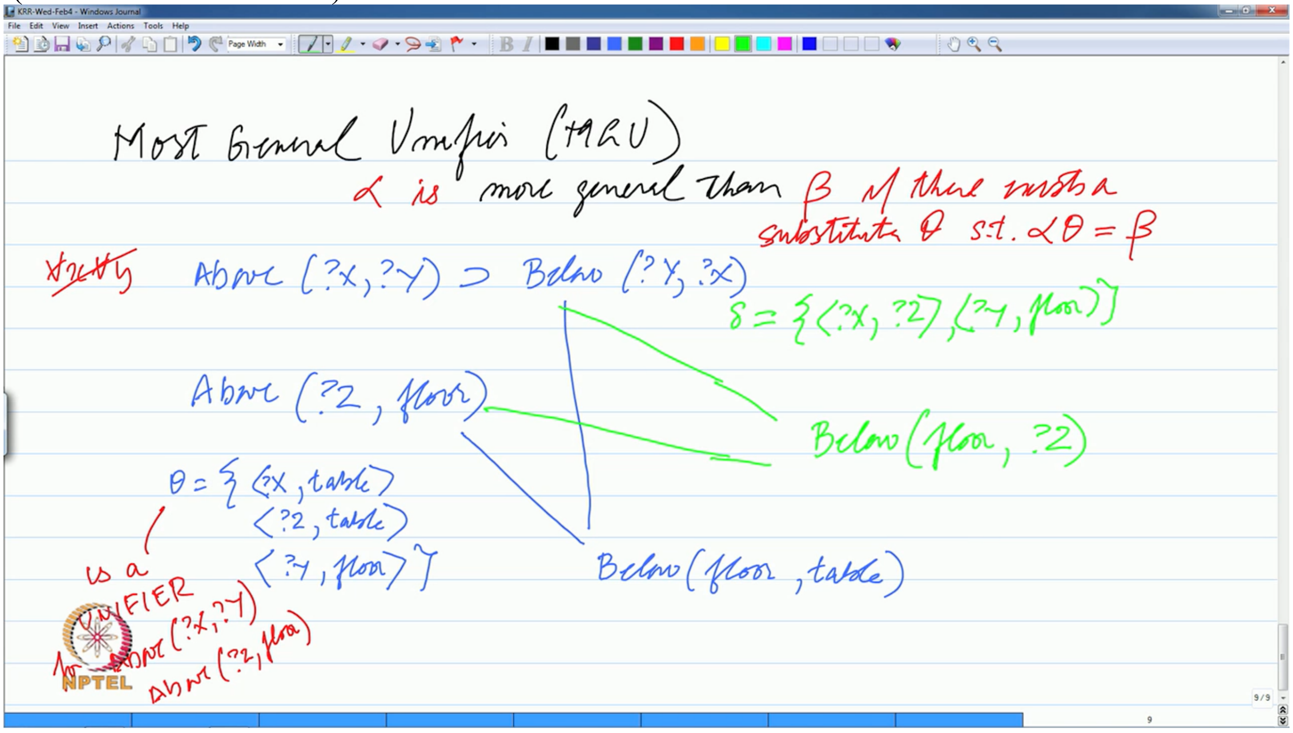
is a UNIFIER for Above(?x, ?y) Above(?z, floor)

Below(floor, table)

so it unifies these formulae. The left side of the implication and the other statement that are given to us. It means it makes them the same. Then we can apply the substitution to the right hand side according to the modified modus ponens rule and for y we will substitute floor and for x we will substitute table and you will infer that the floor is below the table. So it's a perfectly valid inference to make essentially.

But we could have chosen another substitution as well. So let me draw this with a different colour. And this substitution says, let's call it delta. We say that for x I will substitute z and for y i will substitute floor. I am allowed to do that. Remember you are allowed to substitute a variable with anything, you can even substitute with another variable.  From this I will infer Below Floor z.

(Refer Slide Time: 14:33)


KRR-Wed-Feb4 - Windows Journal
File Edit View Insert Actions Tools Help

Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ if there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

~~$\forall x \forall y$~~ Above(?x, ?y) ⊃ Below(?y, ?x)

$\delta = \{\langle ?x, ?z\rangle, \langle ?y, floor\rangle\}$

Above(?z, floor)

Below(floor, ?z)

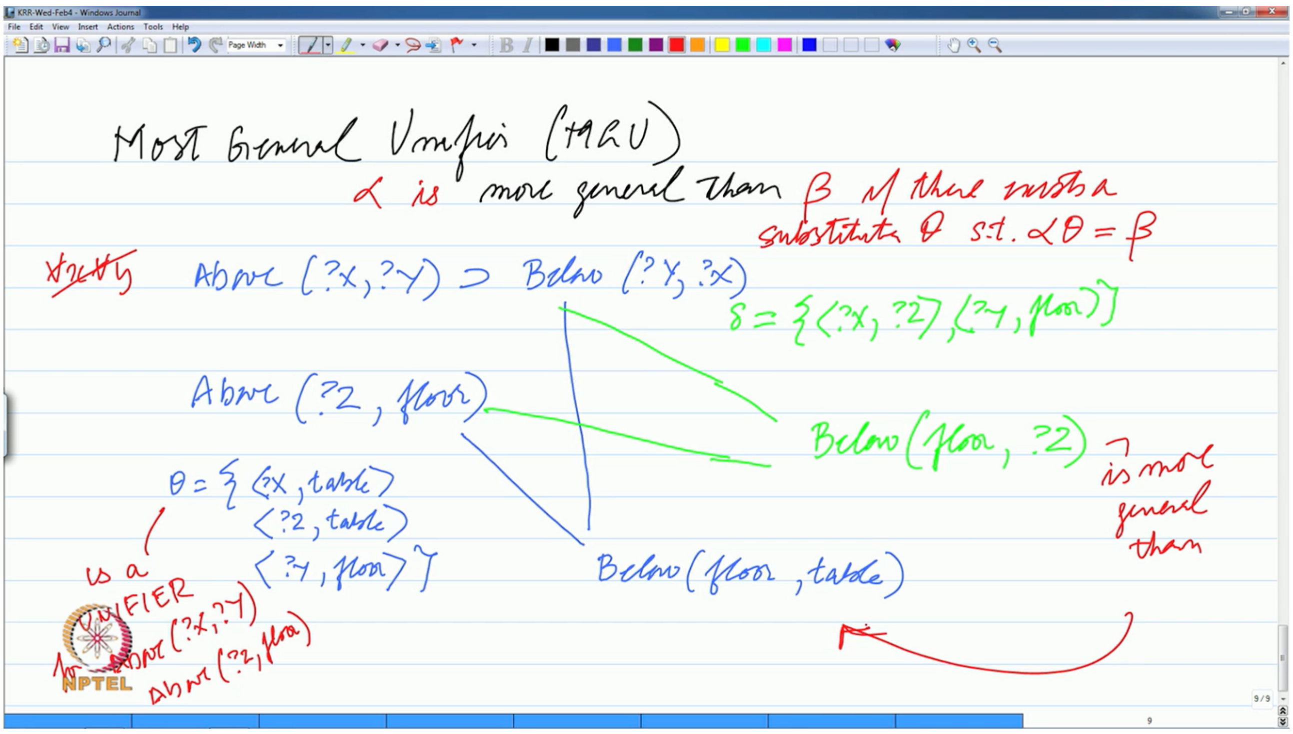
$\theta = \{\langle ?x, table\rangle$
$\langle ?z, table\rangle$
$\langle ?y, floor\rangle\}$

is a UNIFIER for Above(?x, ?y) Above(?z, floor)

