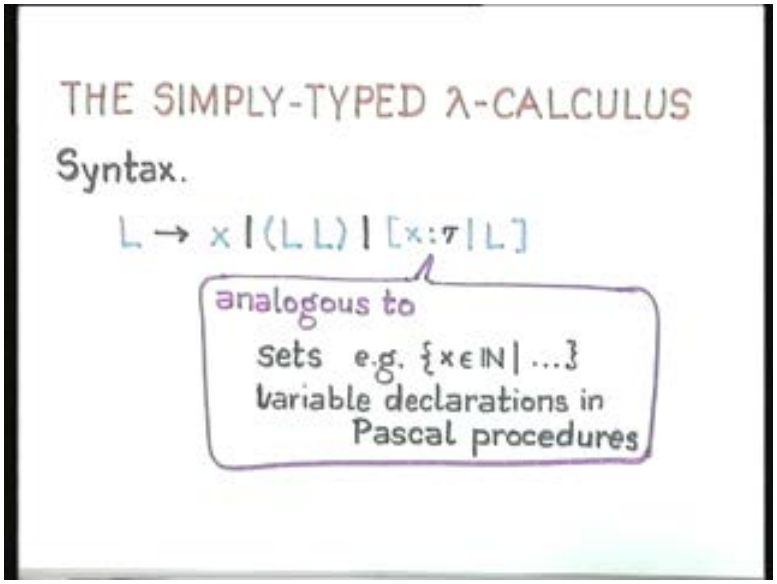
Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 33
Polymorphism

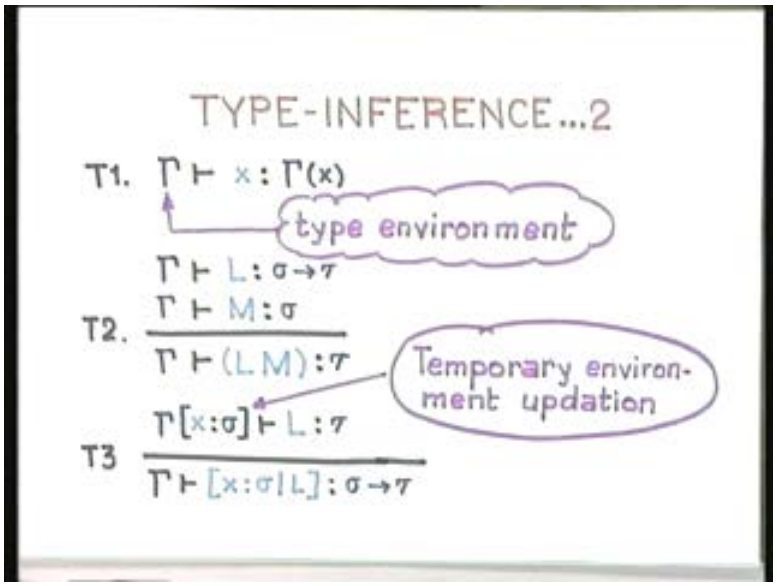
Welcome to lecture 33. So I will just briefly recapitulate what we did about monomorphism and then go on to polymorphism today. So we defined the language of this simply type lambda calculus where there is a notion of some base types and the construction of the construction of higher types from the base types. And these base types which for simplicity I took to be integer and Boolean are really what might be called type constants. When you give something the name integer in the domain of types it is a constant. Now the significance of that will become evident slightly later but it is important to remember that they are type constants. So you can construct complex types using the simple language that we gave for constructing higher types from these type constants.

(Refer Slide Time: 1:36)



So essentially what it means is that all your types will be of the form, if int and bool are your base types then you will have types of this form int arrow int, int arrow bool may be you could have for example higher types like int arrow int arrow bull arrow bool and so on. What I mean is all your type expressions are going to be of this form, all your type expressions will have the names of the base types always occurring in it and there is nothing else to it.

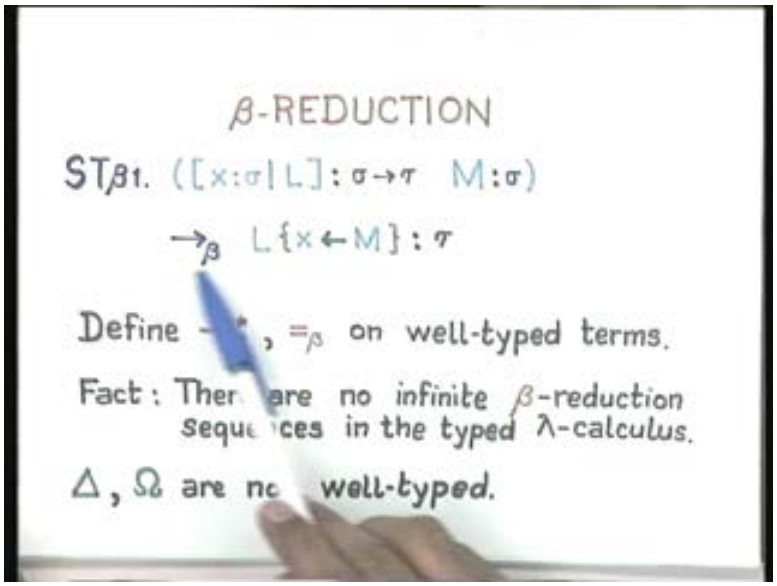
(Refer Slide Time: 2:47)



Then we gave type influencing system for this from which you can prove that every combinator therefore or every lambda expression in the simply type lambda calculus actually has a unique type based on the type expression based on these inference rules. So, given that a variable has a certain type, a variable could have a higher type also remember that, we are treating functions and values all as equal objects.

