First Order Logic: Unification

Prof. Deepak Khemani

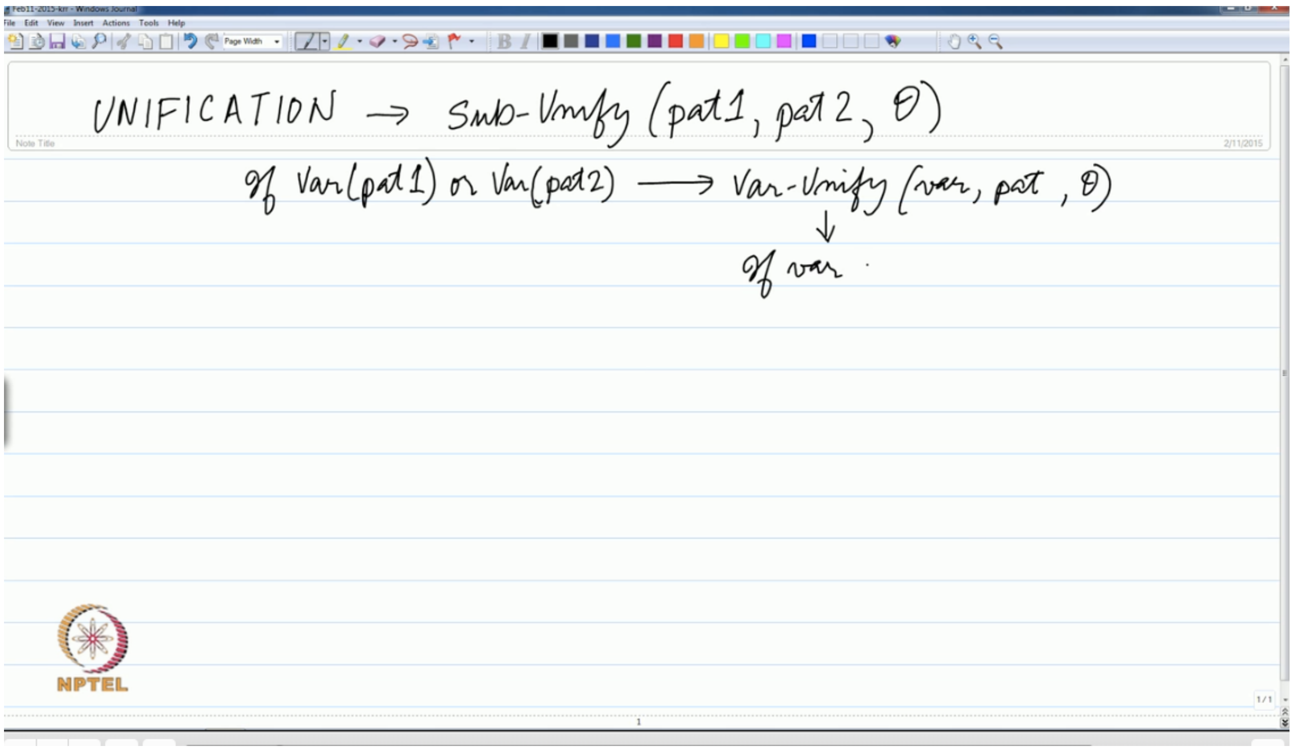
Department of Computer Science and Engineering

Indian Institute of Technology, Madras

Module – 03

Lecture - 16

Okay, So we are looking at algorithm for unification which takes two patterns, which may contain some variables, which try to find substation of two variable substitution of which will make two patterns identical. So the task is to make them identical. So it calls a module called sub-unify. It takes two patterns. Lets calls them pattern one and pattern two and a partially constructed substitution. Remember substitution is a set of variable value pairs. As far as we are concerned patterns are essentially list because we have converted connotation of FOL notation which is Charniak and McDermott notation of using a list. So a pattern is either a list or it is atomic in nature. When it is atomic it is either a constant or a variable essentially. So these are different cases we want to cater to when you are looking at pattern one and pattern two. So first case is I will do a quick review is



UNIFICATION $\rightarrow$ Sub-Unify (pat1, pat2, θ)

If Var(pat1) or Var(pat2) $\longrightarrow$ Var-Unify (var, pat, θ)

