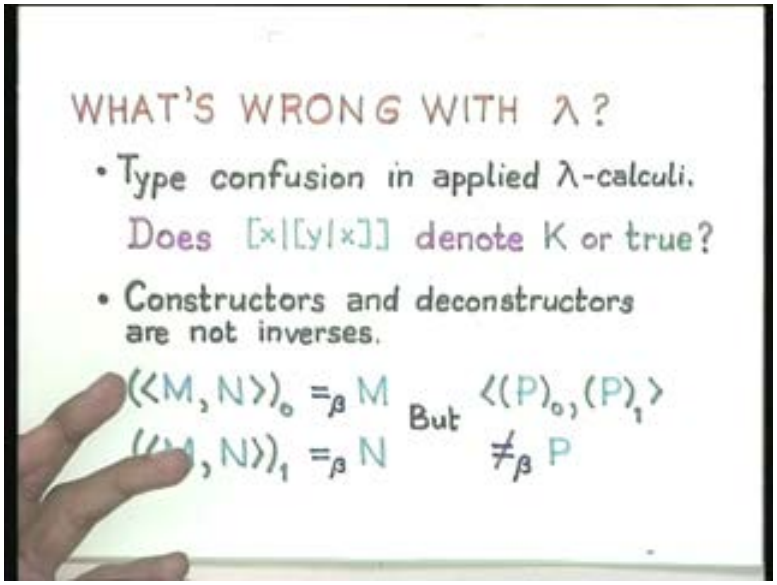
Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 32
Monomorphism

Welcome to lecture 32. So today I will briefly go through what I did the last time and pick up from there on the simply typed lambda calculus. We just looked at what's wrong with the untyped lambda calculus. Firstly there is a type confusion often you do not know whether you are applying K to the argument or not for example. You might be asking is zero true so then are you applying K or true to zero and one may be.

(Refer Slide Time: 01:01)

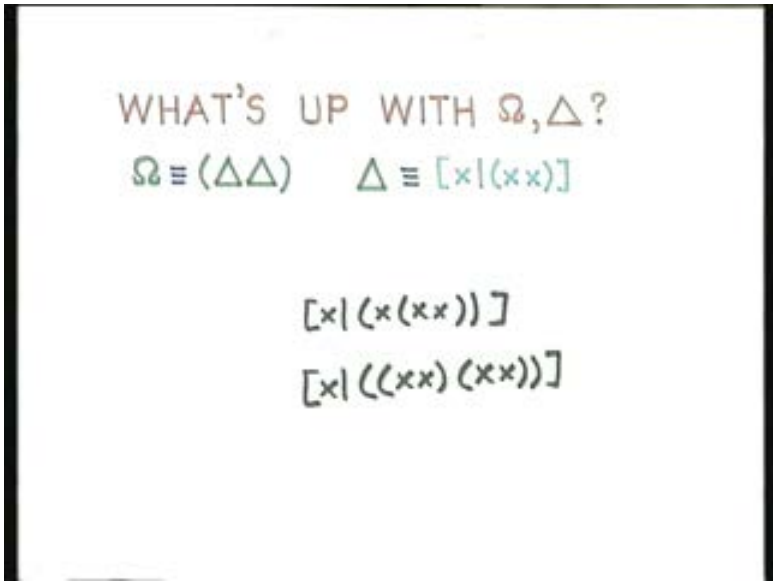


So there is some type confusion but that is not so serious. A slightly more serious problem is that constructors and deconstructors are not inverses which mean that if you apply deconstructors onto some arbitrary term which is not explicitly constructed through a constructor then you might get some result and you will not know whether the result is wrong. That is one thing. One thing is to find out who the culprits are?
One possible thing is to actually look at this very general purpose language definition and see whether everything in it is really meaningful. May be what one should do is may be one should restrict the language somehow so that only really meaningful things are actually used and this is something that is quite natural even in our natural languages. Therefore one thing is that one of the reasons for meaninglessness if you were to actually apply this lambda calculus as a calculus of functions to some data type applied somewhere then one thing that really does not seem to make much sense and which mathematicians over thousands of years have agreed upon is for a function to apply to itself or for something like this. So what happens with things this is that they complicate

matters in more ways than one. One thing is that they do not actually seem to mean much.