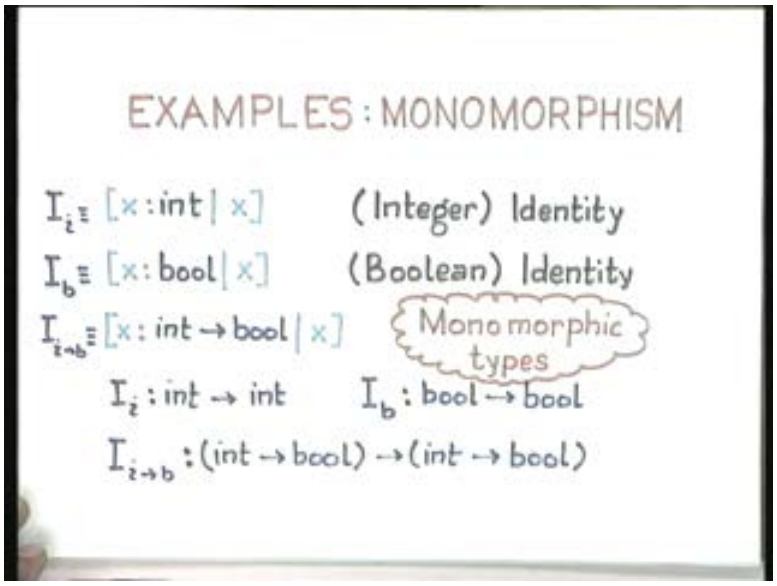
For example, this x could be of type int arrow int which denotes that it is a function from integers to integers and you could infer therefore the types of all the lambda expressions in the language which are typable and if they are not typable then of course it does not belong to the simply type lambda calculus.

(Refer Slide Time: 4:04)



The beta reduction is of course modified to take typing into account so that only if you have a lambda abstraction which is typable and something arrow something as sigma arrow tou and you have an operand which is of type sigma assuming that these types can be inferred therefore they are well-type terms and then you can perform a beta reduction and get a value of type tou. We looked at some of the examples dealing with the identity functions.

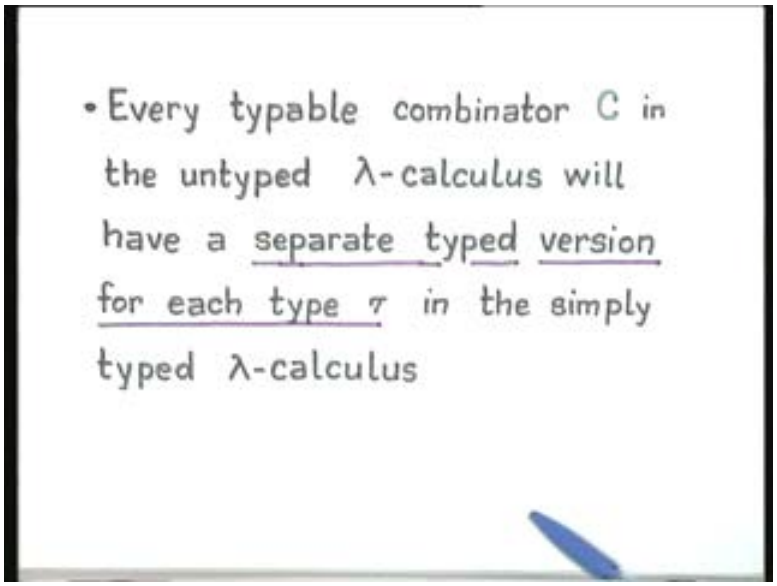
(Refer Slide Time: 4:34)



What we noticed at this point is that every combinator is useful functions because every combinator has a unique type which is built up from the base types. Therefore for what

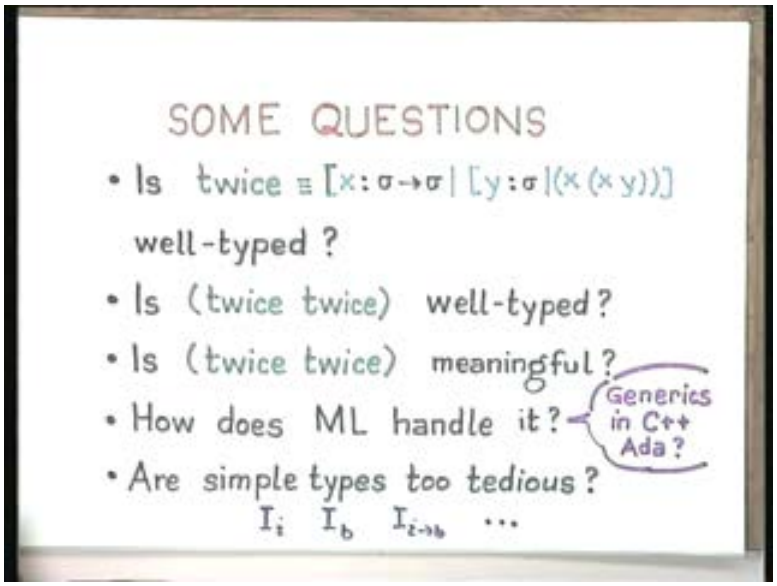
used to be typeless combinators which could be applied anywhere in the untype lambda calculus for each one of them you have several copies depending on the type of application. So even a simple identity function has an incarnation for integers, has an incarnation for Booleans, has an incarnation for all functions from integers to Booleans so for every tou that you can think about there is a separate identity function I_{tou} and this is because of the unique typing feature of this simply type lambda calculus. So, for each type tou the analog of the combinator C in the untype lambda calculus when you move it into the simply type lambda calculus for each type tou you will get a combinator C tou which respects that typing.

(Refer Slide Time: 5:51)



Of course we ask this question whether we are actually being too harsh by using a simple typing scheme. And one thing of course is that these simple typing schemes were used in some of the older programming languages like Pascal and Modula and so on and so forth.

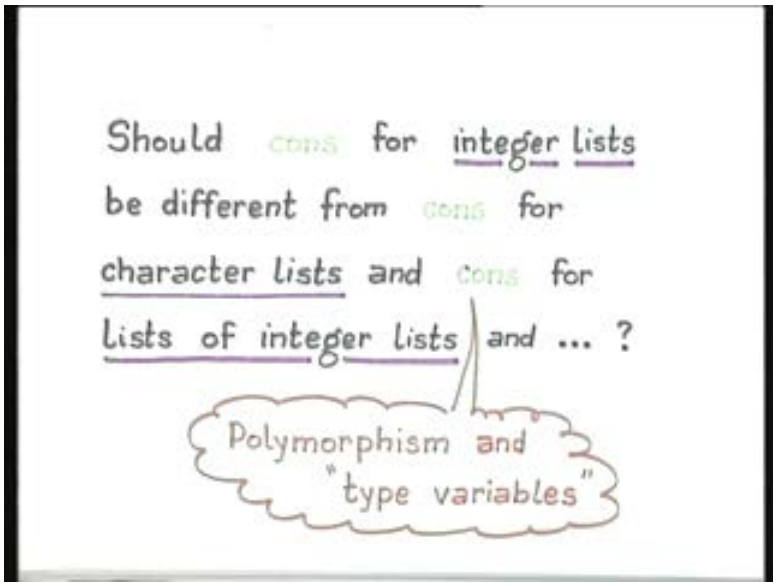
(Refer Slide Time: 6:04)



And of course there are some questions; we actually went through the computations of twice applied to itself and so on and we found that some of them could actually be given a meaning. So, not all self application really is meaningless. And secondly the point is that if you take any combinator C is really too tedious to have so many different copies of it for each, you have a combinator C tou for each possible tou, the combinator could be some complex program. So it is really too tedious to have something like this. when we think of an identity function what we are actually implying is that it is a higher order function which given a function of any type tou returns you the same function actually but returns you a result of the type tou. So the identity function regardless of the type for which it is meant actually is one really a higher order function which could be parameterized on the type.