$\downarrow$

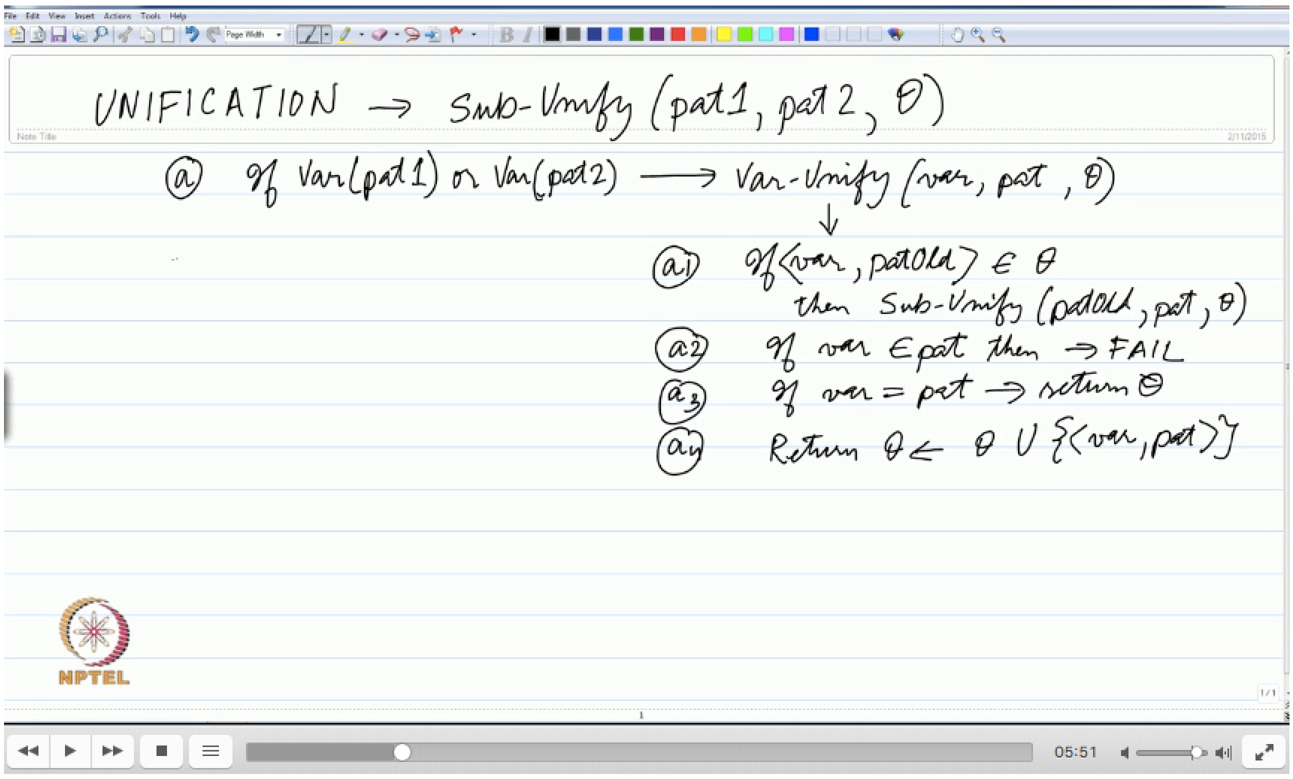
If var

that if . Lets say we have a function which will tell whether pat1 is a variable. This will return true or false or other one. If either of them is a variable we call a routine called var-unify, which takes the variable. I will use this very informal notation that which ever is the variable that is the variable here [Time 2.18].

Whatever is not the variable is called other here and the theta essentially. What this var-unify is trying to do is see other, or lets call it pattern just to be consistent. It will try to see whether it can make assignment of var equal to pattern that is the goal of this one. Before what we have to do a couple of tests.

If var occurs in theta, I will use this notation ‘belongs to’ theta. var already has a value in theta. Somewhere earlier you have encountered this variable earlier and gave this value to var. Or I should really say pattern of two already has a value in theta, then you call sub unify with pat-old, that pat we want to unify and theta. Lets call this case-a. Lets call this a1. [Time 4-23].

And a2 is if var belong to this pat itself this pattern itself. Then we will just return failure. We will see an example of it why it happens.