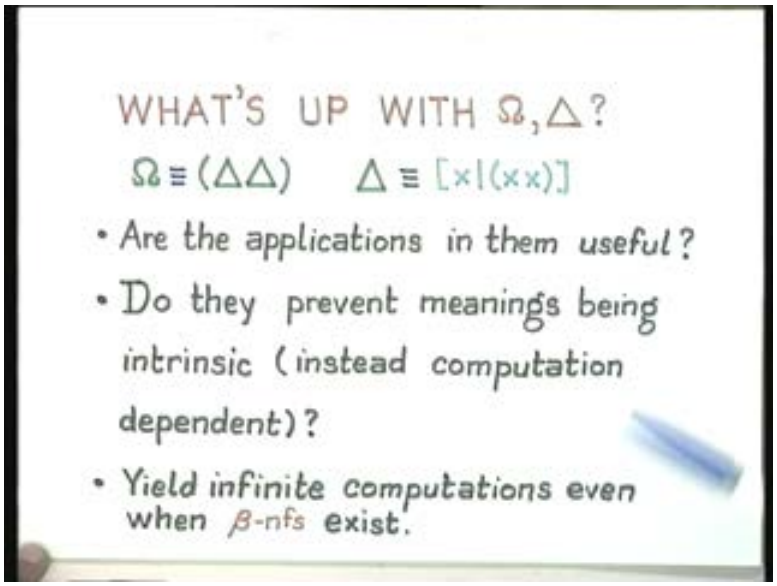
Secondly, if you remember, with such replicating combinators you could have other replicating combinators. For example, I could define a combinator like this which will just keep making three copies somehow applying them or like this which will make four copies and somehow apply them pair-wise and so on and so forth.

(Refer Slide Time: 03:34)



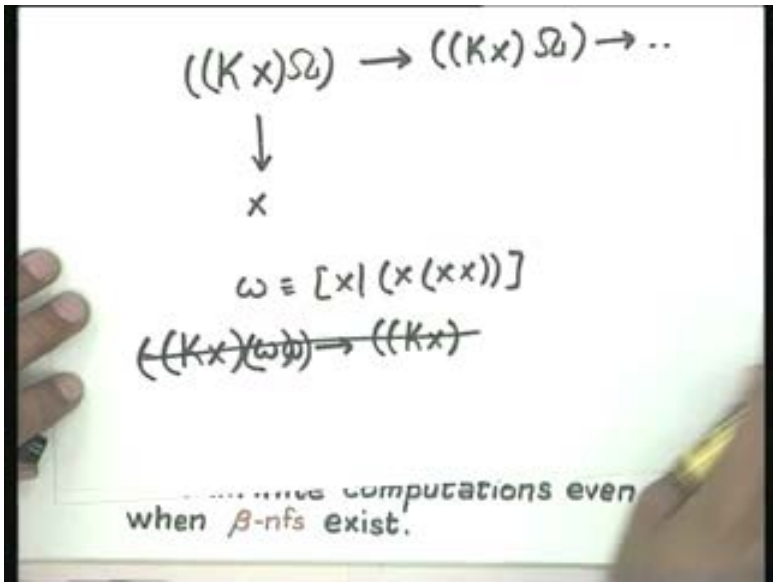
There is an infinite number of such combinators that one can form. And the thing is that it does not seem like we can actually give a decent meaning to all of them. So one possibility is to actually try to restrict the language so that somehow you get only meaningful combinators meaningful lambda expressions which really have the potential for being functions which can be applied in a general fashion on may be some data types or some other symbol positioning system and thereby get a decent model of computation. Another thing is of course that because of the non-deterministic nature of beta reduction even if you were to try to extract a meaning the meaning does not seem to be intrinsic it seems to be computation dependent.

(Refer Slide Time: 04:22)



So one example that I gave before was of this, if you take this combinator K and apply it to x and omega in this fashion then you have a one step beta normal form which is just x or you can do an infinite number of steps reducing this omega to itself. The omega reducing to itself is only the simplest of complications. If instead of this omega if I have actually took this other combinator which replicates things three times then I will get an explosive computation where no two terms are identical. Let me call that little omega.