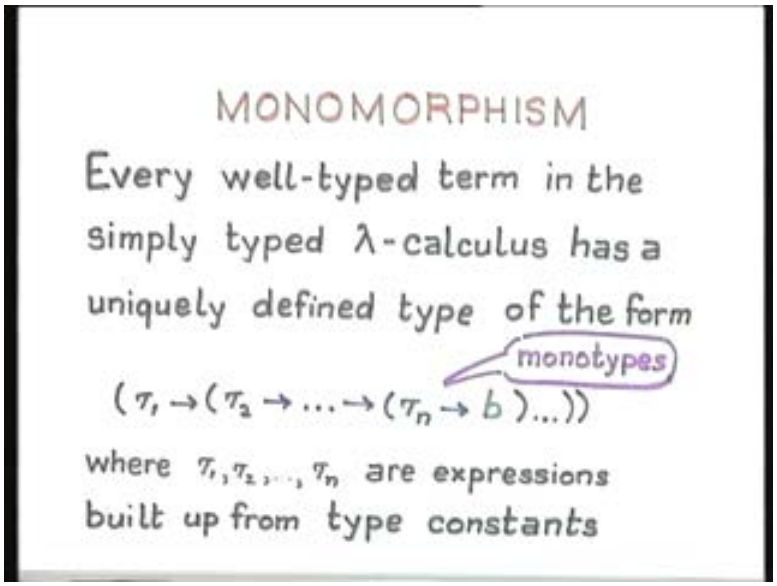
So we can talk about I the identity combinator I being parameterized on a type tou and the result should be I_{tou} which is the appropriate combinator for values of type tou. By values I also include functions. It can be even called as a parameterized typing. So what we should be able to do is we should be able to take this identity call a general identity function I and parameterize it on the subscript and that is what polymorphism is about. Then we have an identity function which actually takes a type itself as a parameter and then specializes to that type. And as we saw there are lots of functions which we use in all our M L programming and so on which really are of that type; the head and tail functions, the cons function, the map function they are all in that sense polymorphic in the sense that the actual function is not very crucially dependent on the underlying base type from which your data type is constructed.

(Refer Slide Time: 8:42)



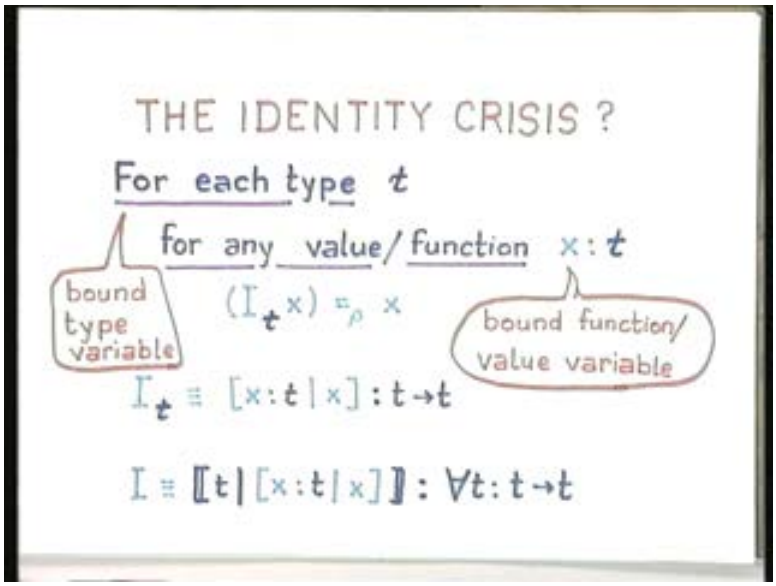
The function remains more or less unchanged except for the type for all possible kinds of arguments that you might get or at least for a class of possible arguments. For example, you cannot apply cons between two integers but there is a class of arguments namely integers and integer lists, Booleans and Boolean lists, may be integer to integer functions and lists of integer to integer functions and so on. There is a class of objects for which cons is going to have essentially the same representation and our intuitive meaning of cons is just that given some argument of type tou and given another argument of a list of elements of type tou you should be able to perform a cons. The implementation or the meaning should not significantly vary with variations in the underlying type tou. So the monomorphism actually has this real problem that you cannot adequately parameterize.

(Refer Slide Time: 10:15)



Hence we generalize monomorphic types so that they give us this general flexibility. That means in addition to these type constants which are the base types we also allow for type variables and that is intuitively even like how we look at the simple function like the identity function. Thus what we are saying by a general identity function is that for any type t and for any value or function x which is of type t the identity combinator I_t when applied to x is some how beta equivalent, actually it should beta reduce in many steps may be to x itself. Now what we can do is if you look at the lambda abstraction as we have looked at it I told you the analog of the lambda abstraction which sets the lambda abstraction also has its analog with universally quantified objects and this is essentially a universal quantification.

(Refer Slide Time: 10:50)



So if you look at a lambda term, let us look at the identity combinatory, what we are essentially saying by this bound variable x is that for any x return back the value of x. In the case of sets what we were saying is if you had a predicate here let us say p dependent on x then for any x such that the predicate p(x) is true. So the set notation, the lambda abstraction and universal quantification are all very similar and we will use this fact.

So if you were to actually read whatever we have been saying now a good way to read it is as if it is a universally quantified object. For each type t and for any value or function x of time t $I_t(x)$ should be beta equal to x. So what we did is that we stopped with this abstraction here in our simply type lambda calculus we just translated this abstraction into the combinator I_t. But if we go beyond this abstraction and also into this abstraction then what you get is a general combinator I. You can read this is for any t and for any x of type t return x. The universal quantifier and the predicate logic is really like lambda abstraction but I am just using the universal quantifier.