UNIFICATION $\rightarrow$ Sub-Unify (pat1, pat2, θ)

ⓐ If Var(pat1) or Var(pat2) $\longrightarrow$ Var-Unify (var, pat, θ)

(a1) If $\langle$var, patOld$\rangle \in \theta$ then Sub-Unify (patOld, pat, θ)

(a2) If var $\in$ pat then $\rightarrow$ FAIL

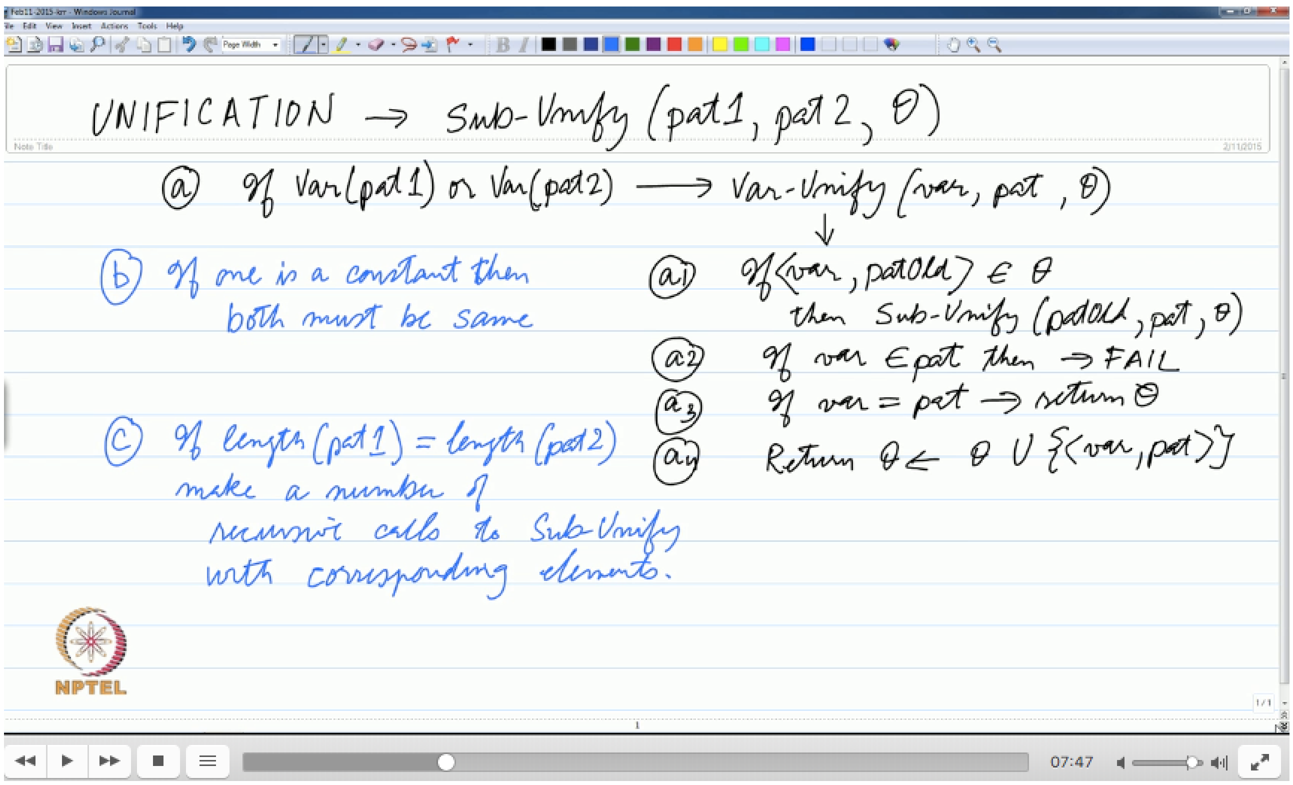
(a3) If var = pat $\rightarrow$ return θ

(a4) Return $\theta \leftarrow \theta \cup \{\langle$var, pat$\rangle\}$

And case 3 is if they are the same and return theta, you do not have to do anything. They are already same. Otherwise which is the case we are interested in a4. We add one more substitution in theta. The var unify is where we do all the work. If either of them is a variable, then we just call var-unify. [Time 05:51]
If we go past case a, then let me write this in different color. If you go to case b, case b means neither of them is a variable. Now if one of them is a constant then both must be same, which means other one must be also same constant essentially.

Case C is that both of them is a neither a variable nor a constant, which means both of them are lists essentially. so we just put a test, if length of pattern one is equal to length of pattern two, make a number of recursive calls to Sub-Unify with corresponding elements. Remember both are lists. [Time 07:47]

Lets look at an example. So let us say we have a statement which is in implicit quantifier form which say if . Left hand of antecedent has two things which are joined by an and, use a lower case. lets say we are looking at two predicates. One is S x. Lets say S stands for student. Lets say B is some other predicate. Two conditions on left hand side. What we are saying is, if x is a student and x is bright, then what. Then let us say fro argument sake , that x is a female. Rule says if x is a student and x is bright and x must be a female essentially. [Time 09:00]



UNIFICATION → Sub-Unify (pat1, pat2, θ)