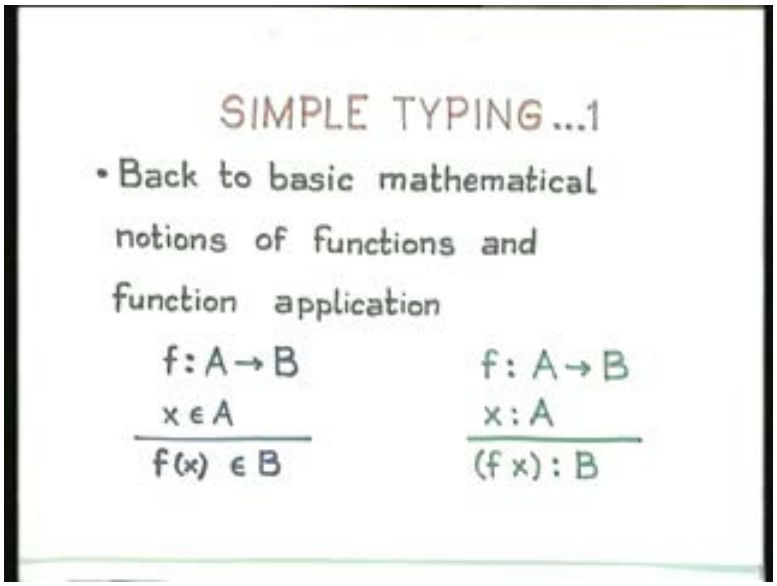
(Refer Slide Time: 05:49)



Let me say that little omega is this then I could take this combinator like this and then if I try to do Kx and little omega what will happen is I will get Kx omega omega omega

applied to itself. So if you take omega applied to it what you get is omega omega omega and this can go in various directions depending on how you want to do the application. All of them will just keep multiplying copies of this little omega. So when you take this Kx omega omega then there is one normal form which is directly obtained otherwise you get these explosive computations where Kx will be preceding each other. And all these do not seem to really have some meaning. Therefore one possibility is to actually try to get rid of them. So one other complication they come up with is that they yield infinite computations even when there are actually beta normal forms. Therefore let us go back to our basic Mathematics and we have a simple typing scheme of this form.

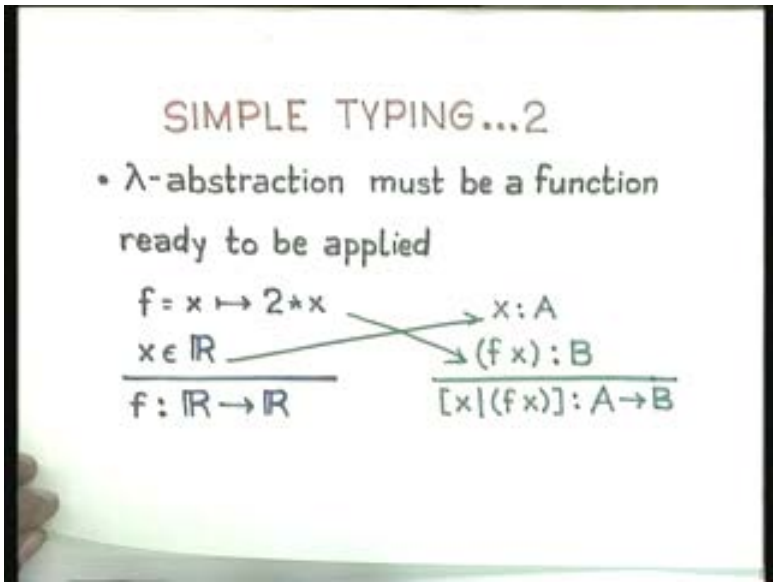
(Refer Slide Time: 06:59)



Remember that we have to look at values and functions as far as possible equally. There is an important result in the theory of computation which says that there is no general algorithm to decide whether two given functions are the same or not. But there is an algorithm to decide whether two given values are the same. However, not withstanding that as far as possible we would like to give them equal status. So one possibility is to unify the notation in this fashion and actually put a check on the construction of terms so that they are well-typed.