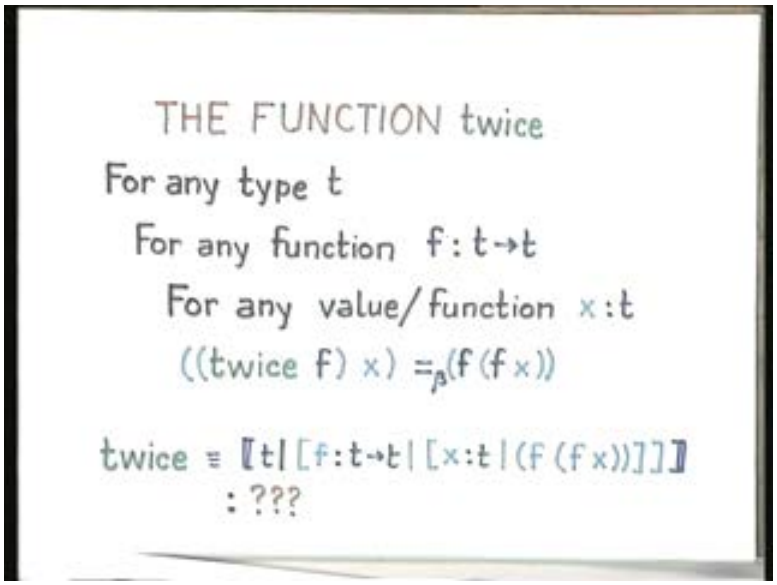
So the generalized identity combinator that we are really looking for really has a type which is given by a universally quantified type variable. Here you have got a case where there is a variable which is bound by this universal quantifier. Again this is like a local declaration, this is like a predicate if you like but this local declaration clearly specifies that the identity combinator has a type such that for any type t it has a functionality t arrow t where I do not really care what value you give this t. If you are generalizing from monomorphic types then what you can say is that this variable t may take any value from the monomorphic types defined by the language of types.

For example, this t could be a value like int, could be bool, could be int arrow bool, it could be int arrow int arrow bool and so on and so forth. Any of the types that we have so far defined in our simple type structure could be a value for t and that identity combinator will appropriately take those values t. We started out with variables and constants way

back in the Paleolithic period where constants and variables really took values from underlying domain of values. Then we got unnamed functions which are also variables. We generalized thee notion of variables to untype functions where the functions could take values from function spaces and underlying domain which is function spaces. Now we are going further and we are taking the domains themselves, types are really those domains and those domains are fixed so far and now we are generalizing them we are saying you take a variable which takes values which are the naming of particular domains.

The domain int arrow int could be a particular value of this variable t and this is like a universal quantification so if this lambda abstraction is like a universal quantification then lambda application or a beta redex is like universal instantiation. You have studied this quantifier elimination in introduction rules so lambda abstraction […17:20] quantifier introduction and beta reduction is quantifier elimination or universal instantiation. So you can instantiate the t here by any particular type constant that you like or any type expression which is built up only from type constants.

(Refer Slide Time: 19:40)



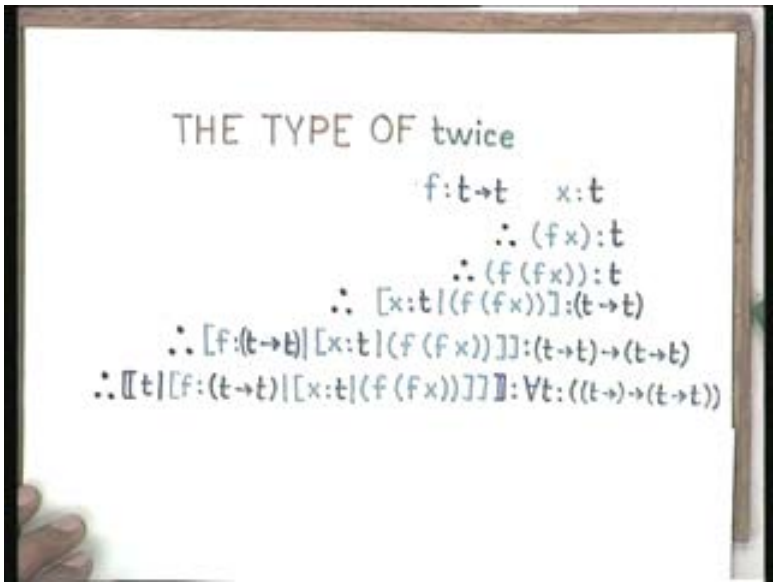
So you can talk of type variables which actually take values and those values are type expressions built up from type constants and that is what polymorphism is about. So this is what is known as a parametric polymorphism and this is the polymorphism that is present in ML. Let us look at the function twice. So the reason twice is meaningful then, is really that it is a function of this form. If you look at the definition of twice I can say that for any type t for any function f of type t arrow t and for any value or function x of type t twice applied to f applied to x should be beta equal to f applied to f applied to x.
This is the basic fact we know about twice and now go back upwards and do the abstraction. So what you get is f applied to f applied to x, you perform the abstraction on x to be of type t then you perform the abstraction on f to be of type t arrow t and so far it is really like the simply type lambda calculus. And then actually we have yet another

abstraction a universal quantification over the type t. So this essentially says for every type t and for every function f which has a type t arrow t and for any argument x which has a type t return the result f applied to f applied to x.

So what is the type of twice?
The point is now that now we have two kinds of beta reductions. If lambda expressions are really like universally quantified predicates then beta reduction is like universal instantiation, then an universal instantiation is like beta reduction which means when you universally quantify on types and instantiate those types you get a form of beta reduction also for types in addition to the beta reduction that you have already for the lambda expressions. As you can see things can get a little airy at this point.

We have variables, we have type variables, we have constants if you are applying the lambda calculus onto some domain and you will also have type constants and then you can have instantiations of those variables, instantiations of those values, instantiation of value variables, instantiations of type variables by type expressions, by expressions of the application and you will be inferring types so types and values also look essentially the same. You are going to have a beta reduction for quantified type expressions which is like another lambda expression. Only this lambda abstraction is on types and it is not on values. And of course values are the same as functions whatever may be the order but types are different. But still types also follow essentially the same discipline of quantification, of application, beta reduction, universal generalization, universal instantiation, quantifier elimination, quantifier introduction and everything.