ⓐ If Var(pat1) or Var(pat2) ⟶ Var-Unify (var, pat, θ)

ⓑ If one is a constant then both must be same

ⓐ1 If ⟨var, patOld⟩ ∈ θ then Sub-Unify (patOld, pat, θ)

ⓐ2 If var ∈ pat then → FAIL

ⓐ3 If var = pat → return θ

ⓐ4 Return θ ← θ ∪ {⟨var, pat⟩}

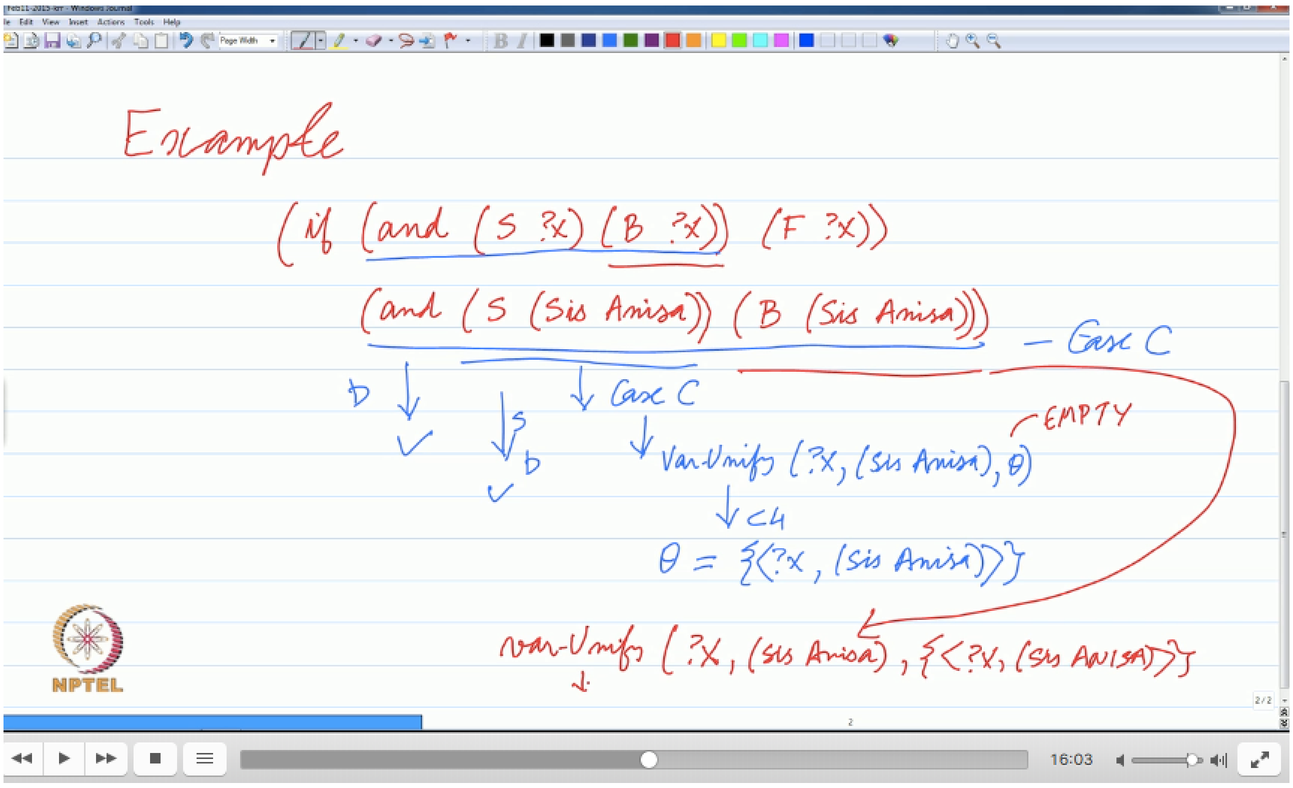
ⓒ If length(pat1) = length(pat2) make a number of recursive calls to Sub-Unify with corresponding elements.