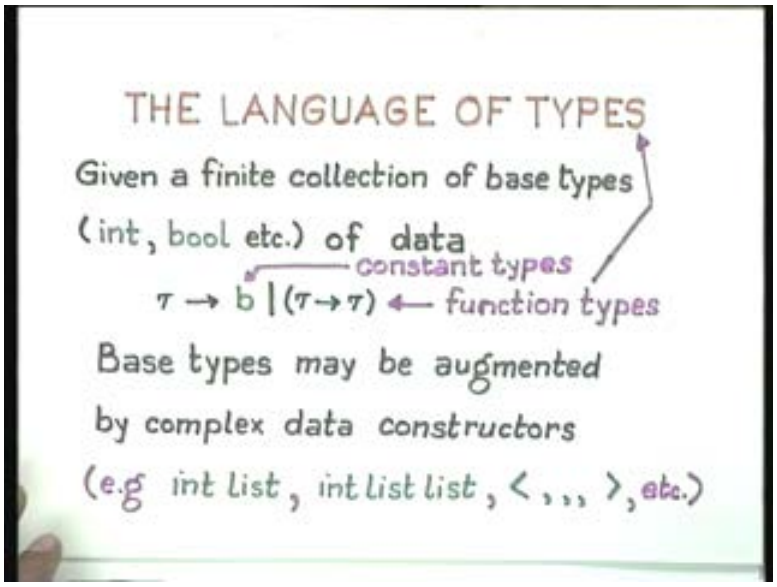
Similarly, put a restriction that lambda abstraction must be a function that is in some sense ready to be applied and the lambda abstraction is actually something that somebody should have discovered probably long ago since Mathematicians are using such notations.

(Refer Slide Time: 07:59)



This kind of a typing scheme leads us to firstly a language of types which allows higher order functions.

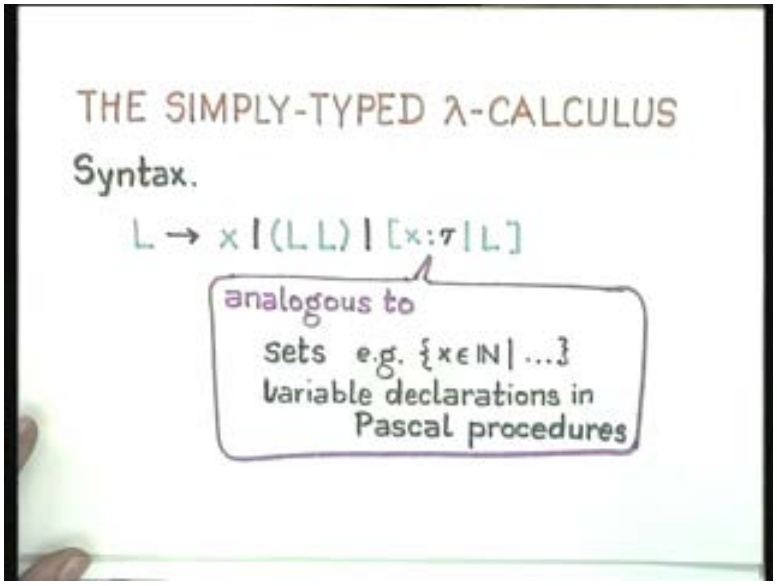
(Refer Slide Time: 08:10)



Hence for example we could start with a finite collection or even an infinite collection of base types if you allow various kinds of pattern formations like this. And for simplicity let us assume that we have a finite collection of base types and we could define a language of type constructions. This language allows us to construct functions of the form int to int, int to bool, bool to int, bool to bool, int to int the whole thing going to int to int, int to int going to bool and so on and so forth. It allows all these kinds of

constructions. And one easy thing to prove is that every type is really of this form. This can be proved by induction on the production rules of this grammar on types. So what we will do is put this typing on top of the lambda calculus somehow. So one thing is that we will define the simply typed lambda calculus as one in which types of variables the bound variables are specified.

(Refer Slide Time: 09:26)



And if you look at a complete program as one which has absolutely no free variables, free variables will only appear only in sub programs. Similarly if you take a complete lambda expression as a function then there should be no free variables all the variables should be bound. Hence only sub terms can have free variables. And of course this notation is analogous to our standard set notation where actually this is like a typing constraint.