(Refer Slide Time: 24:05)



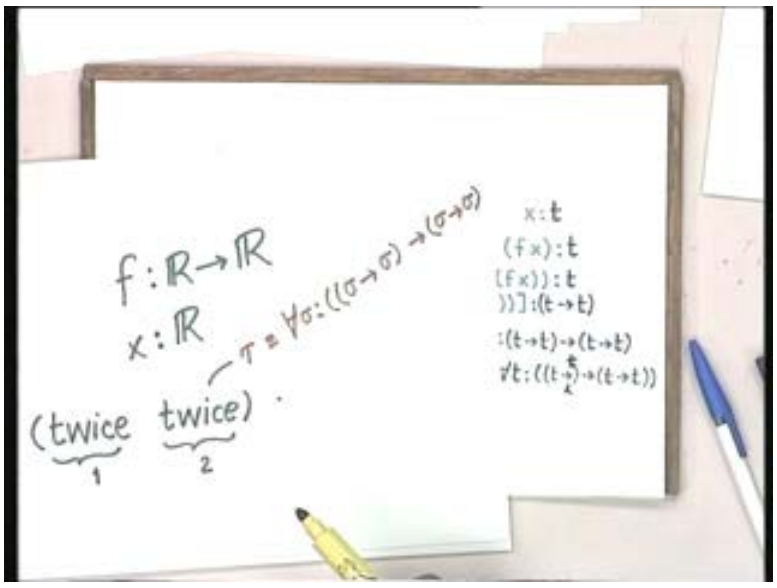
I will just assume that you already know the inference rules, you can get the inference rules by analogy. Thus by structural induction on the inference rules that are predictable now you can actually infer the type of twice in this fashion. So given that f is of type t arrow t and x is of type t, f applied to x will be of type t, f applied to f applied to x will be

of type t too because of this f is of type t arrow t and fx is of type t therefore f applied to fx will be of type t then when you perform the abstraction over x I am doing this bottom up of course but that is always easier to understand. This is not necessarily how the machine will do it, how your compiler will do it.

Note that all these have to be done by the compiler. So it will do it by a structural induction you can assume for practical purposes it will be doing it by recursive descent parsing method as part of the parsing process, before the code generation you do the type determination in order to decide whether the code has to be generated at all so you will do it in a recursive descent parsing fashion and come up. Then this abstraction gives you this type, then the abstraction over f gives you this type and the abstraction over t gives you this type. So essentially what we are saying is that functions like twice which are actually meaningful they are meaningful even under self application because when twice is applied to twice, and since twice is polymorphic its type is universally quantified.

We can always choose a type for the operand twice and generalize it so that the operator twice has a higher type than the operand twice. This is in fact what normally in Mathematics. When you apply one function to another the operator always has a higher type than the operand. Since twice this polymorphic this twice when you apply twice to itself this is the operator twice and this is the operand twice. The operator twice is of a much higher type than the operand twice.

(Refer Slide Time: 27:19)



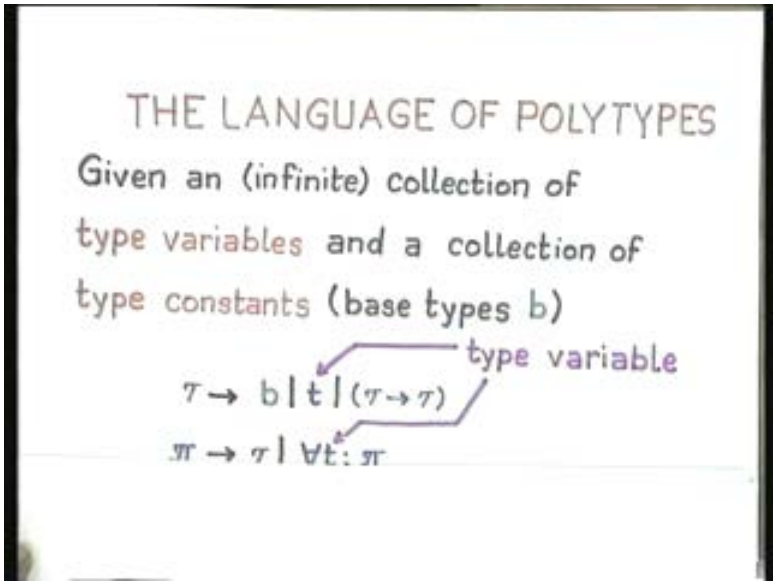
So if you assign to twice a type tou that type tou would be an expression of this form universally quantified, a universally quantified type expression of this form then if this has a type tou where tou is some for all sigma such that, here it must be t arrow t goes to t arrow t (Refer Slide Time: 26:38) so for all sigma if this twice has the type for all sigma sigma arrow sigma arrow sigma arrow sigma then the result of applying this twice to this twice should give you an element of type tou. Then this twice has a type which is really

given by tou arrow tou arrow tou arrow tou on application. Treated in general it has the type and that tou arrow tou arrow tou arrow tou has to come out as a particular case of this sigma by a suitable substitution process.

Essentially an expression is polymorphic if it can actually have different types depending on the context in which it is applied. Therefore the application of "twice to twice" is meaningful provided the operator twice has a higher type that is compatible with the twice of the operand. A particular case of this is something that you can see in any standard book on polymorphism or programming languages. So what we will do is let us formalize these notions. So we have the language of what I might call polytypes.

Now let us see the language of polytypes as opposed to the language of monotypes which is what we did in the simply typed lambda calculus where firstly you assume an infinite collection of type variables and a collection of type constants and these type constants usually consist of the base types which you are going to start of with. So let us say integer and Boolean and then firstly we build up the monotypes in the same fashion that we did for the simply type lambda calculus. If b is a base type then b is also a monotype and if tou 1 and tou 2 are monotypes then tou 1 arrow tou 2 is a monotype. In addition you allow type variables also to be regarded as monotypes.