Below(floor, table)

Let's just apply the modified modus ponens rule, apply this substitution theta and you have produced a new inference which says that the floor is below everything else. So observe that the second this thing is more general than this one.

(Refer Slide Time: 15:10)



Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ if there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

~~$\forall x \forall y$~~ Above (?x, ?y) $\supset$ Below (?y, ?x)

$\delta = \{\langle ?x, ?z\rangle, \langle ?y, \text{floor}\rangle\}$

Above (?z, floor)

Below (floor, ?z) is more general than

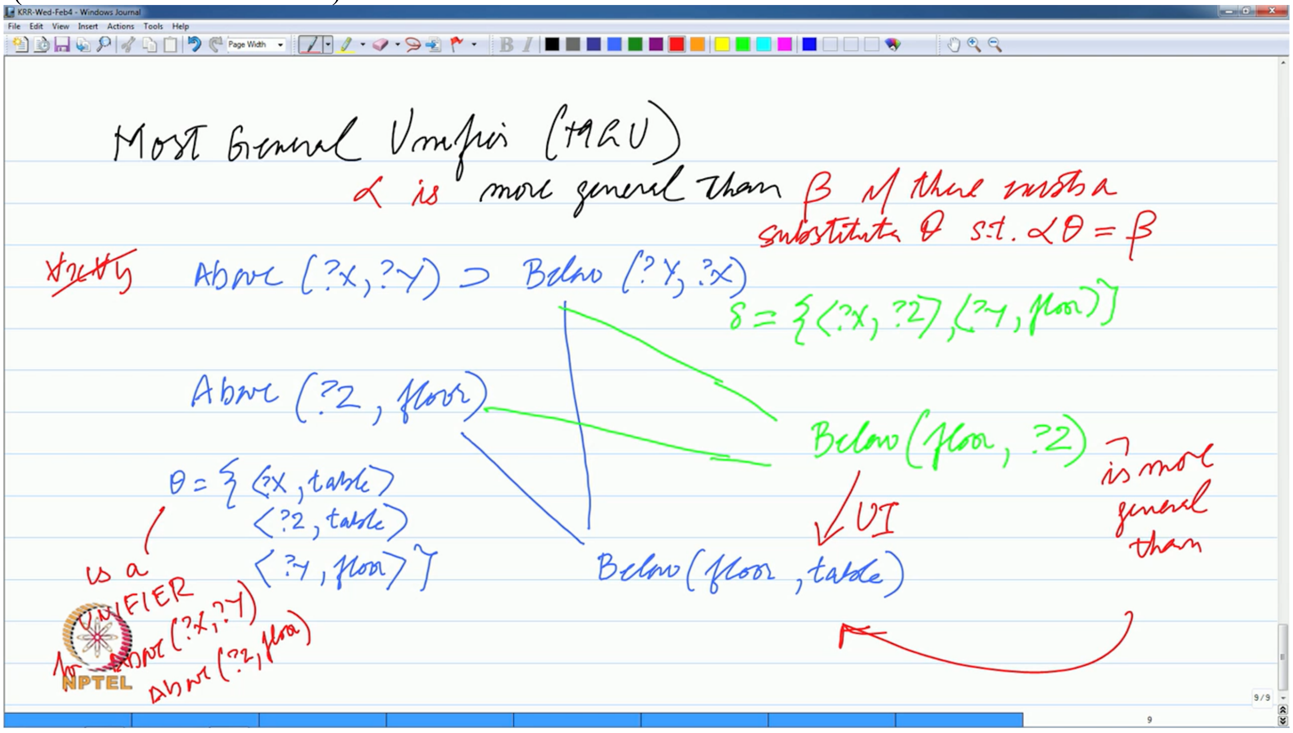
$\theta = \{\langle ?x, \text{table}\rangle$ $\langle ?z, \text{table}\rangle$ $\langle ?y, \text{floor}\rangle\}$

is a UNIFIER for Above (?x, ?y) Above (?z, floor)

Below (floor, table)

We have made two inferences, one of them is more general than the other essentially. Why is it nice to make more general inferences because you can always apply the universal instantiation rule? I can apply UI here and get the other. So it makes sense to make more general inferences than less general ones.

(Refer Slide Time: 15:45)



Most General Unifier (MGU)

$\alpha$ is more general than $\beta$ if there exists a substitution $\theta$ s.t. $\alpha\theta = \beta$

~~$\forall x \forall y$~~ Above (?x, ?y) $\supset$ Below (?y, ?x)

$\delta = \{\langle ?x, ?z\rangle, \langle ?y, \text{floor}\rangle\}$

Above (?z, floor)

