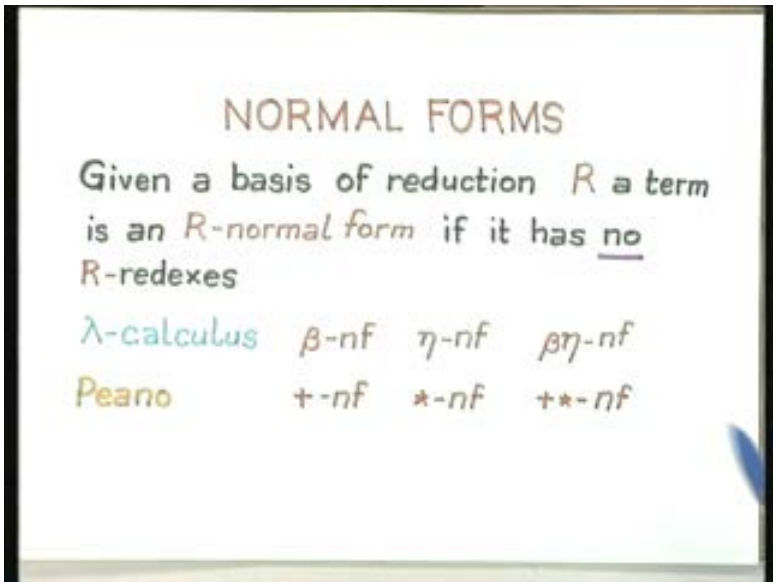
Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 31
Typed Lambda Calculus

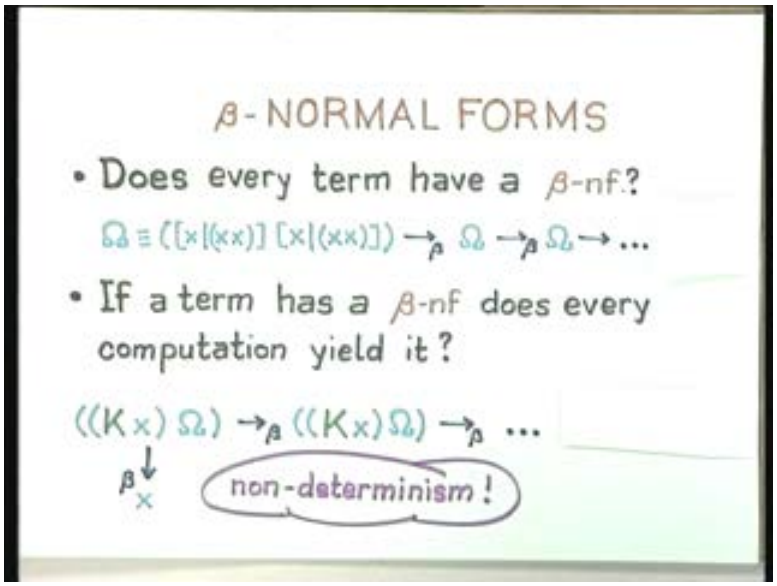
Welcome to lecture 31. Today we will begin the typed lambda calculus after briefly recapitulating what we did last time. So last time we looked at the question of normal forms. Normal forms are sort of fundamental to assign meanings and right from the cradle upwards they have always thought of the meaning of an expression as a value it denotes which is the normal form. So for any basis of reduction R if there are no redexes R redexes then you would say that it is a normal form.

(Refer Slide Time: 1:03)



Correspondingly for each basis of reduction we have a corresponding normal form. And in the lambda calculus we asked various questions; the main reduction being the beta reduction what we should ask is, are beta normal forms somehow guaranteed.

(Refer Slide Time: 1:34)

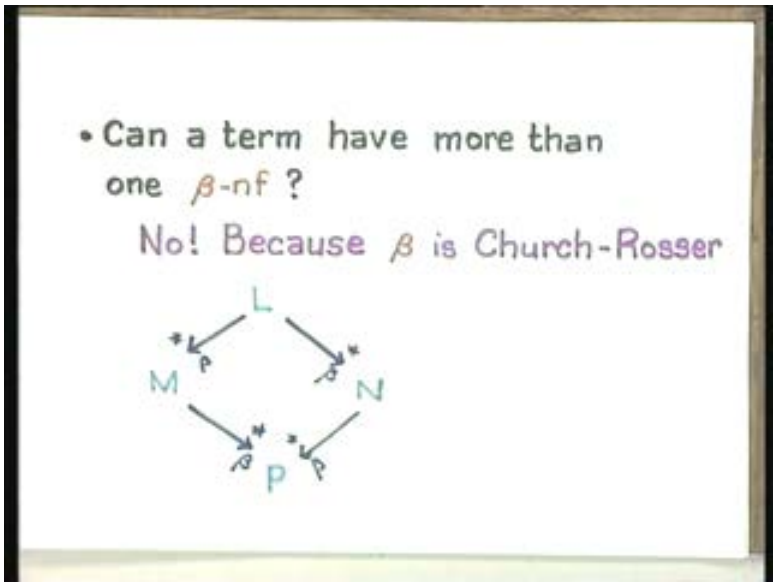


Does every term have a beta normal form? The answer is no because there are terms like omega which just goes on forever. If a term has a beta normal form does every computation yield it?

In case if you have non-determinism in the semantics of the language then it is clear that which such examples not every computation will yield that. However, the next question is if a term does have a normal form does every terminating computation yield the same beta normal form?

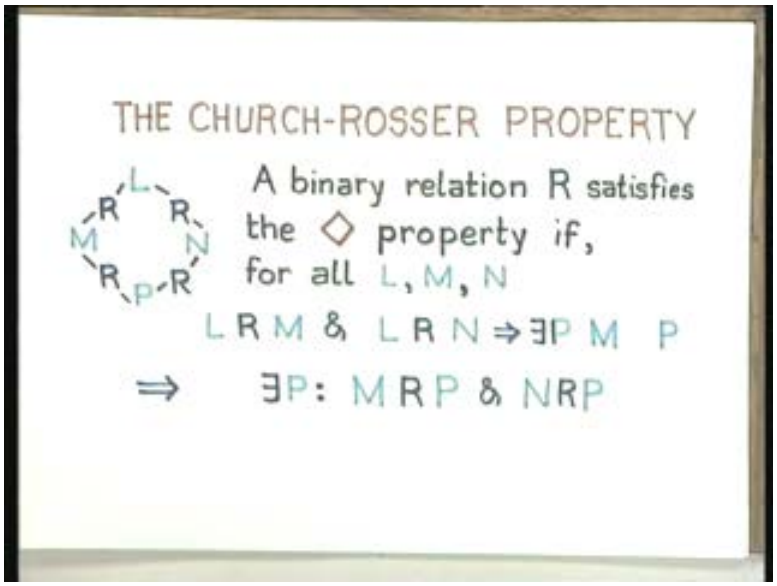
The condition for that is that if the basis of reduction is Church-Rosser then it is guaranteed to yield the same normal form for all possible terminating computations and if it is not Church-Rosser then there are no guarantees. So let us just briefly look at the Church-Rosser property.