Lets say we have another pattern which is the following. So we have this two patterns .[Time 1013]. And we want to apply this rule, then we need to see if left hand side of the rule which is underlined here [Time 1017] matches this pattern [Time 1022] here. If that is the case, then left hand side matches and using the modified Modus Ponens we can conclude the right hand side essentially. [Time 10 35]

So let me ask you here. What is (Sis Anisa) here. In this statement what this (Sis Anisa) refer to?
Student: a constant.
Anisa is a constant. What is sis? It must be a function. Remember our First Order logic, arguments to a predicate, So S is a predicate which stands for a student, sis must be a function. Which tells that it must be a function of Anisa. Lets say there is a function called sister, which means that everybody has only one sister. So it must be a function essentially. (Sis Anisa) is a term. Now in this case you can see that case a applies. [Time 1134]
If you remember case a, Sorry not case a, case c. because they are both lists. So we make corresponding calls with each of the element in the list essentially. So we make a call with and and and [Time 11 57] . In this case, case b will apply. And they are constants and they are same, so that will be accepted and the next element which is this one [Time 12 15] . Again Case c will apply. This is a list. We make corresponding calls. S, which is a constant as far as we are concerned. So this is case b. and both are same S and S, this is fine. The second element has a variable. question mark x is a variable. It is a universally quantified variable. So now we make a call to var-unify and call is with this variable and other thing a pattern in this case it is sis anisa. I must have whatever theta was. Lets say theta is empty to start with. We are starting this whole process essentially. [Time 1320]

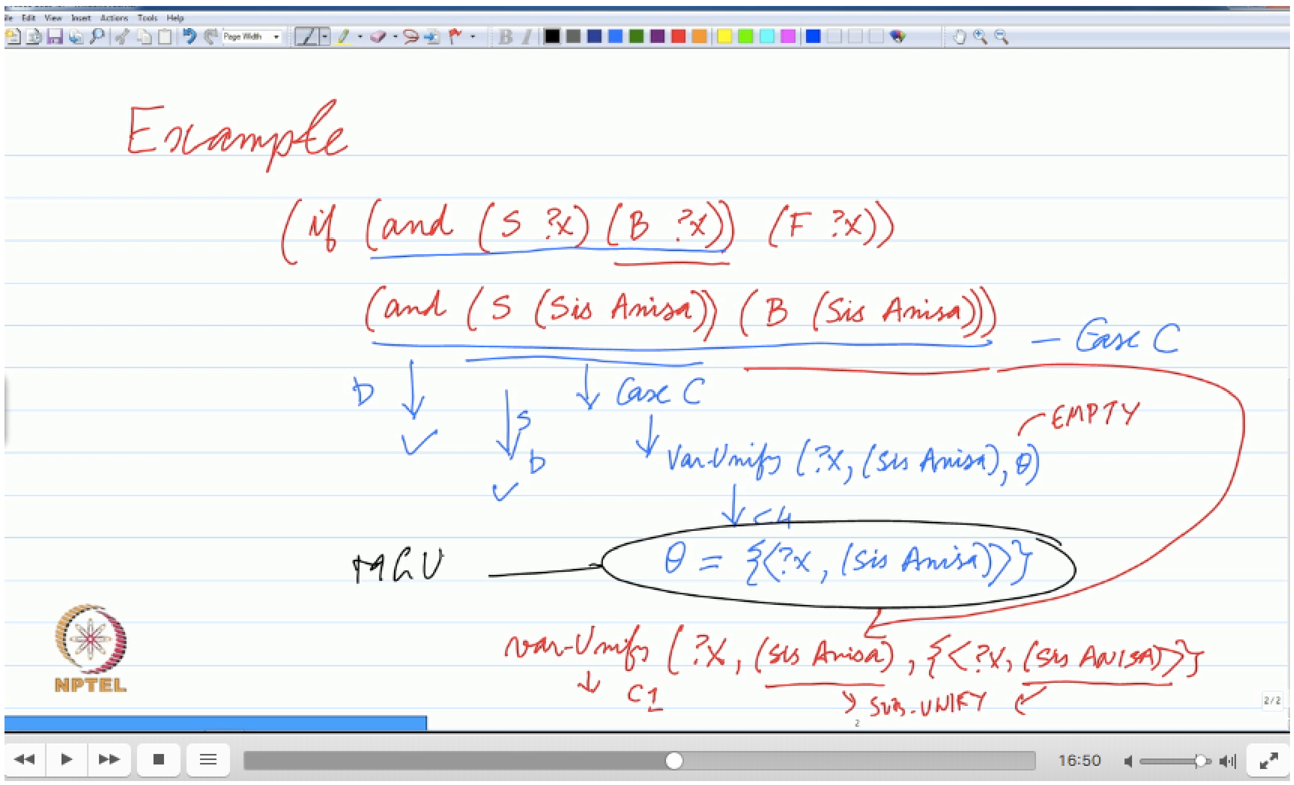
Now this var-unify will work fine if you go through cases. You will see case c four will apply. We will go through the test and none of the tests are true. essentially theta will now become x sis anisa [Time 1355]. Lets say theta was empty to start with. So we are finished with second argument s of x. Now we deal with the third argument, which is may be use a different colour here. x`