Below (floor, ?z) is more general than

UI

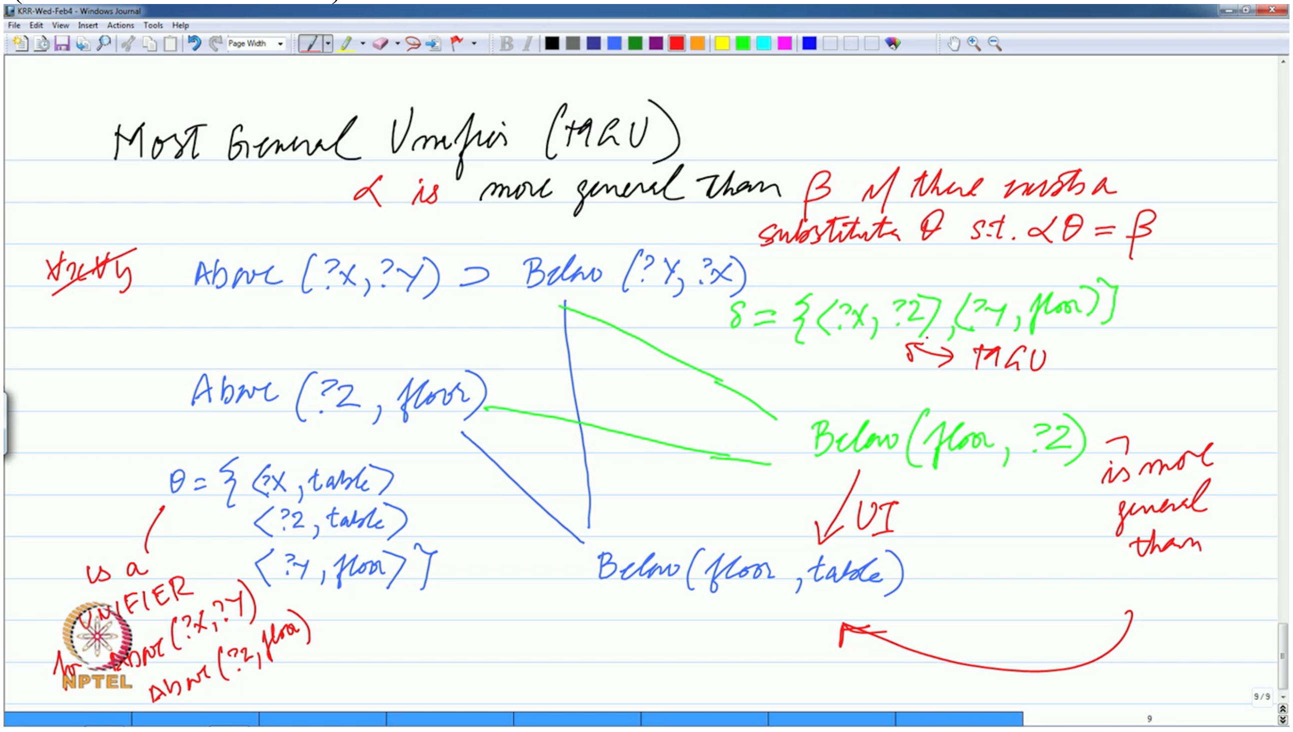
$\theta = \{\langle ?x, \text{table}\rangle$ $\langle ?z, \text{table}\rangle$ $\langle ?y, \text{floor}\rangle\}$

is a UNIFIER for Above (?x, ?y) Above (?z, floor)

Below (floor, table)

If I am told that everything is above the floor I can infer that the floor is below everything else. And then I can infer that the floor is below the table, floor is below the fan, floor is below the chair. All kinds of things I can do essentially. And you can see that Below floor everything is more general than Below floor table because when I substitute the variable z with table we will get this new formula. So the notion of more general than holds between these two formulae. So we generally extend the notion of seeking more general inferences to most general inferences because essentially we are saying that the more general the inference we can make the better for use essentially. So make the most general inference that you can make and the substitution that allows you to make the most general inference is called the most general unifier. So in this example this is the MGU.

(Refer Slide Time: 17:02)


KRR-Wed-Feb4 - Windows Journal
File Edit View Insert Actions Tools Help
Page Width
Most General Unifier (MGU)
α is more general than β if there exists a substitution θ s.t. αθ = β
~~∀x∀y~~
Above (?x, ?y) ⊃ Below (?y, ?x)
δ = {<?x, ?z>, <?y, floor>}
δ → MGU
Above (?z, floor)
Below (floor, ?z)
is more general than
θ = {<?x, table> <?z, table> <?y, floor>}
is a UNIFIER for Above (?x, ?y) Above (?z, floor)
UI
Below (floor, table)
NPTEL
9/9
9

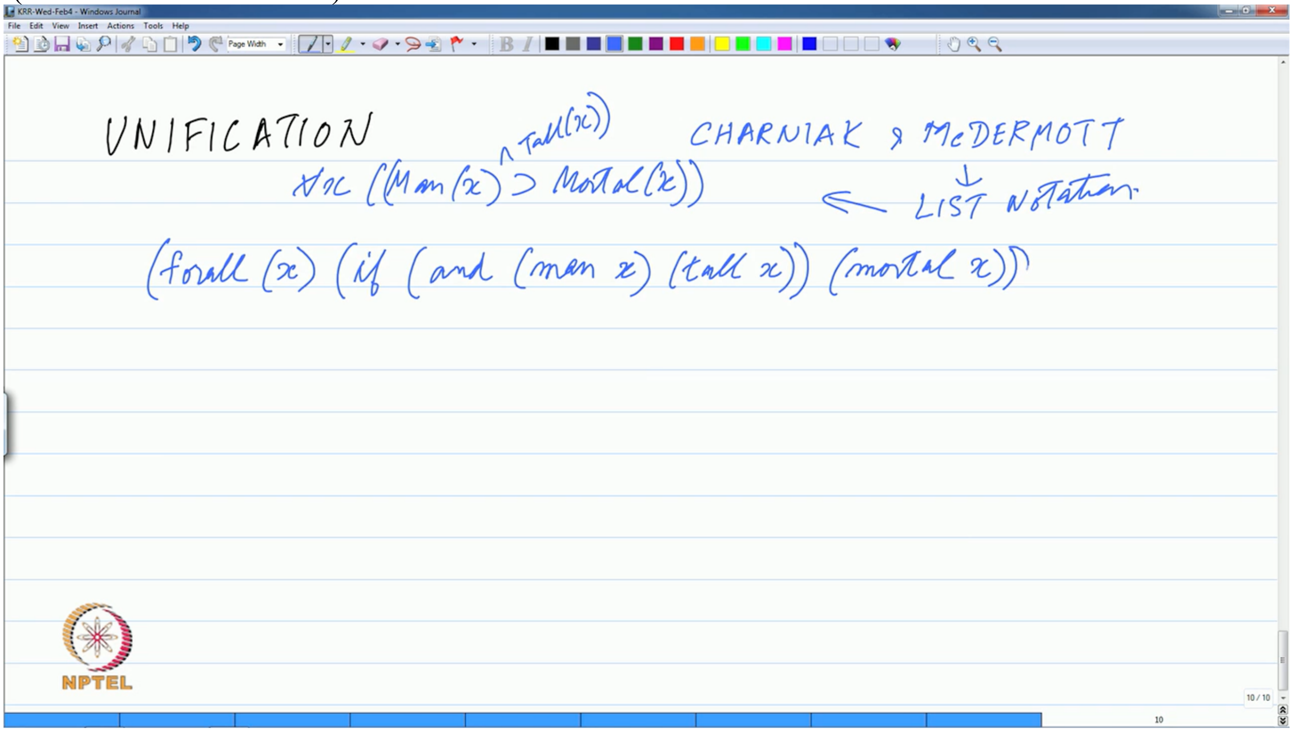
And we are interested in the algorithm which will allows us to find the most general unifier and this algorithm is a very well-known algorithm, called the unification algorithm. Its kind of central to theorem proving in general or logical reasoning in general. I have not defined what is the meaning of most general but you know that given in any partial order or given any lattice when you have an ordering which says that more general you can easily define what is most general and so on and so forth. So something is most general if there is nothing which is more general in that so we will not define it here essentially.