(Refer Slide Time: 2:40)



We first looked at this diamond property a binary relation R satisfies the diamond property if for all (L, M, N) this diamond can be completed. That is if L is related to M and L is related to N then there exists a P such that M is related to P and N is related to P. So, given these three forming a triangle if you can complete the diamond for any arbitrary relation which does not necessarily have to be a basis of reduction then you would say that this satisfies the diamond property.

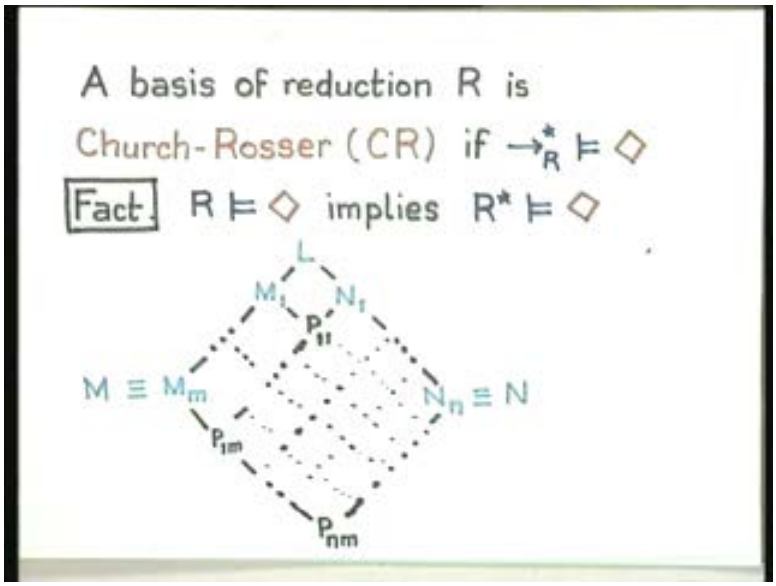
(Refer Slide Time: 3:35)



In particular if this binary relation is a basis of reduction then we would say that you take this basis of reduction, you close it compatibly over all possible term formations, take the

reflexive transitive closure of that which finally yields this many step reduction relation and then if this many step reduction relation satisfies the diamond property, this many step reduction relation is also a binary relation and if this satisfies the diamond property then you would say that the original basis that you started out with is Church-Rosser.

(Refer Slide Time: 4:30)



This fact just says that if you take any binary relation which satisfies the diamond property then its reflexive transitive closure will also satisfy the diamond property and the proof is very simple. In particular what we are saying is that if this fact is true for all possible binary relations and in particular if this was a one step reduction that means at compatible closure of some basis of reduction if a one step reduction satisfies the diamond property then the many step reduction also satisfies the diamond property and therefore the one step reduction is derived as compatible closure of some basis and that basis is Church-Rosser.

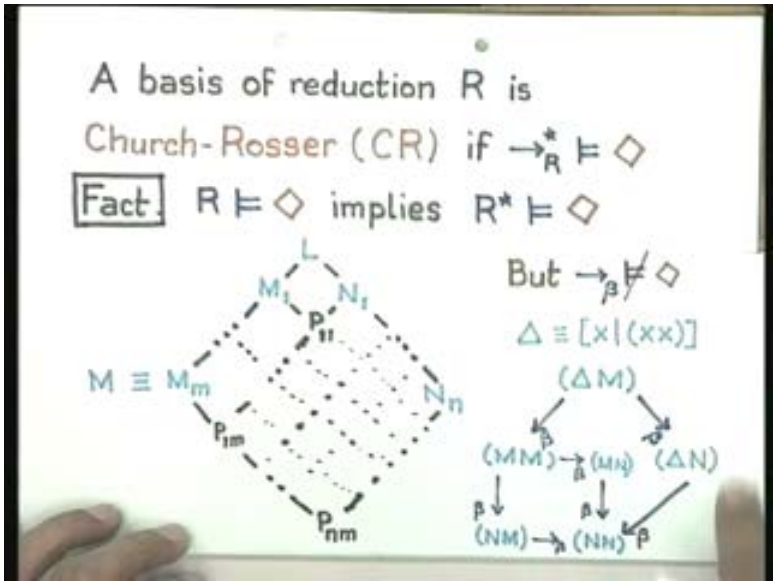
The proof of this is of course very simple, if you assume that you have a relation R which satisfies the diamond property so what it means is that if L is related to M and L is related to N then there exists a Pe such that M is related to P and N is related to P. And in order to show that the reflexive transitive closure satisfies the diamond property what we do is we prove it by induction on N and M assuming that L is related to M through M steps of composition and assume L is related to N through n steps of composition then we perform an induction on N and within it we perform an induction on M and we complete this slice.

And if you can complete this slice then by the inductive property you can complete this diamond. and to complete this slice you use the induction on M to complete each of these little diamonds and that would finally yield, if you can complete this slice then you can use lets say N_2 to complete this slice and then you can use N_3 to complete the next slice and so on and so forth till you reach N_n and when you complete the last slice you will get

this common term (P, N, M) which actually is again R power star related to M and R power star related to N and therefore it completes the diamond.

Therefore our problems would all be greatly simplified with respect to beta reduction if the one step beta reduction satisfies the diamond property. So if the one step beta reduction satisfied the diamond property then we would have proved that the basis beta is Church-Rosser.

(Refer Slide Time: 8:01)



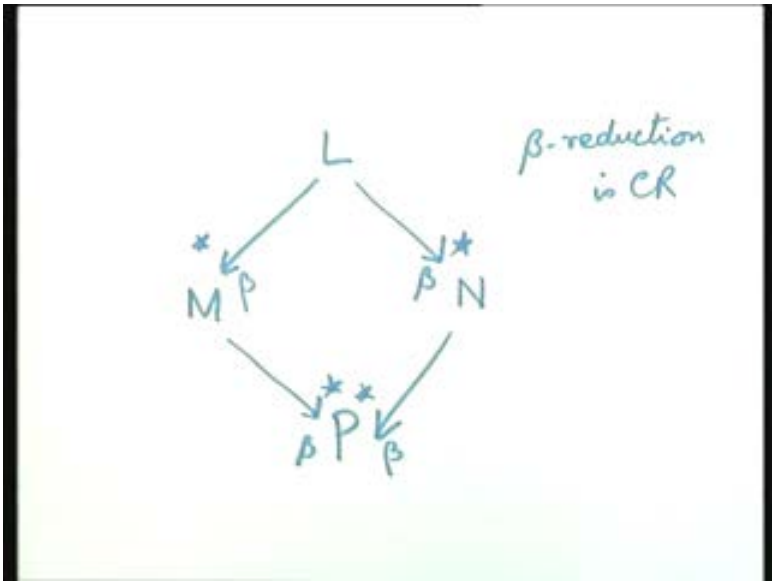
However, this example shows that the one step beta reduction does not satisfy the diamond property. So what it means is that it becomes some sort of a technical matter now because of such pathological examples. There exist such pathological examples which prevent the one step beta reduction from giving you a perfect diamond. So the normal proof of the Church-Rosser property for beta reduction would be that you allow for a parallel evaluation of many steps of the original beta reduction to be regarded as a single step and then complete the diamond.