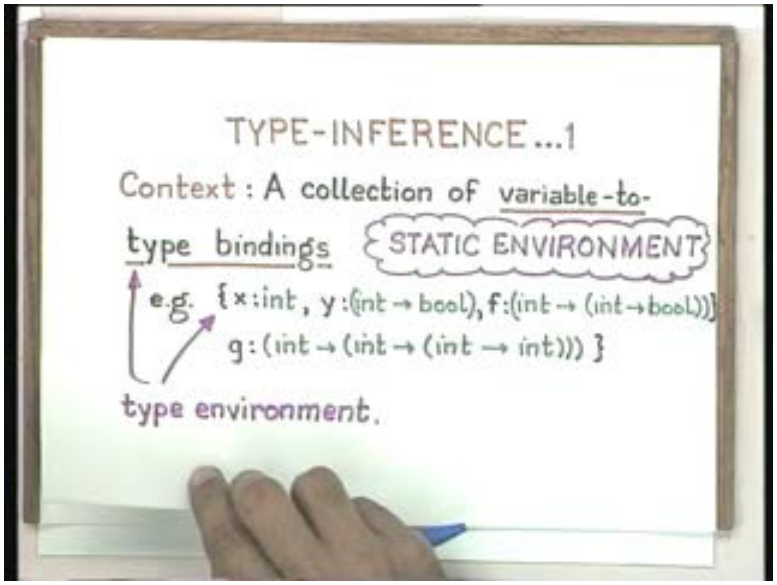
For example, if you define the set of all numbers which are even, even does not make any sense with real numbers or complex numbers and so on and so forth. So it makes sense with naturals so you put a typed constraint on the numbers. This is a very common practice and we just follow that practice and construct the simply typed lambda calculus with this two level syntax. One level of syntax is for the typed language using the base types and the other level syntax is for the actual lambda expressions.

Now the important thing is I can actually determine the types of various lambda expressions statically. Thus we will define a context as a collection of variable to type bindings and we will call this a static environment. The reason for calling it static is that whatever we are going to do something that can be determined at compile time it does not require run time checks. Whether a certain lambda expression has “a good type” is something that can be determined by a compiler without executing the program. That is why the word static is mentioned here. In anything to do with programming languages and compilers if the word static is used it means it is something that can be done before execution at translation time or at compile time. If it is an interpreted language it is still

translated, you can do it before execution. Whatever can be done before execution is said to be static otherwise whatever can be done only at run time is dynamic. So we can construct a collection of such bindings and essentially this context is what constitutes the symbol table.

The symbol table that a compiler constructs has this context as an essential part of this symbol table, the types of the various identifiers and so on and so forth. So an essential part of the feature of the symbol table is really a static environment as opposed to a dynamic environment which is the activation record, stack and so on and so forth. So an essential part of the symbol table is really type checking and it is all something that can be done at compile time without actually running the program. So whatever we do is something that can be done at compile time. So let us look at the type environment and we can give actually inference rules of this form.

(Refer Slide Time: 13:00)



We should look at type inferencing of course in a structurally inductive fashion going down the syntax tree somehow. When you go down this syntax tree even if you started out with a program which had no free variables when you look at sub expressions and the sub programs they are bound free variables in that context. So we will assume that there exists a context so for a fully defined program for a full program especially a program which does not use library routines and so on and so forth a complete program actually starts with an empty type environment just like often it starts execution in an empty environment with an empty activation record environment. So during the process of compilation you will be collecting a lot of type information about the variables. So whenever there is a reference to a variable you look into your type environment. If the type environment has a type specified for it then that is the type of this variable.

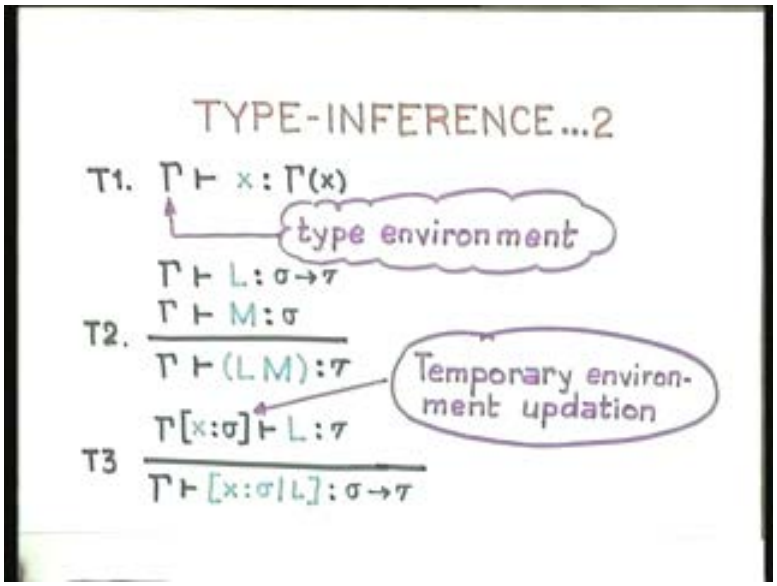
Essentially you just access the symbol table and see whether that variable was already declared and if it was declared in all languages which insist on declarations preceding use