But I expect that you have accepted the notion of most general unifier and we are looking at an algorithm for doing that. So only for this algorithm I want to also introduce this different way of writing formulae which you can find in this book by Charniak and McDermott, the book is called Artificial Intelligence and they use what is a LISP notation for writing the same logic formulae essentially. It's nice to use LISP notation because when you are writing programs they are easier to process as compared to the mathematical notation FOL uses essentially. So in this LISP notation the statement for all x Man x implies Mortal x. This has become our favourite statement will be represented as follows and I will just use this illustration and you can work out what this LISP notation works essentially.

So the first thing is that the connectives are not infix, that they are prefix, they come before the two operands and secondly we don't use any special symbols like the connective symbols we are using or the quantifier symbols that we are using. We would write this formula as we use for all and then we give the variable name, that is the quantified variable, then we start the scope of the quantifier, in this example, that's rest of the formula. Then we first write the connective and we use this notation *if* since we are familiar with that so let me use lowercase. Then we begin a bracket to put the left hand 0:49

So now I have to put the antecedent of the implication. Antecedent is a conjunct so I start by writing the connective then my first argument to and is man x which also I write in this notation. So I put the predicate symbol also inside the list. The first element in the list is always the important thing, it’s either a predicate or function or a connective. That if you keep in mind you will remember. The next argument to this is tall x, then I close the bracket to say that I finished with the and part. Now I write the then part in the sentence. We don’t write then because it is enough to identify that, we write mortal x and an appropriate number of brackets. That’s always the problem with the LISP notation to figure out how many brackets to use.

(Refer Slide Time: 22:05)



UNIFICATION

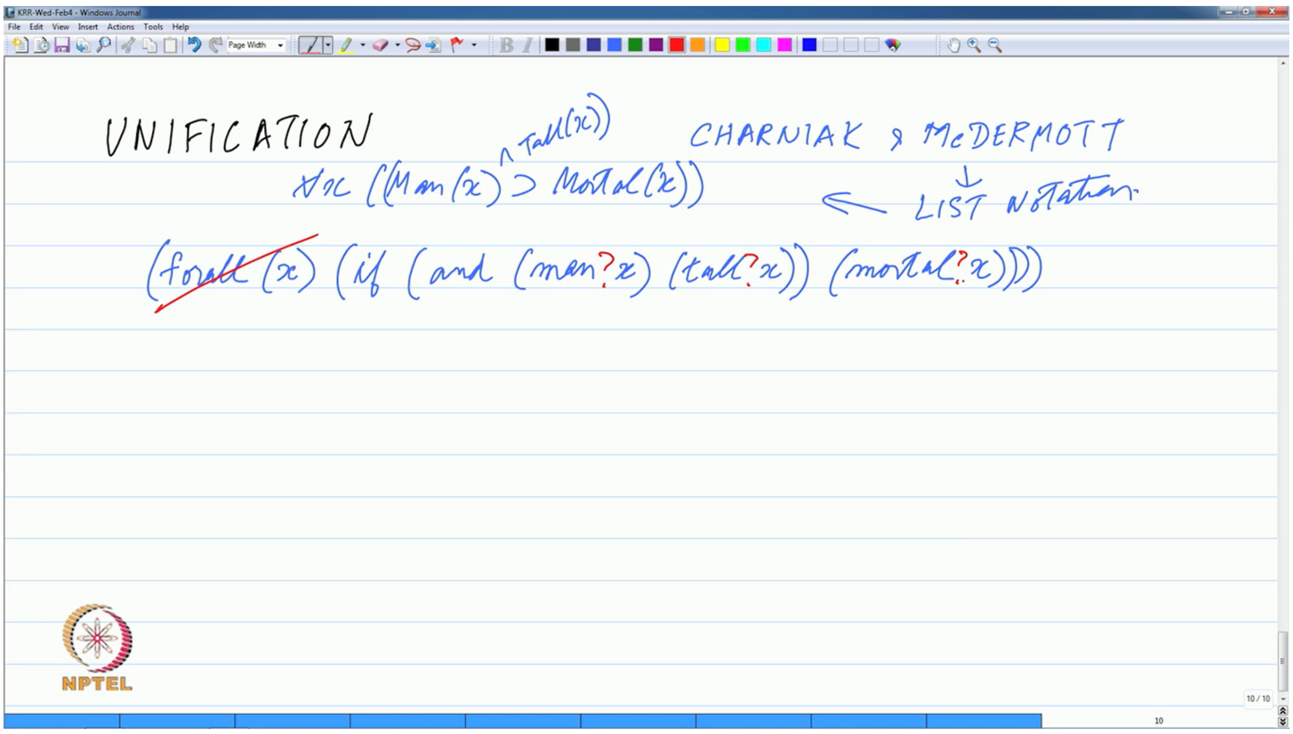
CHARNIAK & McDERMOTT

$\forall x\ ((Man(x) \wedge Tall(x)) \supset Mortal(x))$ $\longleftarrow$ LIST Notation

(forall (x) (if (and (man x) (tall x)) (mortal x)))

So let’s hope this is correct, maybe you need one more bracket. So only for this algorithm sake we will use this notation but in practice of course if you are writing programs without any reasoning then you will see that this notation is simpler to handle than the mathematical notation which requires special treatment to different kinds of things. Here everything is a list essentially.  So the unification algorithm that we will look at, we will just look at tool essentially and in the implicit quantifier form we simply throw away this part and we put a question mark wherever the case may be. So that is simple. So we are only left with this if and formula essentially.