You can see that first both of them are list or both of them , either of them is a variable. Case C will


Example
(if (and (S ?x) (B ?x)) (F ?x))
(and (S (Sis Anisa)) (B (Sis Anisa)))
— Case C
Case C
EMPTY
Var-Unify (?x, (Sis Anisa), θ)
C4
θ = {⟨?x, (Sis Anisa)⟩}
var-Unify (?x, (Sis Anisa), {⟨?x, (Sis ANISA)⟩}
NPTEL
16:03

apply and correspondingly with correspondent element b will match b. We will make a call here also with var-unify x with Sis Anisa. Now theta will already have Sis Anisa. Let me just rewrite this. So with x, with Sis Anisa. This is a pattern and theta which contains this assignment already x Sis Anisa [Time 1602].

Now var-unify case c1 will apply, in this case x already has a value in theta. We will now make a call with these two arguments with sub-unify. May be you should just work out this problem. You will see that yes indeed we will find a match and unifier we will find is essentially this. This is MGU for both these patterns. [time 1650] Then we can infer that sister of Anisa is female essentially.


Example

(if (and (S ?x) (B ?x)) (F ?x))

(and (S (Sis Anisa)) (B (Sis Anisa))) — Case C

D ↓ ✓ ↓ S b ✓ ↓ Case C

↓ Var-Unify (?x, (Sis Anisa), θ) EMPTY

↓ C4

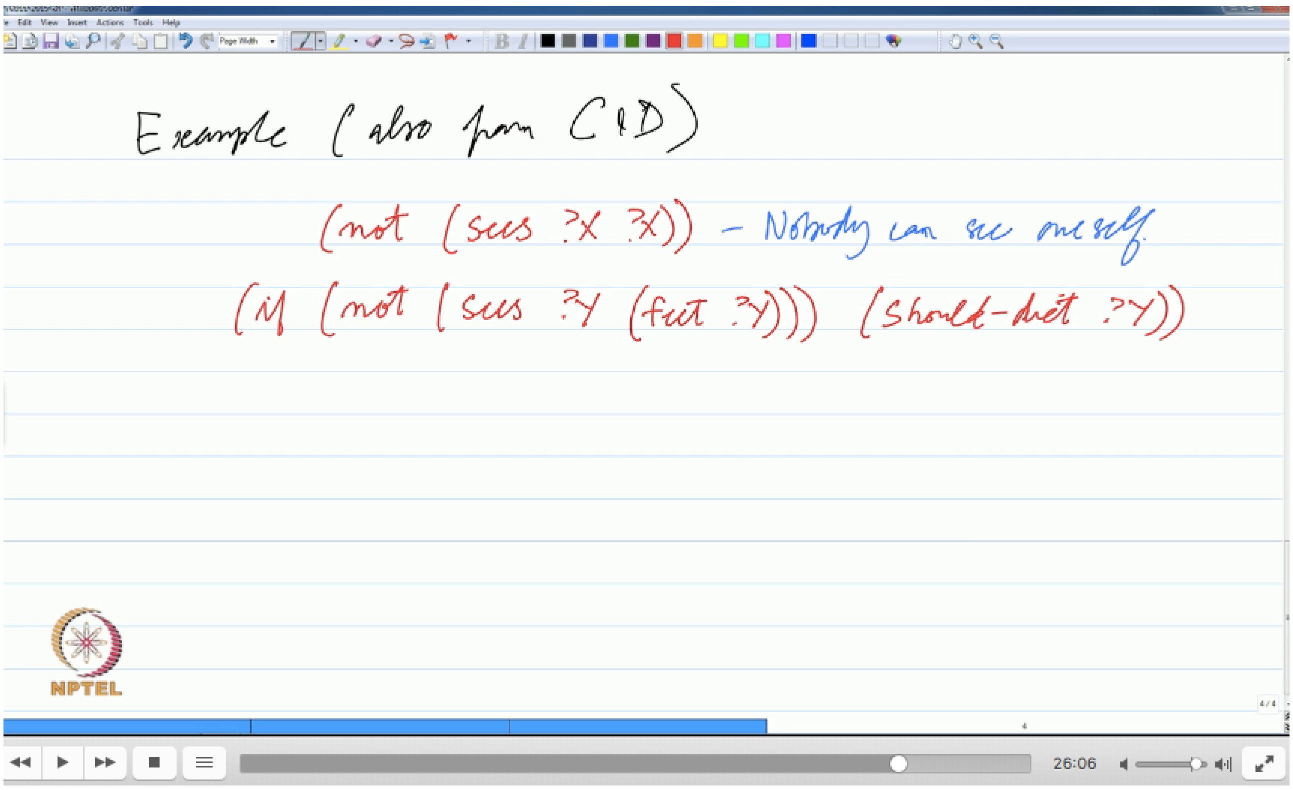
MGU — θ = {<?x, (Sis Anisa)>}

var-Unify (?x, (Sis Anisa), {<?x, (Sis ANISA)>})