then there must be type information unless it is a forward reference in which case it has to be back patched later**.** But in general the symbol table should contain the type information if declaration precedes use. Of course there are implicit declarations and so on. Most of them are otherwise algorithmic aspects which have nothing to do with the system of typing.

Now we would say that this application is well-typed and actually has a type tou only provided L has a type sigma arrow tou for some sigma and m has a type sigma. Essentially what we are looking at are something acclaimed in Mathematics to domain and range information, domain and co-domain information. If there is a function from natural numbers to natural numbers then it has to get an argument only which is a natural number. So you insist that all applications are meaningful only if the argument to an application the operand is of a type that is consistent with the domain of the operator. And if it is so then you can infer the result, the result is whatever you get in the co-domain, it is as simple as that.

And in the case of a lambda abstraction you insist that it has to be a function because it has a bound variable which is somehow going to be replaced by a beta reduction to a function application which means a lambda abstraction should actually represent a function. Its type must be something which contains an arrow, it cannot just be a base type. A lambda abstraction cannot be a base type, you cannot have a lambda abstraction which is of type int. It has to be a function from something to something.

(Refer Slide Time: 16:50)



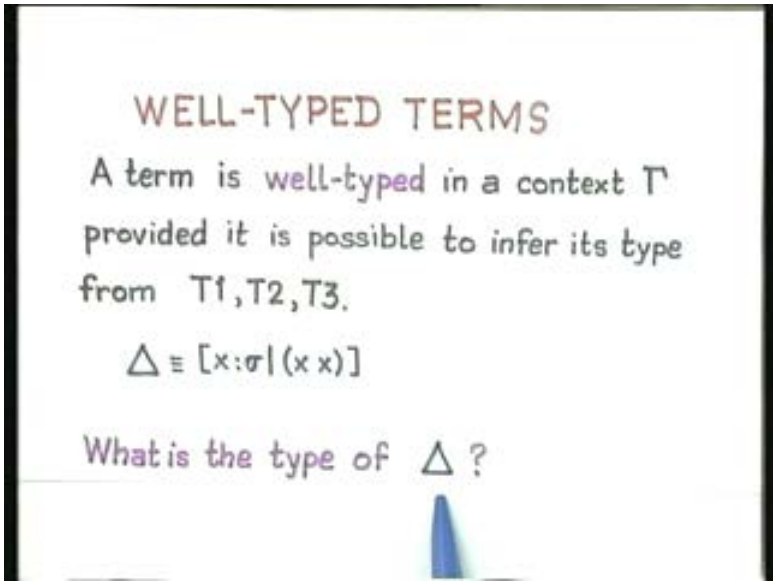
However, if you go about things in a structurally inductive fashion assume that the body of the lambda abstraction has a type tou and if x has been declared as being of type sigma then this lambda abstraction represents a function which goes from sigma to tou which takes arguments in sigma and gives you results in tou. The important thing about all this is that there are no executions involved, it is all something that can be done by a

compiler. That is why most of the typing information is done at compile time. There is absolutely nothing at run time that is necessary for such things.

The static environment is very much like the dynamic environment. you are doing temporary updations because you will have newer and newer declarations with bound variables being re-declared, you will have newer declarations may be for the same identifiers so you have to temporarily update your environment so that you know what is local and what is global and these things are all meant to be local so the static scoping rules are in force. So you require this temporary updation because of static scoping rules. These rules actually tell you what a well-typed term is. So we would say a term is well-typed in a context gamma provided it is possible to infer its type by the application of rules T1, T2, T3 a finite number of types.

Remember that these are rules and they have to be applied only a finite number of times otherwise your compilation will be a non-terminating process. If it requires an infinitary proof then your compilation itself is going to be non-terminating forget about executions even your compilation is going to be non-terminating which you do not want. So the next question is have we actually achieved the purpose, have we banned combinators like delta?