(Refer Slide Time: 22:51)


UNIFICATION

CHARNIAK & McDERMOTT

$\downarrow$

$\leftarrow$ LIST Notation

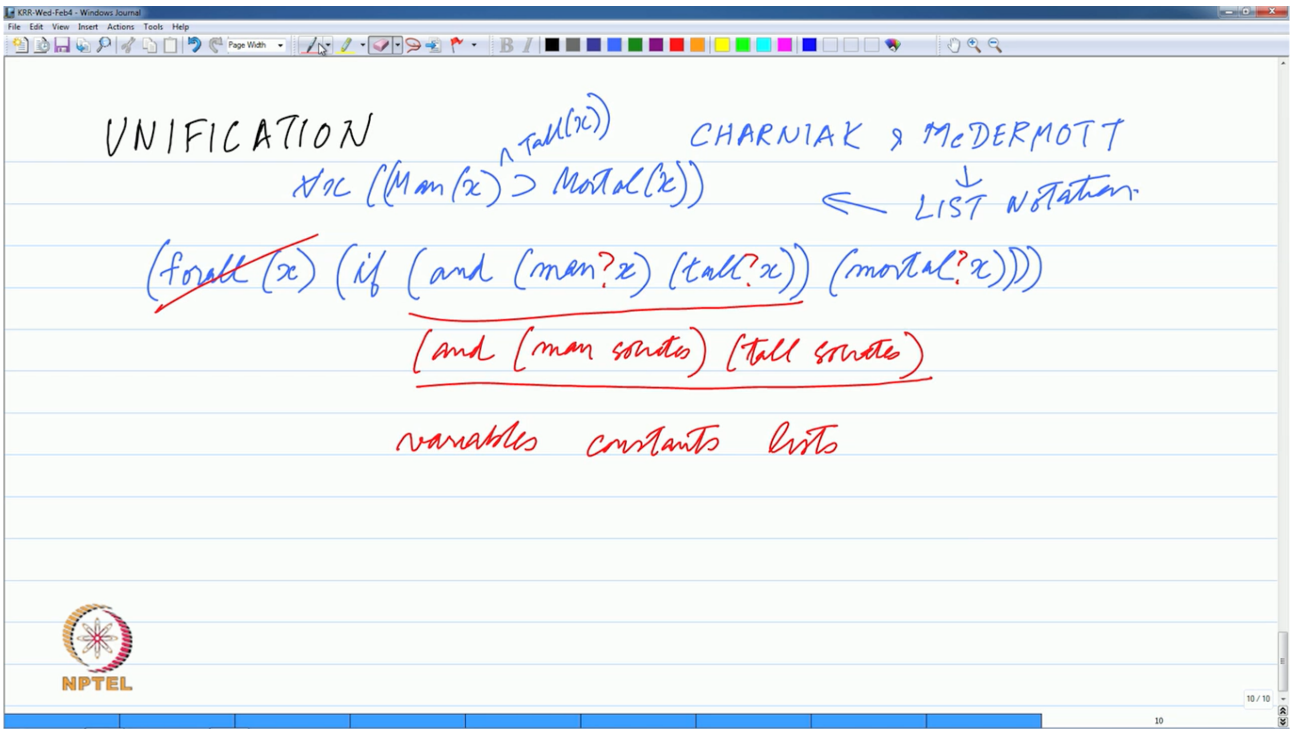
$\forall x\ ((Man(x) \wedge Tall(x)) \supset Mortal(x))$

(~~forall~~ (x) (if (and (man ?x) (tall ?x)) (mortal ?x)))

So for the sake of this algorithm we will distinguish between variables, so we will assume that we have a way of testing that it is a variable, constants and lists. So for the sake of unification, remember what a unification is supposed to do is to take two formulas or two expressions and find the substitution which will make them the equal. We don't particularly care whether it is a logical connective or it's a predicate name or function name, it doesn't really matter. Because we are now really interested in making the two expressions equal essentially. And the only thing we are allowed to do is we are allowed to substitute variables with something else. So that's the only thing we are allowed.

So if I had to let's see we are given this fact that if and man Socrates tall Socrates, I don't know whether he was tall or not. Let's assume he was tall. The goal is to unify this formula, sorry the left hand side which is this. I am not given sentence, if I am given sentence that all tall men are mortals which in implicate quantifier from the first sentence in blue to and the second sentence which is Socrates is a tall man which is the sentence in the red. Then if I can unify the underlying cards, them i can apply the modified modus ponens rules too, infer the right hand side essentially.

(Refer Slide Time: 25:16)


KRR-Wed-Feb4 - Windows Journal
File Edit View Insert Actions Tools Help
Page Width
NPTEL
10 / 10
10

UNIFICATION

$\forall x\, ((Man(x) \wedge Tall(x)) \supset Mortal(x))$

CHARNIAK & McDERMOTT
↓
← LIST Notation

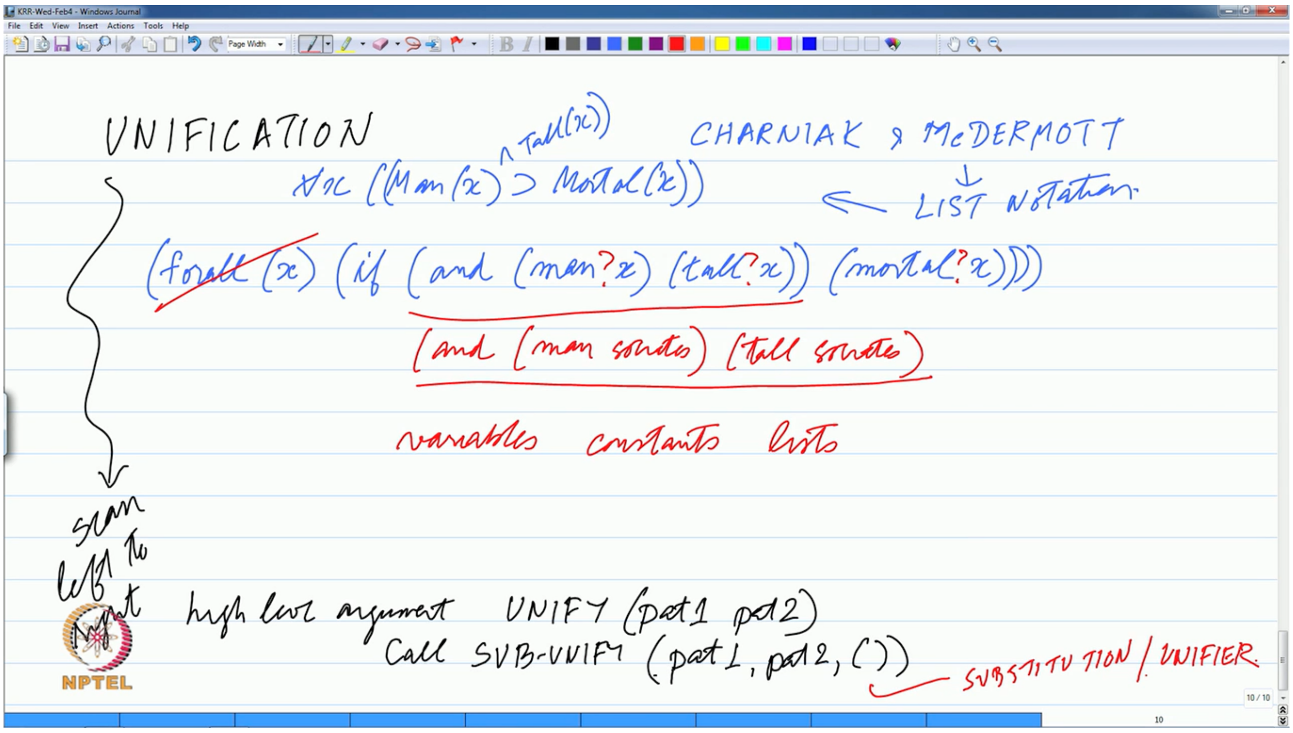
(~~forall (x)~~ (if (and (man ?x) (tall ?x)) (mortal ?x)))