(Refer Slide Time: 29:57)

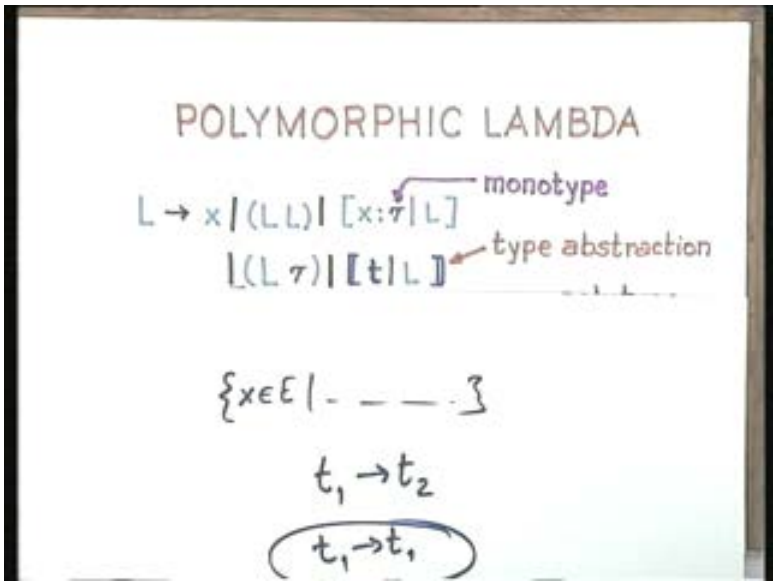


So type variables actually are going to denote particular instances of monotypes. Then we build our polytypes like this. any monotype is also a polytype and any polytype quantified over a free type variable, the notion of free and bound variables is as before over type expressions, if pi is some type then any type variable that occurs in pi is a free type variable and you can quantify over type variables and then that variable becomes bound. This is how we construct polytypes.

If you just extend this argument further we could also construct super polytypes by a similar grammar. Given that pi is a polytype you could define an infinite collection of super type variables, a collection of polytpes over polytypes and then super polytypes being defined in a similar fashion. The type hierarchy actually can add infinite upwards though lowest part of the type hierarchy is the monotypes and below the monotypes of course are values and functions. Let us limit ourselves to the type hierarchy at two levels. There is just type variables and type variables can take values only from monotypes and monotypes was whatever that were defined in the simply type lambda calculus. Then you can construct a polytype by quantifying over the monotypes. We are quantifying over the type variables.

So the polymorphic lambda calculus is defined in this fashion. (Refer Slide Time: 32:22) So we have the usual syntax of the simply typed lambda calculus. remember that this is a monotype, remember that the simply typed lambda calculus was very nice in the sense that it gave you everything that could be statically type determinable, it could type check statically but the only problem was that the simply typed lambda calculus could not account for generalized combinators like I or twice which are polymorphic. I is of course very general, twice is not so general, twice will type check only for certain classes of argument types. For example, you cannot give twice a value from the base type, it will not type check.

(Refer Slide Time: 35:42)



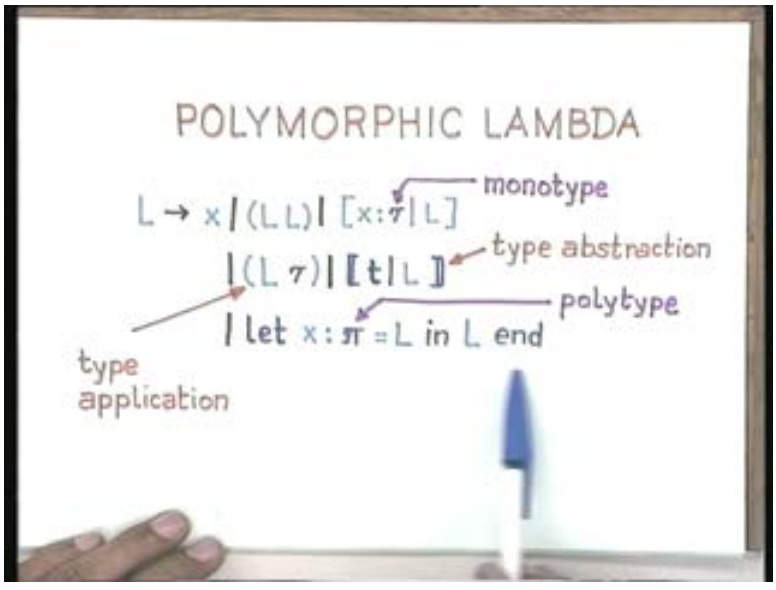
For example, you cannot apply twice on an integer. Twice can be applied only to another function which means it should have a type of the form some t_1 arrow t_2. It could only apply to another object which has a type t_1 arrow t_2. It cannot apply to just a base type but it has to apply to a function. But that is essentially like the way we write sets. Let us take this x such that x is even and then it satisfies some other property. You can take subsets of the naturals and write generalized set definitions. So functions like twice are polymorphic in the sense that they do not range over the entire type hierarchy whereas

the combinator I actually ranges over the entire type hierarchy. You can give it a value, a function, a type variable or anything and the identity combinator will just give back whatever you ask, whatever you gave it. But the twice requires an argument which has a type of a certain form that it should be explicitly a function form, it cannot be a value form. Not only that it should explicitly be a function form and it cannot be a function form of type t_1 arrow t_2 where t_1 and t_2 are completely different. It has to have a type t_1 arrow t_1.

For all instantiations of t_1 such that you have functions from t_1 arrow t_1 the twice can be applied that is the type of twice. That is what is obvious from the definition of the function abstraction in twice. So now what we have is in addition to the monotypes we have the type abstraction on lambda terms. So this is the type abstraction which I pointed out and this is actually an application of a monotype of presumably a lambda term which takes a type as a parameter. So if this lambda term were the polymorphic combinator I you could give it any monotype as an argument and it will specialize to that particular I tou.

In the case of I of course this tou could be anything in the monotypes. In the case of twice this tou would have to be of the form sigma arrow sigma where sigma arrow sigma is constructable from the base types through the monotype context free grammar. So this is actually an application of a lambda term to a monotype so as to specialise that combinator for that particular type. Here since this was in Ravi Shetty's book I decided to add this construct also.