All you need to know is that beta reduction is Church-Rosser. Therefore for any term all possible reductions which do terminate should yield the same normal form. Since beta is Church-Rosser what it means is that you cannot have two distinct normal forms. So for any term through a many step beta reduction you have obtained a normal form M and through some other fashion you have obtained another normal form N and if beta reduction is Church-Rosser then what it means is that it is possible to complete this diamond which contradicts the whole issue of normal forms.

So, if there are indeed two distinct normal forms then by the Church-Rosser property there would be a common P such that these two are related by many step beta reductions. But if these two are normal forms then there cannot be any more beta redexes which means M should be the same as P, M should be alpha equivalent to P and N should be

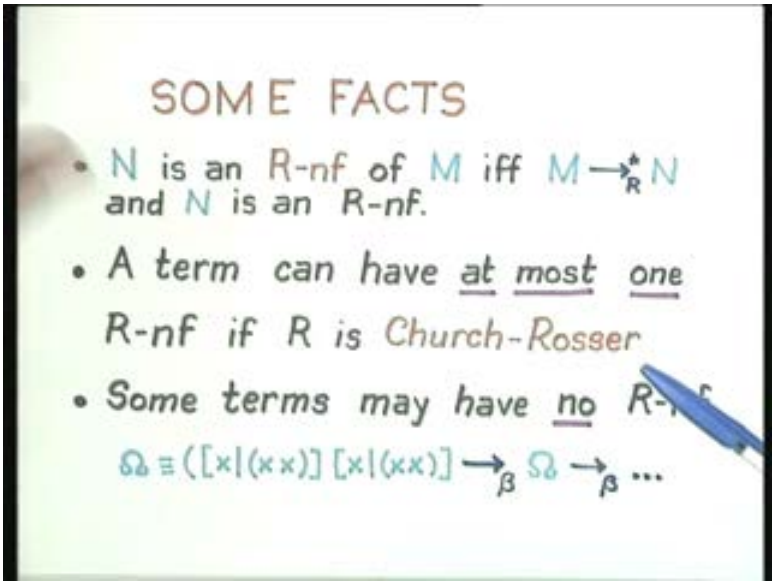
alpha equivalent to P which means M and N should be alpha equivalent to each other. So if our basis of reduction satisfies the Church-Rosser property then it can have at most one normal form, there cannot be more than one normal form and that is the important thing.

(Refer Slide Time: 10:24)



However this does not guarantee that you can always find normal forms.

(Refer Slide Time: 11:01)



In the previous example it was that all computations need not yield the normal form because some computations can be infinite. So some terms may have no normal form, there are other terms which may have a normal form only for certain computations but

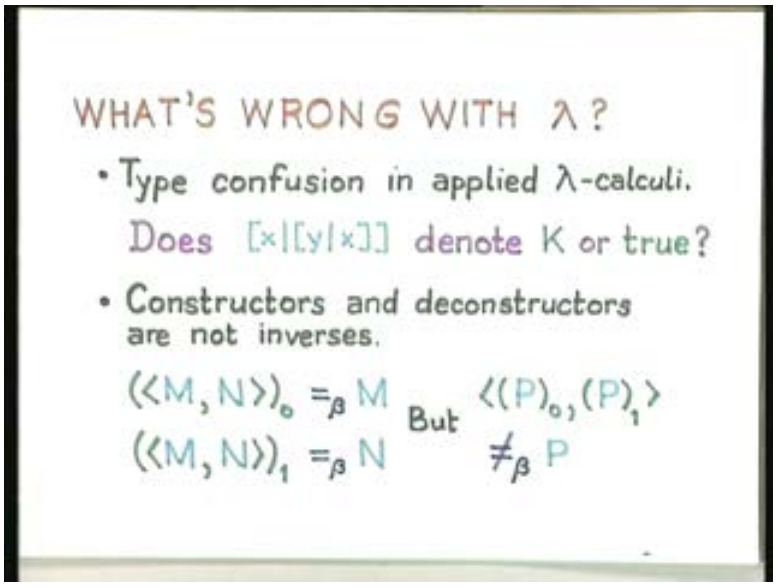
for all computations which terminate they will terminate in the same normal form and for the computations that do not terminate of course there is no question of a normal form. So what happens now is that this omega and the delta are the prickly issues.

And the nice thing about for example pioneer arithmetic and all our fundamental notions that come from school and that historically also came before all these is that they all had normal forms and they all had terminating computations. There were no non terminating computations in basic Peano Arithmetic and so on and so forth. So the important question then is what is wrong with the lambda calculus. There are of course several things wrong with it.

And if you look at it sort of more carefully and look back on whatever we have done then there are some of the things that are really wrong with it as follows. One is that if you actually apply the lambda calculus on to some other formulae, after all the whole idea of the lambda calculus is that you should have an independent formulation of the notion of functions which is applicable to any other domain where you want to build higher order functions from the existing functions. So the first thing that happens of course is in the pure lambda calculus, there is already a typed confusion problem in the sense that at any time if I take this lambda term does it represent the combinator K which is supposed to give me the first of two arguments so it acts like a projection function or does it represent the predicate true.

Thus, if I take the pure lambda calculus and I construct all my data structures, my Booleans, my integers and so on and so forth then I can do arbitrary kind of applications. There is nothing in the calculus itself which prevents me for example from asking whether a natural number is true. So I can actually apply true to a natural number and the most unfortunate thing is that it will actually give me some answer which I may not be able to interpret either in the Booleans or in the natural numbers.