(and (man socrates) (tall socrates))

variables constants lists

Now as far as unification algorithm is concerned we just want to treat variables and constants and lists separately. So I will just give an outline now and I hope you will work out the details. The basic idea behind the algorithm is to scan left to right, the two formulae the two expressions which are given to us, trying to unify piecewise in a recursive manner essentially. So of course at the high level the algorithm will have something so the high level argument would be like this that we would say UNIFY pat1 pat 2, so this is the notation I am following from  Charniak and McDermott, pat1 and pat2 refer to the two expressions, or patterns that you are trying to make equal. The first thing that you do is make a call to another. Ok so I should use the proper notations here. So if you have an algorithm called UNIFY pat1 and pat2, we call another algorithm called SUBUNIFY with pat1, pat2 and an empty list or an empty set if you would like to say. We include a third argument and you must be familiar with this style of writing algorithms where you include a third argument and you keep building up whatever you want to this thing.  Our idea is that substitution will be stored here.

(Refer Slide Time: 28:00)
]



UNIFICATION

CHARNIAK & McDERMOTT ↓ LIST Notation

$\forall x\, ((Man(x) \wedge Tall(x)) \supset Mortal(x))$

(~~forall (x)~~ (if (and (man ?x) (tall ?x)) (mortal ?x)))

(and (man socrates) (tall socrates))

variables constants lists

scan left to right

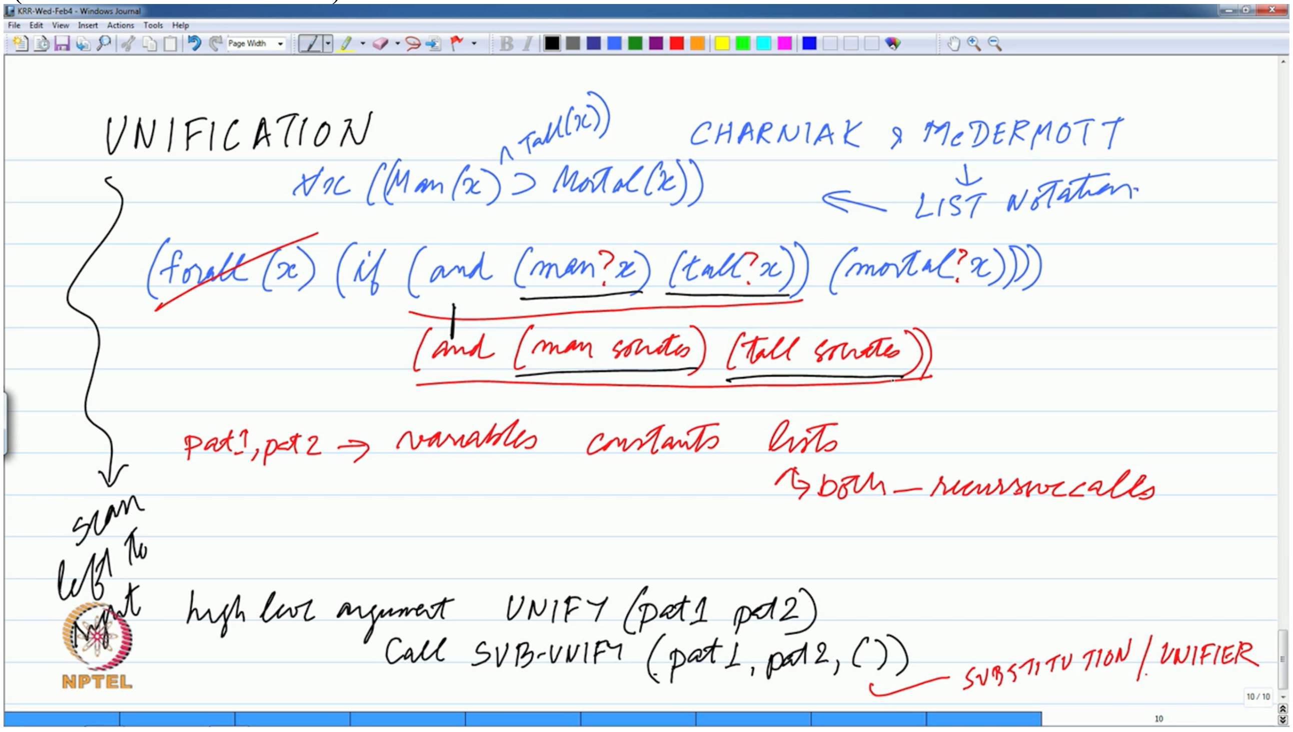
high level argument UNIFY (pat1 pat2)

Call SUB-UNIFY (pat1, pat2, ())

SUBSTITUTION / UNIFIER

so we will essentially scan the two inputs pat1 and pat2, left to right is totally incidental, typically algorithms have to do it in a deterministic way and do one of the following, keep building up one of the substitution essentially. So when you say pat1 or pat2 they refer to these, they are either variables or functions or they are lists. So let's deal with the simpler part first, when both are lists both pat1 or pat2 are lists, then recursively with the first element of this list, the second element of the list, the third element of the list and so on. So in this example that you can see that in fact both the patters that we are trying to unify and Man x Tall x, and Man Socrates Tall Socrates, these are both are lists. So we will first try to unify and with and, then this sublist with this sublist which will be done recursively and the second sublist with the seconf sublist.