(Refer Slide Time: 39:08)



So here is a let expression and wherever I have written tou it is a monotype. That means it is a restricted part of the type grammar that we have got, tou is a monotype means that it comes from the language of types in the simply typed lambda calculus but with the added construction that you could have variables instead of actual expressions built up of

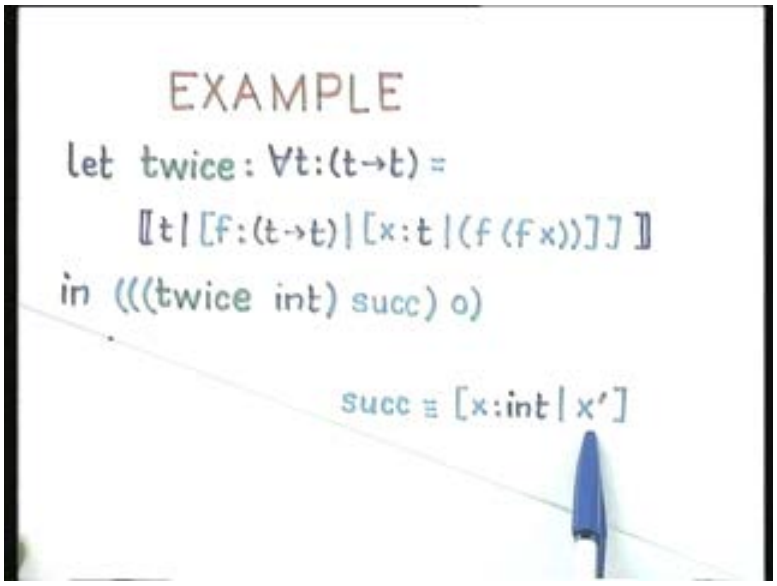
type constants. But now we can take a polytype so if x is supposed to be a polytype then this whole let construct is a lambda application in which all free occurences of x in this will be replaced by this I should type check.

Essentially the most important additions are really this, the new type application in the type abstraction, this type application means that there is a beta reduction for types and this is the construction of more complex functions which are polymorphic from the polytypes themselves. So this is actually a form of lambda application as you will be able to see and when you are given the rules it should become clear.

For the moment let go off this or rather let us keep this because i am going to give an example which illustrates this.

It is a very nice example again drawn from Ravi Shetty's book. Let us take twice so I have this expression which is really let twice which is defined in this fashion, twices of type this it is polymorphic as you can see because it has the universal quantifier over type variables so it is not a monotype. So consider the polymorphic function twice whose definition is given by this in the expression twice int successor, successor is a standard successor function written in the lambda calculus. We will assume applying it on integers.

(Refer Slide Time: 40:31)

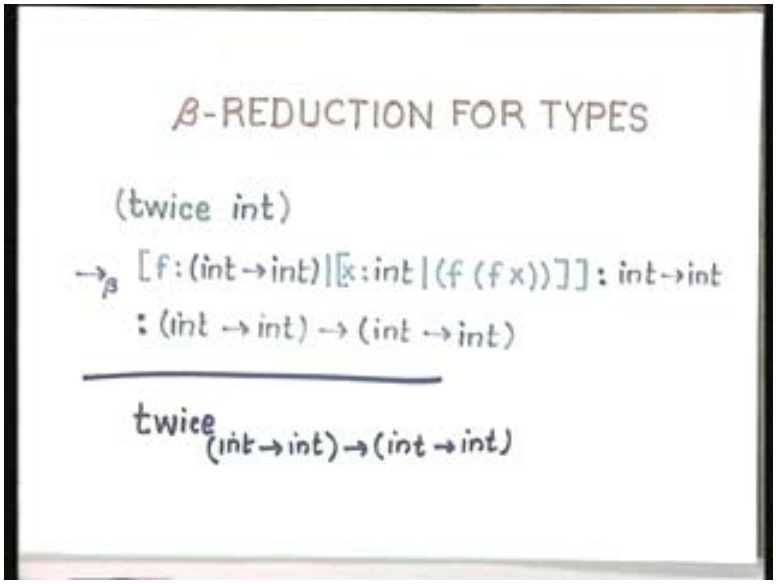


So for any x the succesor of x is as defined by Peano Arithmetic. So now what we are saying here is apply first twice on to integers. That means particularize twice to integer functions that mean functions from integers to integers. So what you get when you apply twice to integers is a new function which is particularized to all functions of the monotype integer to integer. Apply that function on to the successor the result of which has to be applying successor twice on to whatever is the argument. So the result of this application is to particularize twice to the type int arrow int and having particularized

twice to int arrow int you apply it now to successor which is a function from int arrow int so it is perfectly understandable so it is type compatible.

Since successor is lambda x x prime where x prime is going to be int and therefore this lambda abstraction gives successor of the type int arrow int so this twice applied to int gives me a function int arrow int arrow int arrow int which applied to a function successor which is of type int arrow int gives me a result which is a function of type int arrow int which given the argument zero gives me a value in int. So essentially it will give the value two which seems however big the mess is to get into just in order to get the value two but in principle it is a powerful operation. So I have just showed how the beta reduction works for particularizing. So the beta reduction for polymorphic types is to really particularize that function for a certain type or instantiate the universal quantifier in the type to a particular kind. This twice applied to int is actually what I have written, if you follow the notation that I used before it is actually twice particularized to int so really I would have given the subscript int arrow int arrow int arrow int.

(Refer Slide Time: 44:04)



So this twice works only on functions of the form int arrow int. Of course there is a possibility that I could have applied twice on a type variable too. That variable may have been quantified somewhere later, I can have nested quantifiers for all s, for all t and so on and so forth then that variable might get its typed value because of the instantiation of a quantifier that exists somewhere in the outer scope.

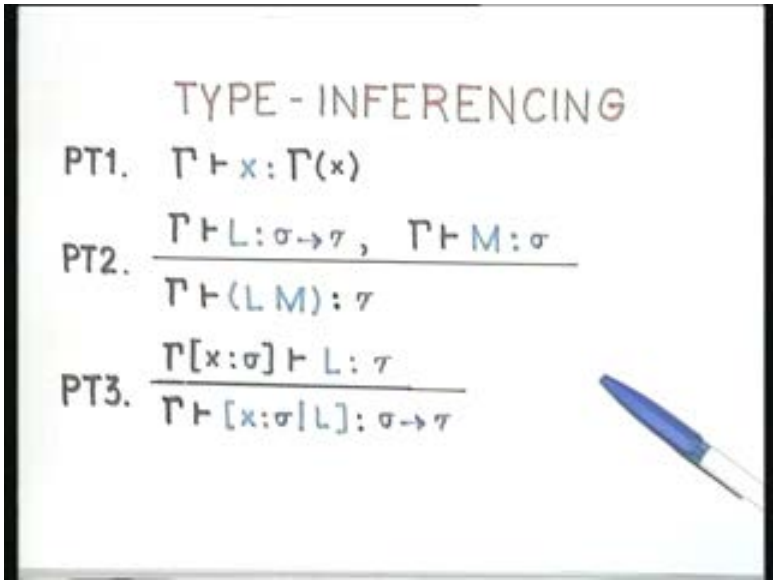
And if I look at this restricted sub term then it would be twice with the subscript t arrow t arrow t arrow t where t should get its value instantiated somehow later. Now you understand why beta reduction is really important:
Checking whether membership in a set is really a form of beta reduction,
Applying functions is really a form of beta reduction,
Universal instantiation is really a form of beta reduction and

Constructing sets by abstraction is really a form of lambda abstraction. Constructing quantified predicates is really a form of lambda abstraction. Constructing types is really a form of lambda abstraction. So applying types and instantiating them is also a form of beta reduction.