(Refer Slide Time: 14:42)

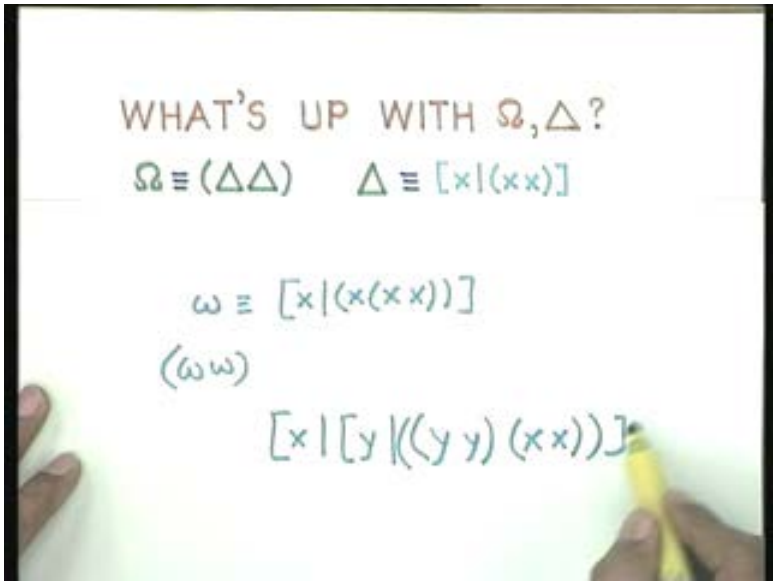


And secondly of course there is this problem of the various constructors and deconstructors that we have are not inverses of each other. While you are guaranteed for example for the pairing function that if you actually explicitly constructed a pair from two terms M and N and then you deconstruct them you will get back the two terms M and N in the separate deconstructors. But nothing guarantees that if you take an arbitrary term P and first deconstruct it and then form a pair you will not get back the original term.

We like to think of constructors and deconstructors as being some are inverses of each other. And if there were no type confusion then there would be no problem with this. For example if there was some typing then this deconstruction operation could not have been applied on any arbitrary term P, it could have only been applied on a term for which these deconstruction operations are actually defined. And if you can somehow put in those restrictions then what it means is that these constructors and deconstructors would also be inverses of each other. So what we require are really typed guarantees in the lambda calculus and especially in the applied lambda calculus.

Here K is this combinatory, and when you actually apply the lambda calculus on to some other domain then you know the whole thing becomes completely berserk, that is now you do not know what is actually happening when some faulty constructions and deconstructions are applied on those terms. And more over the applied lambda calculus itself might be applied on a domain which actually is a combination of two domains the Booleans and integers, integers and reals and Booleans and so on and so forth and then type distinctions become important. The pure lambda calculus when applied directly on those calculi is not going to give you the benefit of those typed distinctions. That was one thing that is important. So can we somehow preserve types when we apply the lambda calculus on to some existing domain? That is one important question.

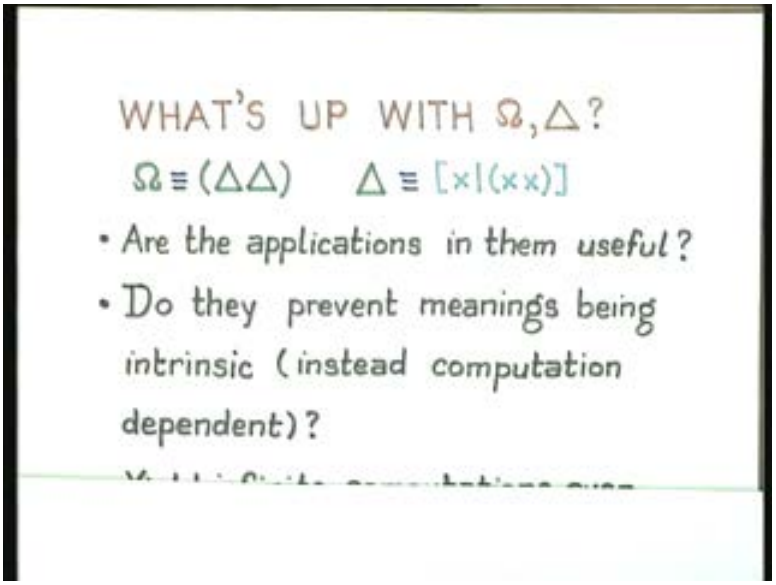
(Refer Slide Time: 18:44)



The second thing with this beta reduction and the non terminating computations in the beta reduction is essentially all our examples had to do with these two but these two are just the simplest possible examples. After all I could construct other equally bad examples like a new combinator omega of this form may be lambda x x applied to x applied to x and then I could consider omega applied to omega, I could take something that gives me four or five applications, I could give I could even take more perverse things like x y y y applied to x x and I could consider a whole lot of perverse things which when you finally boil it down to its basics you go back to the fundamental question; what was the lambda calculus originally intended to do? The lambda calculus was originally intended to somehow account for functions.

The basic notion of function and function application were really the main parts of its agenda. Then in the light of that basic question do these combinators make any sense. Just because my context free grammar for the language allows the generation of such terms I still have to ask this basic question; do these things make any sense to either man or beast?
Then what happens is that you really have to go back to your basic mathematics. And when you look at these combinators and try to interpret them as their effect, after all the whole idea was that we define function application and function abstraction in such a way that it is somehow supposed to intuitively capture the notion of unnamed functions, the construction of higher order functions and generally unary and certain isomorphism properties assure us that it is enough to consider only unary functions because any n-ary function can be carried into a sequence of unary functions where the arguments are taken one at a time so all that is fine, all that intrusion is fine but what is wrong is doing these things will be really be meaningful in an actual application. Now we come to a basic property which mathematics always followed.

(Refer Slide Time: 21:42)

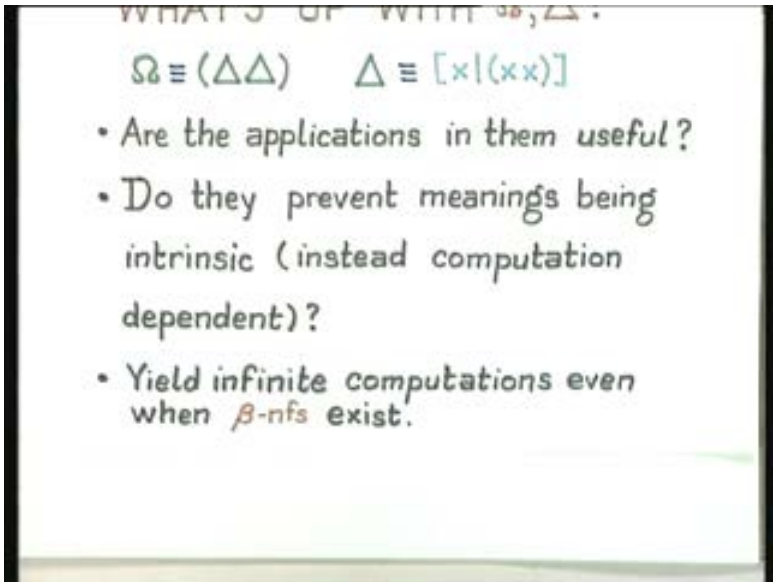


So the first thing is that it is not at all clear that they are really useful combinators. And from the examples that we have seen the second thing is that these things actually prevent meanings from being computation independent. And in particular what I am talking about are these kinds of applications. What is the specialty of these kinds of applications? Instead of being applications they are more like replications.