(Refer Slide Time: 29:46)



UNIFICATION

CHARNIAK & McDERMOTT ↓ LIST Notation

$\forall x\, ((Man(x) \wedge Tall(x)) \supset Mortal(x))$

(~~forall (x)~~ (if (and (man ?x) (tall ?x)) (mortal ?x)))

(and (man socrates) (tall socrates))

pat1, pat2 → variables constants lists

↳ both — recursive calls

scan left to right

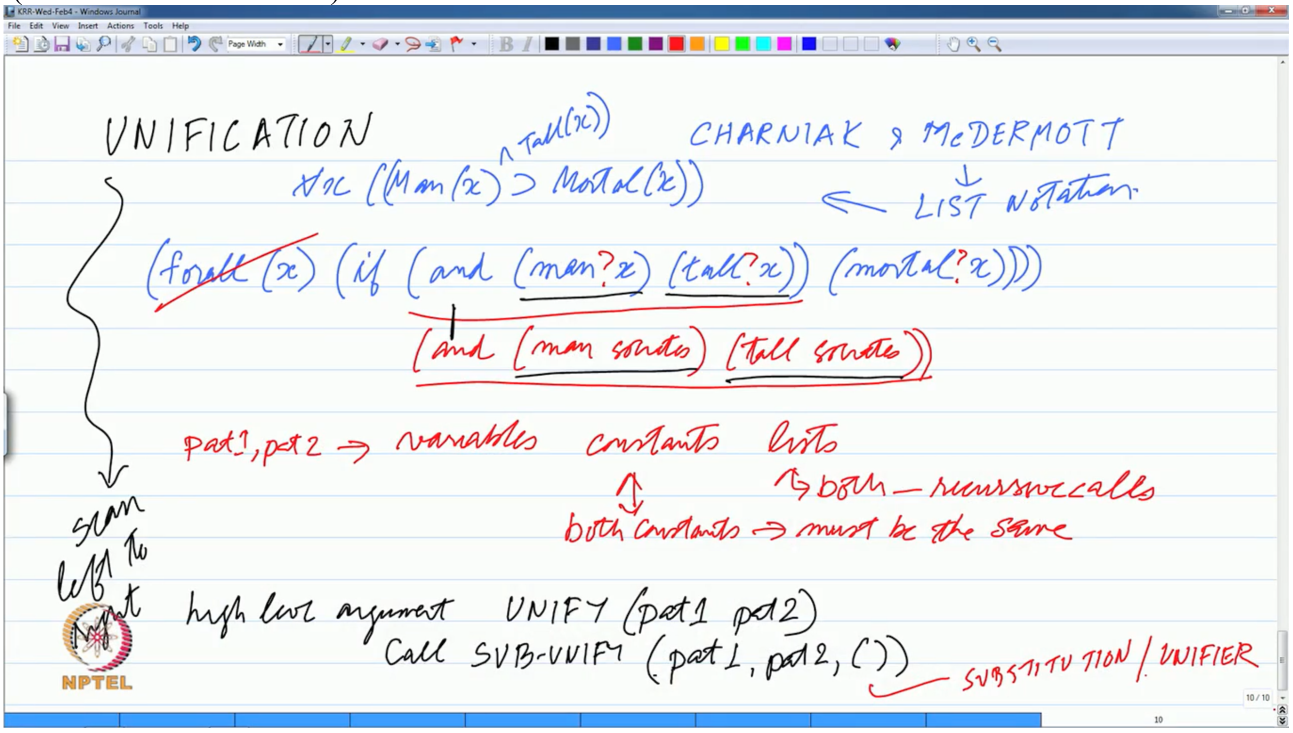
high level argument UNIFY (pat1 pat2)

Call SUB-UNIFY (pat1, pat2, ())

SUBSTITUTION / UNIFIER

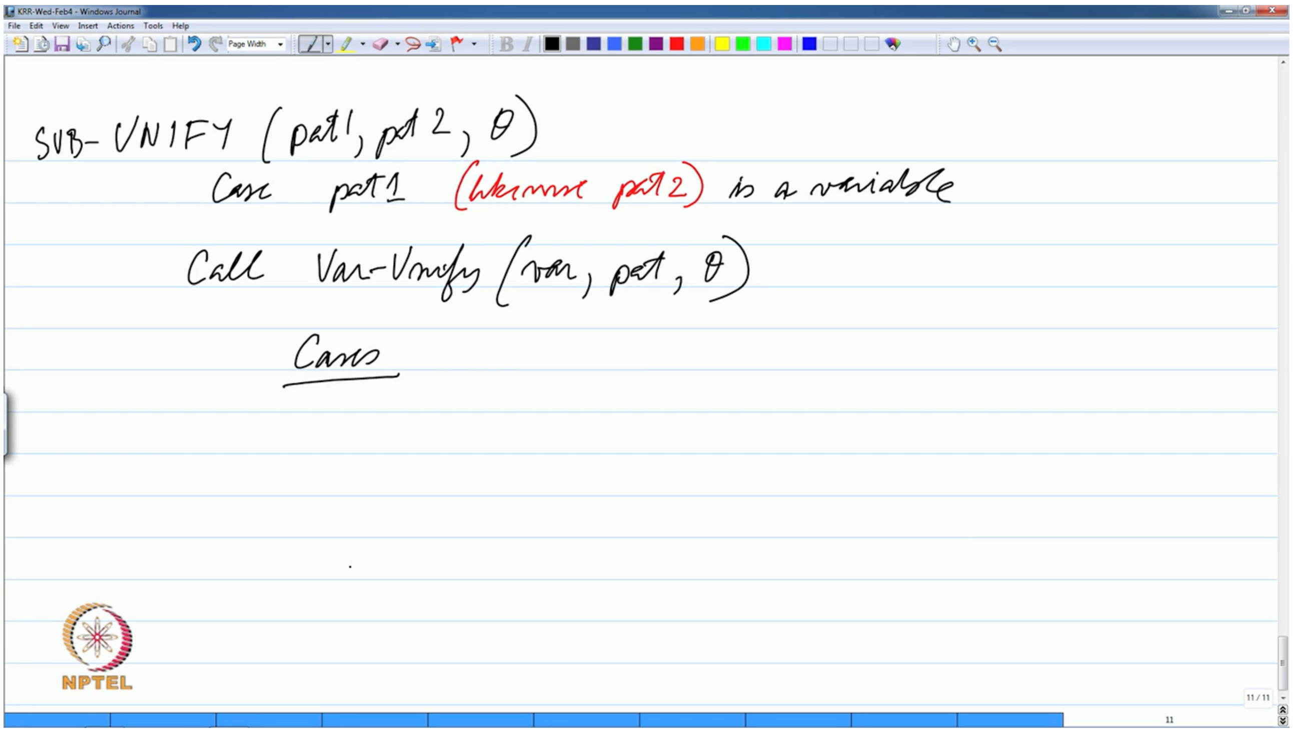
So if both are lists and both must be of the same length which I have not written here, because you cannot unify lists with different lengths, then you can unify them piecewise in a recursive fashion so call SUBUNIFY with the first one, then with the second one, then with the third one in this example and at all the stages keep augmenting the substitution that you are trying to find. If it is constants, if both are constants then they must be the same. So we have only two kinds of things, either lists or constants or variables. Variables are the ones which have question mark below them and everything else is a constant essentially. So and is a constant here in these two lists so they must be the same, an and will match only an and. Man is a constant, it's a predicate as far as we are concerned but as far as unification algorithm goes it's a constant. So Man must match Man exactly, Tall must match Tall. So wherever they occur they must be the same constants. That leaves us with variables