↓ C1 Sub-UNIFY

16:50

Lets study one or two examples to know why we need those conditions.
Let me begin with another one here. This example is from this book i have told you about Charniak and McDermott. Lets say one of the fact given to us is on x. [Time 17:50]. How do you read this in english How do you say this. You have to speak a bit louder.
Students: Something on the table [Time 17:58]
Not something, everything on the table. The variable is a universally quantified variable. It is implicitly quantified. For all x implicit here which we are not seeing. Then other statement we have is if on big bertha then how do you read this? Big-Bertha is a constant, lets say for an individual. collapses is a function. on is a function. It say that if big-berths is on something, that something will collapse essentially. From this can we should be able to infer that table will collapse. Because big-berths is also on table essentially. Now if we have to use our unification algorithm, we have to match this with the left hand side of the rule [Time 1935]. This is always the case in modified modus ponens right. So the first case to both patterns will be applied, three recursive calls will be made. one with on, one with question mark x, and one with table essentially. The first one will match simply because they are constants. In fact they are the same constants. So that is okay. Second will give us the theta which is x … And third call will end up making a call to var-unify. If you go to the procedure, this intern will call sub-unify. Why? because x already has a value in theta. We are trying to unify x in table and x already has a value in theta. This call to sub-unify will be with big-bertha and table. and this theta that we have constructed is already there essentially.
What will this call to sub-unify return ? Why will this particular call to sub-unify fail? This is case 2. If you look at the cases we have. They are constants, but they are not the same constants. So we will return fail. I may not have written that as the part of the algorithm, but if they are not the same constants we say that we cannot unify them. Obviously you can not make big-bertha and the table the same thing [Time 2143].