If you look at all these perverse combinators they are more like symbolic replications of some free terms. It is like a population explosion problem, it is hardly functional applications because no mathematician would ever make this kind of an application. So there is a replication problem where it is not even clear that that replication is meaningful. There are replications which are meaningful like the y combinator or the turing fixed point combinatory. But here it is not clear that these replications are really meaningful.

So the next question is supposing we can somehow get rid of these kinds of arbitrary replications then if you get rid of these kinds of arbitrary replications then you will be guaranteed beta normal forms. What would it mean is all computations would terminate and given that the beta reduction basis is Church-Rosser you would always get beta normal forms.

(Refer Slide Time: 23:11)

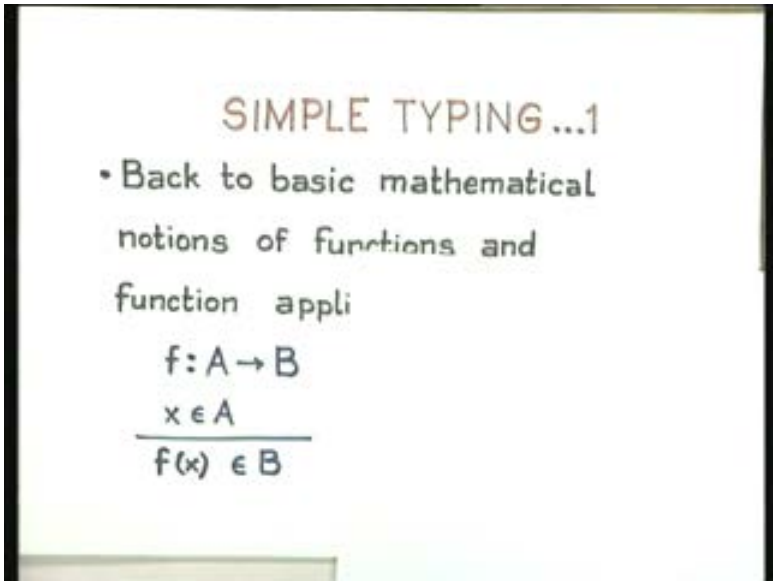


The important thing is that the meaning should some how be independent of computation and this kind of replication operator actually prevents the meanings from computation dependent. They complicate matters they yield infinite computations. I would not mind if you yield infinite computations always you know a certain term regardless of how you apply it regardless of how it is beta reduced it always yields an infinite computation, perfectly fine then its intrinsic meaning is that of something that is undefined, I can interpret it that way.

When it is a bright and sunny day it gives me a beta normal form and when it is a stormy weather it gives me an infinite computation then there is something wrong with what I can ascribe to it as a meaning. The meaning becomes dependent on the computation rather than on the function itself as an object. So the meaning should not be either computation or weather dependent. At least when beta normal forms do exist I should be able to guarantee somehow that there are no infinite computations and I might be willing to relax this provided these things actually had some meaning but it is not clear to me that in any mathematical domain these combinators actually have some meaning.

Now let us look at some simple typing schemes. What we do is we go back and look at our school mathematics textbooks. So we get back to the basic mathematical notions of functions and function applications. And looking at these functions and function applications in my 9th class Mathematics textbook I get this.

(Refer Slide Time: 25:46)



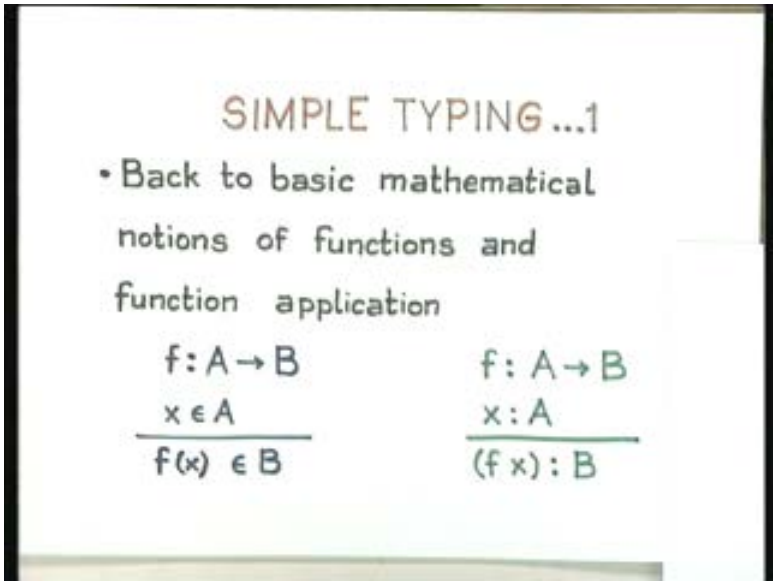
This is what my mathematics textbook says and this is what I have abstracted out of it. If a function is defined from a set A to set B and x belongs to A then f(x) belongs to B and this right from 9th class upwards in all mathematics textbooks I have abstracted that this is the common behavior of all the functions.

Remember that the lambda calculus is supposed to abstract common behavior of all functions across disciplines in any discipline that uses mathematics. So going through my Physics, Chemistry and even my Biology texts I find that functions when used as mathematical functions they satisfy this property. But still functions are as good as values as others in that way I differ from those textbooks, they regard functions as being some separate kind of objects and these values being some separate kind of objects somehow distinguish between the two and say that f applied to x is a value it is not a function, do not ever get confused but the point is that I do not want to make those distinctions as far as possible. Then I look at this and I decide that this f is really as much an element of this

set of all possible functions from A to B as this x is an element of A, there is really no distinction between the two and the same is true about this.

So if I unify functions and values what I get is an inference rule like this. I treat all functions also as values so then I treat f as just being a member of this domain and I treat x as being a member of this domain and I treat function application therefore as being a member of this domain.

(Refer Slide Time: 28:06)



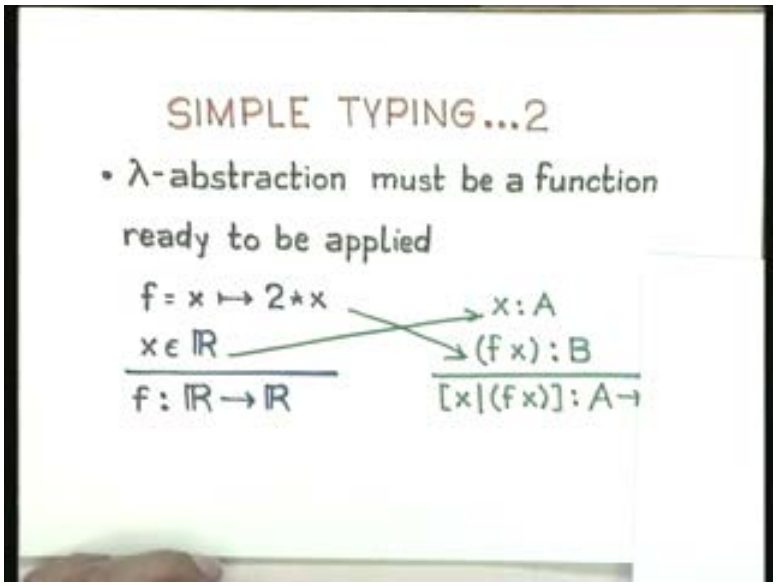
Of course I have cleaned up function application and so on and instead of this standard mathematical text now I use this as my function application. That is the first thing. The second thing is that the lambda abstraction somehow must be a function that is ready to be applied. Thus while still trying to reconcile functions as values I still have to keep in mind that eventually my lambda abstraction is a method for defining unnamed functions which on application yields some values. So the lambda abstraction must still be a function that is ready to be applied.