Hence beta reduction in computation is really the most fundamental concept which has probably evolved over the last forty years. Almost anything that is constructive; by constructive anything that is computationally relevant has beta reduction appearing in some form or the other. Parameter passing and procedures is a form of beta reduction. Whether the parameters are passed by value or by reference or by name they are all forms of beta reduction and now types are also forms of beta reduction.

So the generics that you have in C ++ an Aida are really some very restricted form of beta reduction applied only to some base types. So far in terms of implemented programming languages the most sophisticated type system that has so far come up is the M L polymorphism which is completely statically determinable where polymorphic types are statically determinable which means they are determinable at translation time without going into executions. That is one of the reasons for studying M L because it is not just that it is a functional language but also that it has a very sophisticated type system. The most sophisticated type system in existence in an implemented programming language is in M L.

(Refer Slide Time: 47:38)

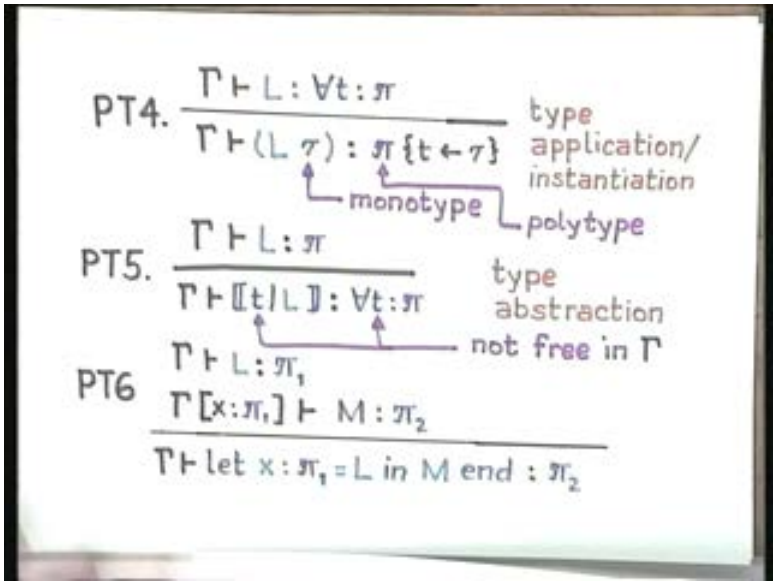


Actually by now you would have got a flavor of type inferencing rules that you should have and these are essentially those. Again given a type context gamma or a type environment gamma, and this x is free so the context does not contain x then it does not type check and you throw out that program, it is as simple as that. So whatever the context we can give for x is the type of x and this is the usual application of functions.

If L is of type sigma arrow tou, by the way sigma and tou are monotypes, this is the monomorphic application and M is of type sigma which is again a monotype then L applied to M is of type tou. If with the assumption that x is of type sigma added to the context you can infer that L is of type tou then the lambda abstraction x sigma L has the type sigma arrow tou where again all these sigmas and tous are monotypes. So the type inferencing rules here are really like the type inferencing rules as the simply type lambda calculus and there is no difference. But what we require now are type inferencing for the polymorphic lambda calculus.

If it is determinable that L has the polymorphic type for all t pi where pi could itself be another polymorphic type because you could have a sequence of quantifiers in nested quantifiers so pi could be a polymorphic type then given tou a monotype you cannot apply L to another polymorphic type but you can apply it to a monotype. That is applying L to a monotype means particularizing L to a certain type, applying L to a polymorphic type does not exist in our language so far. But as I said there is no reason to build up the type hierarchy, you have quantifiers over polymorphic type variables too then you would have a particular reason. There is actually a reason for doing that, when you go from to types higher than this it turns out that lot of problems of undesirability crop up. This is about the limit that we have currently reached.

(Refer Slide Time: 54:15)



So given that tou is a monotype you are particularizing the polymorphic lambda expression L so that an incarnation of it for the type tou is created by this application. That means this universal quantifier for all t has to be eliminated by instantiating t with the value tou and that s what this substitution does. So it takes pi and for all free occurences of t in pi it replaces those free occurences by the monotype tou. This is clearly a case of universal instantiation it is also a form of beta reduction for the types. So this is a type application or instantiation where pi is a polytype and tou is a monotype. So this is

like universal instantiation so there should be a corresponding universal generalization or quantifier introduction rule and that is what this does.

Given that in the context gamma you can show that L is of polytype pi then this abstraction over the types. Over the free variables t over the free type variables t in pi gives you a polymorphic lambda expression which has a type for all t pi. So this is type abstraction. It is very similar to the lambda abstraction. The only thing of course is that whenever we are talking about free variables whether its type variables or value variables and whenever we are talking about binding them we should ensure that there is no capture of free variables because of the binding you are creating. Because of the introduction of the quantifier you are binding this variable t and as a result there should not be any t in the context grammar already defined because if that t occurred in pi then that t would get captured by this quantifier.

So the usual confusion of free variables, bound variables, alpha conversion with quantifiers and so on, of course alpha conversion also exists anywhere where there is binding or declaration if that is alpha conversion. So you have to do aplha conversion to a bound variable to ensure that there are no free variable captures. Hence, t should not be free in gamma that means t should not already have been declared in gamma and then the last role is just really a form of typed lambda application for the polymorphic case; if L is of type pi_1 which is a polymorphic type and with the assumption that x is of type pi_1 if you can infer that M is of type pi_2 then this whole let expression really has the type of M because M is really the expression and the meaning of any let expression is the body of the expression which in this case is M so it has a type pi_2. So, let expressions really are like applications because the semantics of a let expression is equivalent to substituting all free occurences of x in M by L. Therefore now you can go back and try to determine what the type of this is so you have really reached the highest levels that types can reach in a desirable fashion.