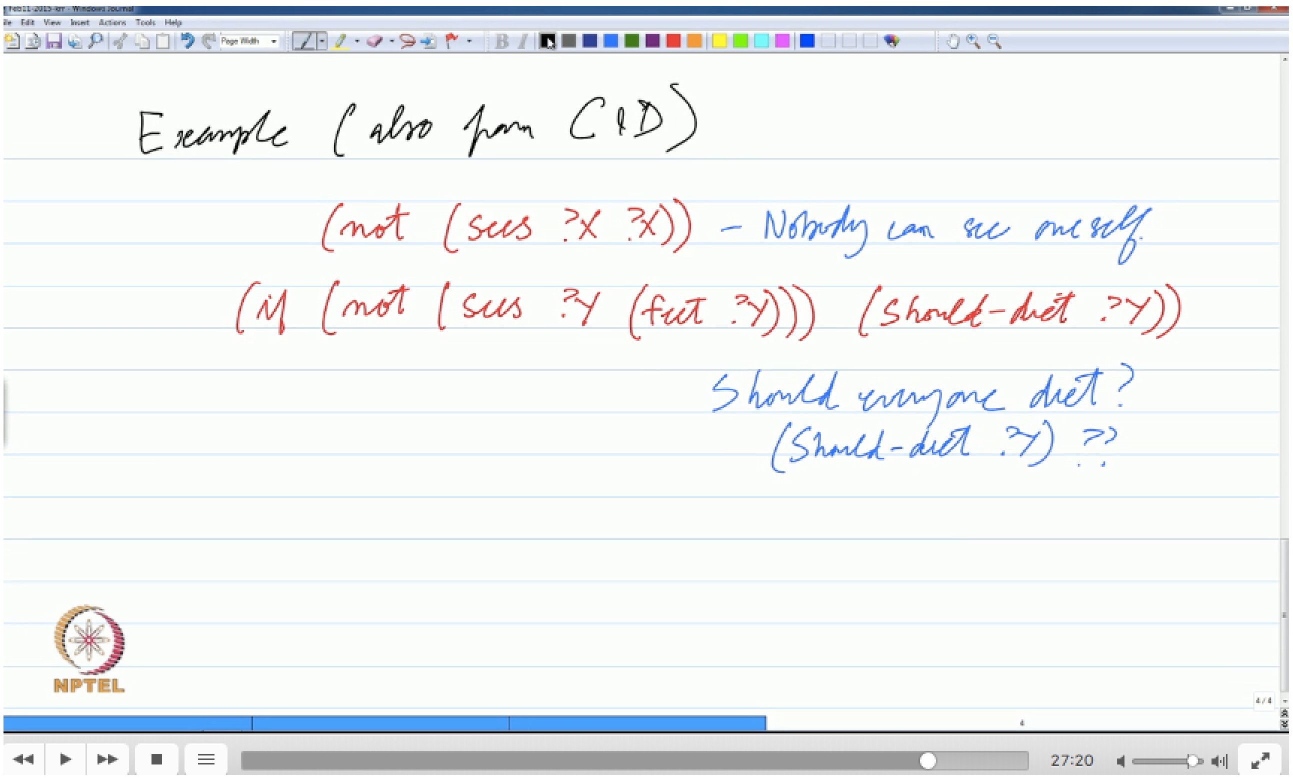


Example (also from C&D)

(not (sees ?x ?x)) – Nobody can see oneself

(if (not (sees ?y (feet ?y))) (should-diet ?y))

So we are not able to make the conclusion. So what has gone wrong. Any guesses to what gone wrong here? Variable names are same essentially. So the problem is these two are same variable names. If I have instead written, for example, the first statement I have written on y table. It would not have any difference logically to the statements at all. Because what this statement is saying that everything is on the table. For all x, x is on the table. For all y, y is on the table or if i say for all z, z is on the table, it does not really matter. They are the same statement. It is just that because we used the same variable name, we ran into this problem. So the lesson here is, let me write here, we must use different variable names in every statement. The term we say is standardise variables apart. This is something we must be careful about. Whenever we are using universally quantified statement and they are different statement, we must use a different variable name. Because once we put it in implicit quantifier form, there is nothing to tell us that they are different variables. As long as they were with in universal quantifiers, we can say that this variable is in the scope of this quantifier. This is something which must be done. Variables must be standardised apart. Lets look at another example, also from Charniak and McDermott book. This example has one statement. This is a logical not which is unary operator, it will take one argument. The argument is a predicate see. Lets say see stands for ability to see. sees x x means x can see x. If not we put there, it means that x can not see x. what is x ? x is universally quantified variable. What this statement is saying that nobody can see oneself. And then we have a statement. another statement in knowledge base . if not sees y , something, lets say, i will use example from the book, [Time 2606].
What does feet of x says here. It stands for feet of x. Logically what is it? in first order logic? Logically, in first order logic, it is a term. Just like sister of x, it is a term essentially. You must be careful about it. What this statement is trying to say in some sense is that if you can not see your feet they you should diet essentially. Now the question is, is it the case that should everyone diet? In other words, is this formula true? [Time 2700]
It should not be true. All we have said that you can not see yourself. And second statement says that if you can not see your feet , then you should diet. Lets see how unification handles it. We match this with this [Time 2724]. So first one is again constant, second one is a list which has got three components, out of which first one is a constant. sees matches with sees. Second one , they are variables. So Let us say that we say we substitute y with x and add it to theta. Because they


Example (also from C&D)

(not (sees ?X ?X)) – Nobody can see oneself

(if (not (sees ?Y (feet ?Y))) (should-diet ?Y))

Should everyone diet?
(Should-diet ?Y) ??

both are variables, you can simply replace one with another. This is an exercise you must work out. But the third case is of interest. You are trying to do var-unify x with feet of y and theta. x already has a value in theta. You end up calling var-unify with y and feet of y and the same theta. And here our case c2 [Time 29:11] will apply. Case 1c will apply or whichever the number is. And you will return fail eventually. And the test we have that if variable occurs in the pattern, then return failure, will catch this discrepancy here and we will say no, you can not apply this rule as you can not match this. Unification will return failure. So, luckily for us , all of don't have to go and diet essentially. Just think about what you are trying to do here. We are trying to, in this var-unify call, is substitute y with feet of y. Now what will happen. If you try to substitute y with feet of y. Inside that you will have to substitute y with feet of y and inside that you will have to substitute y with feet of y and obviously it is an never-ending process. So we have a rule that , if we have a variable in the pattern, just say you can not unify them. This takes care of the unification algorithm. Remember we need this algorithm to do forward chaining, forward reasoning. In the next class, we will come back to that and look at the strategies we can use for reasoning essentially. We will come back to forward chaining. And unification is just a module which we will be calling hence forth essentially.