This (Refer Slide Time: 29:11) is not something that I find in any school or college mathematics textbook. But if you go into the higher Maths textbooks very often when they draw these mappings from one topological space to another they often write a function in this fashion. It is actually very close to the lambda abstraction notation and they have done it without knowing it. What it says is, you consider pull backs from open sets to open sets or some such thing and you loosely say that it takes x to some 2x or some such thing.

So the fact that this function takes x to 2x this is a function that takes x to 2x for any x which is a real and what you infer from that mathematical text is that therefore this function must be a function from reals to reals. And here actually the mathematician has come closest to define a lambda abstraction without confusing it with application. Right

up to M. Sc level they have always confused the abstract notion of a function from its application just because in order to define functions they had to define the effect of its application on something and so they very often have confused functions with their applications.

(Refer Slide Time: 31:08)



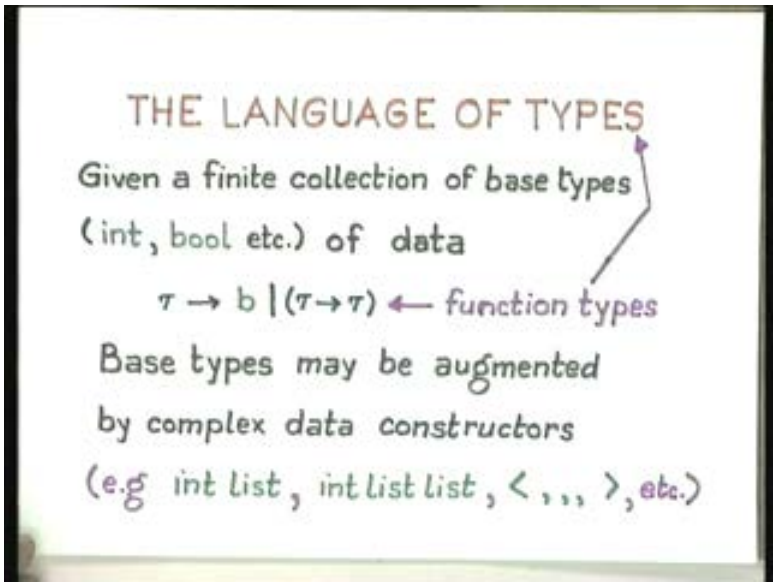
Therefore if I have some x belonging to A and the effect of f applied to x, I have inverted these two by the way, this corresponds to this, the effect of f applied to x gives me a value in B then this abstraction this thing abstracted out is really like a lambda abstraction of this form then this whole thing must represent a function from A to B and that is the simplest possible typing you can think of in the lambda calculus. So essentially what we do now is we formalize this again into a language. So these notions are what we are going to formalize into a simply typed lambda calculus. So here is the language of types.

We will assume that there is a finite collection of base types. If you want to look at its parallel with M L actually it is no longer just a finite collection of base types. there are there are constructors and deconstructors in M L for example which allow you to produce int list, int list list, int list list list list and bool list, bool list list, int star bool, int star bool star int star bool and so on and so forth. So the base types are not really a finite collection because they are new collections of data.

And since you have constructor operations for the data, for forming tuples, for forming records, for forming lists and so on from simpler types what you actually have in terms of just all the base types, again when you distinguish data from functions what you have is actually an infinite collection of different base types. You start with a finite collection of really primitive base types and you apply these data construction operations to construct complex data from simpler data and you actually have an infinite collection of base data types.

“Data types” is another technical word which you should not confuse with this and then what you do is you construct functions on these data types. So we will just assume a finite collection of base data types for the purposes of examples just int and bool and then what we have is this simple notion of function types. So if b is a base type let us say integer or bool, it is a type it is a form of function type and given two types tou 1 and tou 2 tou 1 arrow tou 2 is also a type. This is the simplest language of function types that I can think of. Therefore, what happens is that if you actually have nice constructors and deconstructors for the data types you can relax this condition of finite collection to infinite collections which are generated from finite collection. You can give a grammar for this if you like but let us just look at the function types for the moment assuming just these two base types.

(Refer Slide Time: 34:48)

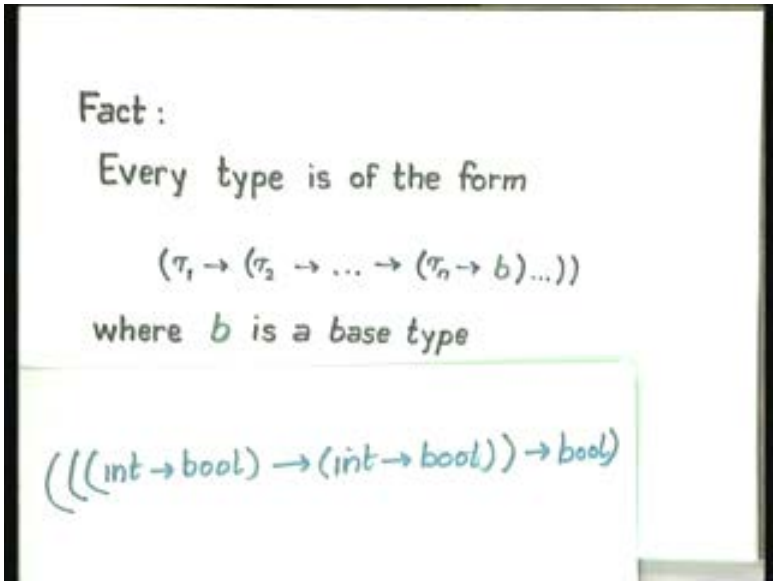


What we are going to do is we are going to some how incorporate this language of types into the pure lambda calculus. I would like to study the theory of the pure lambda calculus assuming some base types only then I will know how to apply it to an applied lambda calculus. So do not take this int and bool to be very serious, we are not having boolean algebra and Peano Arithmetic and all that will just complicate. What we want to look upon is we still want to look at the lambda calculus assuming just some finite collection of base types let us say b_1 to b_N.
What is the effect of typing on the lambda calculus?
This is the language it is a context free grammar and essentially all I am saying is every base type is a type. If tou 1 and tou 2 are types then tou 1 arrow tou 2 is another type. It is a non terminal symbol here. This is a context free grammar, it is a generation rule.