(Refer Slide Time: 31:11)


KRR-Wed-Feb4 - Windows Journal
File Edit View Insert Actions Tools Help
UNIFICATION
∀x (Man(x) ∧ Tall(x)) ⊃ Mortal(x))
CHARNIAK & McDERMOTT
LIST NOTATION
(~~forall~~ (x) (if (and (man ?x) (tall ?x)) (mortal ?x)))
(and (man socrates) (tall socrates))
pat1, pat2 → variables constants lists
both — recursive calls
both constants ⇒ must be the same
scan left to right
high level argument UNIFY (pat1 pat2)
Call SUB-UNIFY (pat1, pat2, ())
SUBSTITUTION / UNIFIER
NPTEL

So I will just take a few minutes to talk about variables. So unification, so we are talking of SUBUNIFY pat1, pat2 and some theta that we have built up when this particular call to SUBUNIFY is being made essentially. So we have dealt with constant and lists, constants must be identical, lists must be of the same length, and we will make recursive calls to this SUBUNIFY. Now we are left with the case that we have to deal with variables. So Case pat1 and likewise I would say for pat 2. Essentially if either of them is a variable, so let's say pat1 is a variable, then we say call another function which is really doing the hard work so we will say var, pat and theta. So this is another subroutine that we are writing or submodule we are writing in which the first argument is a variable, the second argument is a pattern and the third argument is a theta. So we have to look at a few cases here

(Refer Slide Time: 32:58)


SUB-UNIFY (pat1, pat 2, θ)

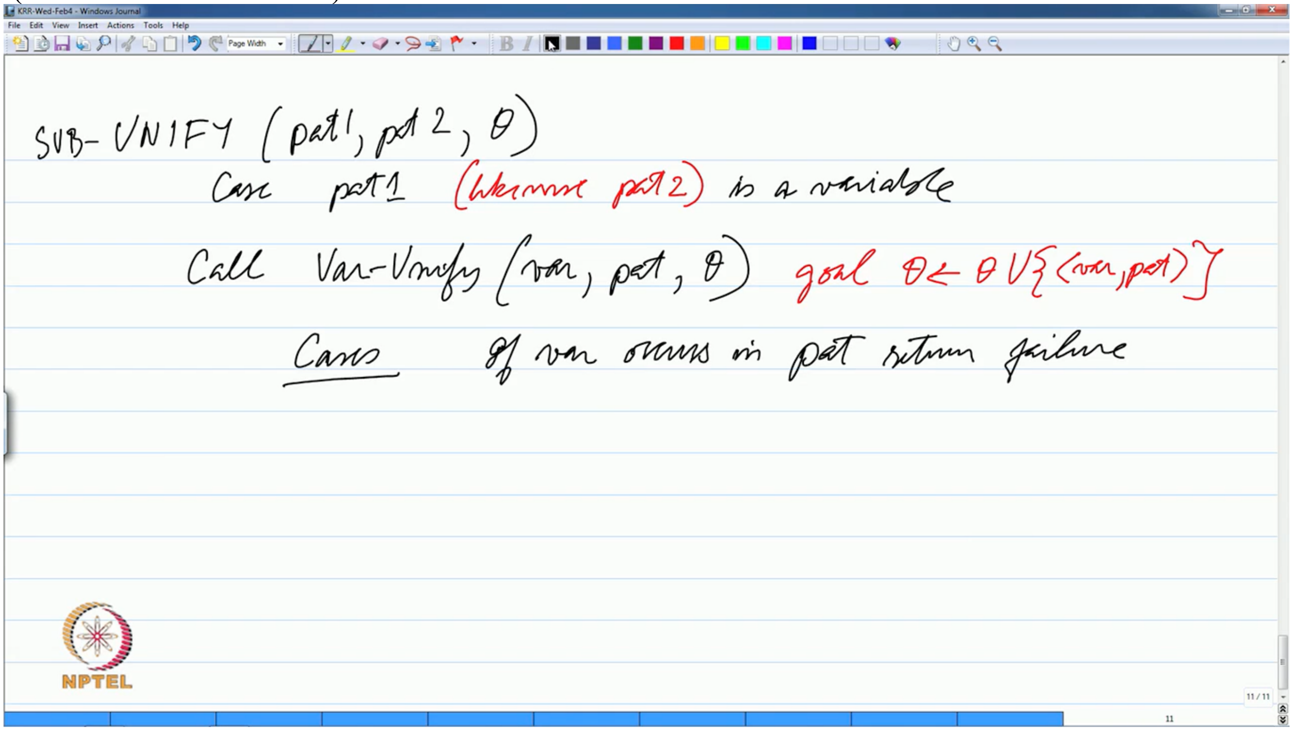
Case pat1 (whichever pat2) is a variable

Call Var-Unify (var, pat, θ)

Cases

If variable occurs in pat system failure. I will just state it simply, in the next class we will see an example and we will probably revisit the algorithm. So if the variable occurs inside the pattern, which means somewhere inside the pattern, it's there. Now remember what we are trying to do is to, remember our goal is to be able to add. Our goal is to eventually say that theta will become theta union var pat. If everything is fine, we will say substitute this pattern for this variable and that's one step towards making the two formulas the same. So that's the goal of this VAR-UNIFY module, to substitute pat for variable but we are putting but we are putting a check first, that if the variable occurs inside the pattern you cannot do that. You must think about this a little bit and we will see why this is the case.

(Refer Slide Time: 34:41)


SUB-UNIFY (pat1, pat 2, θ)

Case pat1 (whichever pat2) is a variable

Call Var-Unify (var, pat, θ) goal θ ← θ ∪ {(var, pat)}

Cases If var occurs in pat return failure

Then if variable pat3 belongs to theta, if the variable already has a value in theta then call SUBUNIFY with the pat, pattern that you are trying to unify the variable with, and this pattern which already has the value. Try to see if you can make them the same. If that is the case, then whatever the value is returned by them will be the value that you will add essentially. So if variable already has a substitution in theta then see whether that substitution and this current pattern will match or not. If you come though this otherwise you simply say theta or return theta union this set. So this is where really all the work is being done in this VARUNIFY this thing, it has these two tests before you can substitute the pattern for the variable and it has been shown that this algorithm will return the most general unifier which will make the least amounts of substitutions to make the two input patterns identical essentially. And it has also been shown that most general unifier for any two patterns in unique. This we will not go into detail here. So in the next class when we meet we will revisit this unification algorithm and we will see examples of why these two particular tests have to be done before you can actually apply add the substitution.