The whole point is that this grammar allows you to construct types like this. Then here if you look at it from the point of view of this grammar then this right hand side tou is bool and this left hand side tou is this whole thing which again has some right hand side of type int arrow bool and left hand side of type int arrow bool and so on and so forth.

It is a generation process; you can generate very many different function types. Something that you can prove without too much problem is really that every type is of this form, tou 1 and tou 2 and so on might be there but eventually there has to be a base type somewhere inside here. It has to be something of this form, this thing confirms to this form.

(Refer Slide Time: 37:58)



So these things could be various bracketed types with various arrows and nested arrows and brackets and so on and so forth. So all these tou 1 tou 2 tou 1 could be any kind of types generated from this thing but eventually they all yield a base type because you are applying the lambda calculus onto an existing something.

Your typing discipline is somehow determined by the base types that are of interest to you in your application. This is something that you can prove by induction on the generation process on the context free grammar. Then what I will do is I will define a static semantics. So what I am going to do is I am going to say now that the lambda calculus is originally meant to capture the notion of functions, lambda application is supposed to capture the notion of application of functions of appropriate type to appropriate arguments to yield values or functions of appropriate types.

I am not going to permit any application that violates these type constructs. That is how I would be able to apply this lambda calculus to some existing application in Mathematics. So we have our type inferencing rules of this form. Of course we can have very complicated expressions so we have to see what happens when we go deep inside some lambda expression in order to find the type.

So in general I will assume that there is something called a context available. given a whole lambda expression or some application in isolation the context is going to be some empty set but assume that you go deep into an expression you will be building up your

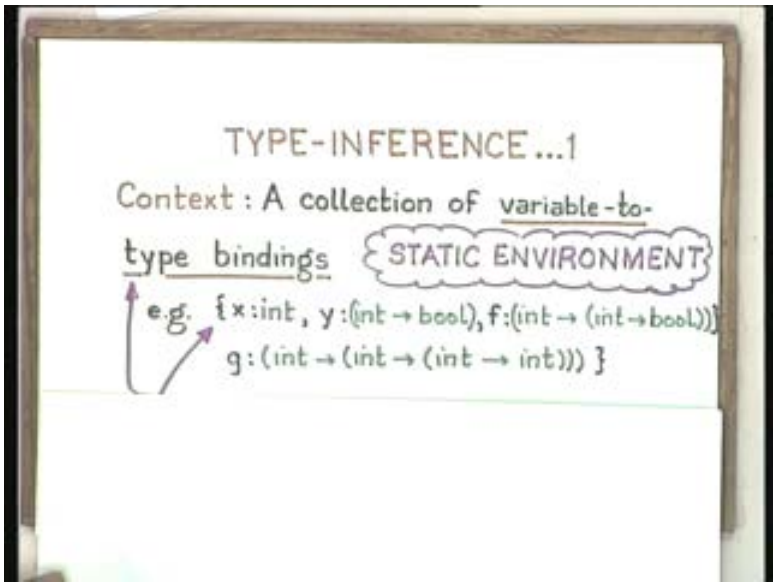
context some how and you will be building up this context very much like the way we built up environments.

An isolated program executes in an empty environment but as you go through the nestings within the program you have the environment building up with more and more declarations. In a very similar manner we are going to call a context a collection of variable to type bindings. so the types are like the values or variables except that they are not really values they are now types, they are not just values of the variables so this is what I would call a static environment as opposed to the environment that we have already done which was a dynamic environment which actually represented activation records at run time. This is a static environment because all these can be done at compile time at translation time without executing. If it is going to be done at translation time remember that there are no values available to the variables yet. Values become available to variables only at run time. So this is a static environment and this static environment is typically a symbol table that is constructed during the process of compilation.

So if you go through your page zero compilers or some such compiler there is a building up of a symbol table and that constitutes the static environment. So the variables have types but they do not have values. So we will create these variables to type bindings and what we are going to do is since we are doing everything at translation time, this is all translation time that means just from the syntax of the term we should be able to infer types because we cannot do application.

An actual application is like an execution so the whole point is that we should not permit applications unless you have the right types for those. Unless you have a function of a type and an argument of type which corresponds to the type that the function takes as an argument that checking is on the thing that is done statically at compile time.

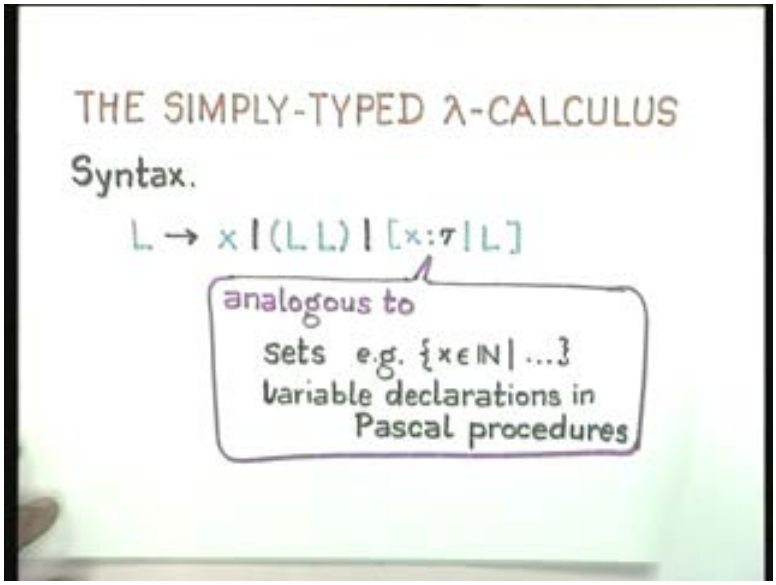
(Refer Slide Time: 43:34)



So our context will typically consist of a collection of lots of names basically names to type bindings. The static environment is too general a term. In the actual process of translation it might actually be bound to values or so in the case of identifiers which are constants. Hence static environment is a much more all encompassing term.

What we are interested at this point is just a type environment, just a collection of variable to type bindings. So now what we will do is, remember that we have this language of types, we are going to do syntatically sugar up the lambda calculus to use those types somehow and that gives us Church's original simply typed lambda calculus. Here is the grammar for the simply typed lambda calculus.

(Refer Slide Time: 44:54)



All variables are lambda terms, I can apply two lambda terms, but here is where here is where the problem is that is something we have to fix. I can do abstractions but now in these abstractions what I ensure is I ensure that every variable in the abstraction has a type associated with it in the grammar itself. And this actually analogous to the way we often define sets.

For example, the set of all x belonging to N such that some properties and some stuff, this is very much like that and of course it is very much like your variable declarations in Pascal procedures var, x, colon, integer, semicolon, begin and then you have the body. So this notation also just carries forward our original set of theoritic notation which was analogous to procedure declarations in Pascal and so on and so forth it carries through here. So now what we have is just a simply typed lambda calculus where every abstraction has the bound variable declared to be of a certain type.

In Pascal programs this type is one of your base types. In the case of the lambda calculus it need not be a base type. It could be a type constructed from that grammar of types, it could actually be a function type and that is how we get higher order functions. Here this

type is really whatever that can be generated from that grammar and it is not restricted to being a base type. Then this way we get higher order functions from lower order functions by abstraction.

Why we have not we taken any base type to x is because the syntax of the lambda calculus is such that if x is a term which is anyway a term, it is free where x has x itself as a free variable which means that it gets its value during execution from some environment in which it is somehow applied then it will also get its type from that environment.

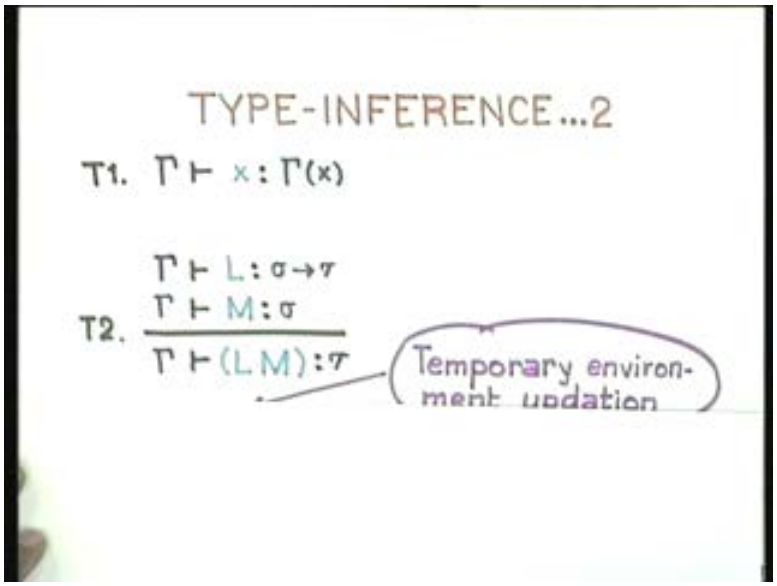
What is the idea of a free variable?
The idea of a free variable is that it somehow gets its values from somewhere outside that you do not know. If you do not know its value, if you are not going to know its value when you look at it at isolation you are also not going to know its type, for all you know the type may be determined by the value that it obtains from the outside world.

However, when you have local declarations then you are specifying this x. If this x occurs free and if you actually know the context then actually it is going to be bound somewhere so there will be a declaration which has its type specified. We are not able to find the exact type definition mechanism. A closed program has types assigned to every identifier so we are mostly interested in closed programs only they will really be able to execute. But for the purpose of getting congruencies and so on we require to also consider non closed terms so that whatever we say for non closed terms also applies to closed terms later.

Remember the compatible closure of beta reduction and so on and so forth we are taking even non-closed terms because when you compatibly close it in context those properties are carried through that is the whole fundamental purpose of having pre-congruencies and so on. You consider its you consider things with free variables and isolation and whatever properties you can prove for them will also be able to satisfy those properties when they are placed in context and closed. This is something that happens in any logical language too. There is an assignment of truth.

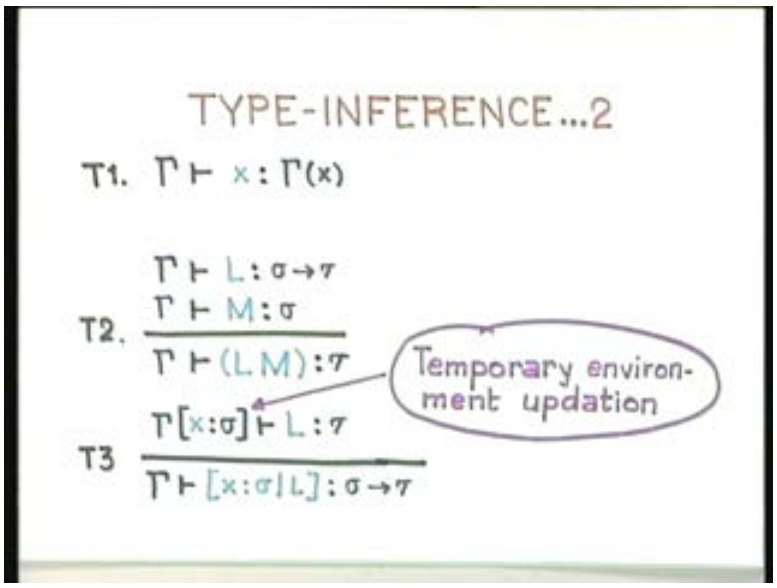
For example, in a first sort of logic you assign truth values, you have free variables you do not know what truth values they are going to have but you do not care. They get their truth values from somewhere but you what ever you can prove you will be proving for both truth values for the free variables. For the bound variables you are not going to do that kind of assignment. Whatever you can prove for both possible truth values then it is going to be true for whatever properties you prove regardless of what the truth value of that variable is.

(Refer Slide Time: 52:53)



You can also prove when the variable is closed up some how when it occurs in context. It is like being able to look at a program segment in isolation from the rest of the program and trying to find some values semantical equivalence. Semantical equivalence we define as, over all possible states in all possible environments if the two constructs yield the same results then they are semantically equivalent.

(Refer Slide Time: 53:33)



Therefore you are considering all possible bindings if you can prove it in a general form then you can prove it any context. Then in any context where one of those program segments is present you can replace it by the other. Otherwise your semantical

equivalence becomes context dependent and it is no longer context independent. This is the modified language of lambda calculus and the type inferencing just goes as follows. We have the three rules for the syntactical construction of lambda terms.

So, for any variable x if it actually appears in this context then it is typeable as being of this whatever is the type assigned to x in this context grammar, here is your context coming. So if the x is really free then you cannot assign any type to it. If it does not occur anywhere in gamma you cannot assign any type to it. And for application if you can infer in from the context grammar that L has the type sigma arrow tou for some sigma in tou in the language of types and you can also infer from the context that M has the type sigma then you can infer that this application has the type tou. And if one or more of these inferences is invalid then you cannot infer any type for the application. This is really one of the rules we abstracted from our school mathematics.

And then for the lambda abstraction it is very simple. The type of an abstraction in a context grammar is obtained by adding this to the context temporarily and then trying to find out what is the type of this term L. Therefore by temporarily attaching x of type sigma to the context updating the context with this type if I can infer that L has the type tou then I infer that this lambda abstraction has a type sigma arrow tou. Going back to our agenda you will notice that the whole idea is that a lambda abstraction should really look like a function should behave like a function should have a type of a function so it has to have an arrow in it. So we will do some examples later which illustrates this type inference, for example, why delta omega is not well typed and what it leads to.