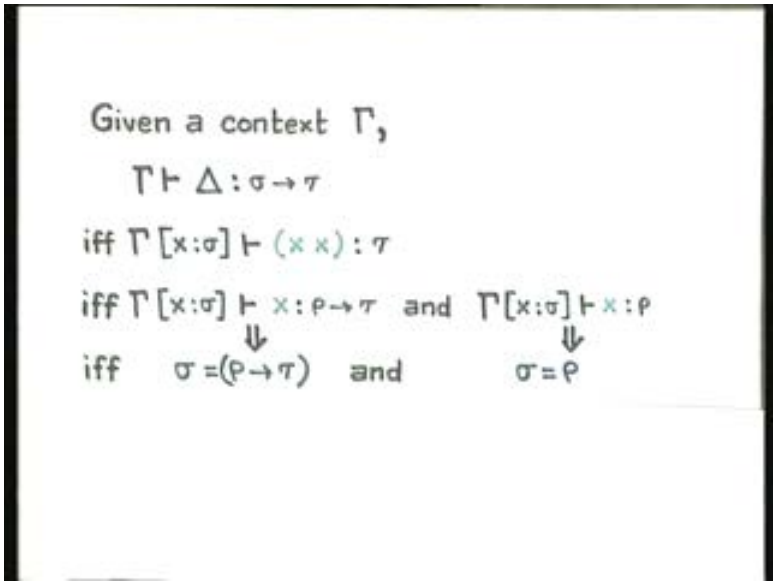
(Refer Slide Time: 19:19)



So let us look at delta itself and try to see whether by applying these rules T1, T2, T3 we can actually infer a type or what actually happens. Let us assume given an arbitrary context gamma. So supposing delta was of a type since it is a lambda abstraction it could quite well be of a type sigma arrow tou for some sigma and tou in terms of the base types. But delta could be of types sigma arrow tou if and only if the body of delta which is x applied to x is of type tou.

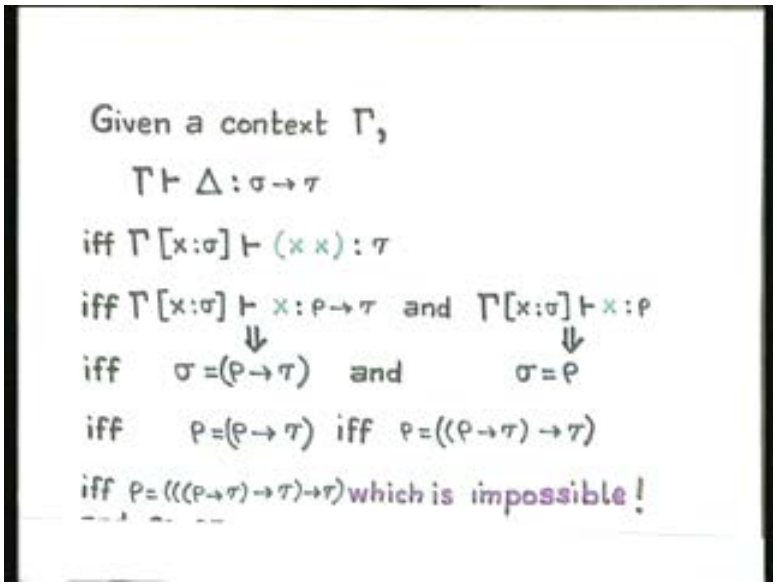
Now x applied to x could be of type tou only if this x was of type rho arrow tou and this x was of type rho otherwise how would the application be meaningful. So what you are embarking on is going down as deeply as possible in order to infer a type. You are trying desperately hard to give delta a type and the conditions the constraints you are getting are these. So I can infer a type for delta only provided I have x of type rho arrow tou and x of type rho both in the same static environment. Now this thing is possible only if this sigma is equal to rho arrow tou and sigma equals rho.

(Refer Slide Time: 21:13)



After all it is possible that these things look deceptively different but they might be actually solvable as being equal. How do you know they are not solvable? But when you do this, when you get it in this form what you see is that, when you do the substitution essentially what you are trying to do is you are trying to unify these two terms in order to find a solution in fact a most general unifier. So when you try to do this what happens is that you just keep expanding out infinitely.

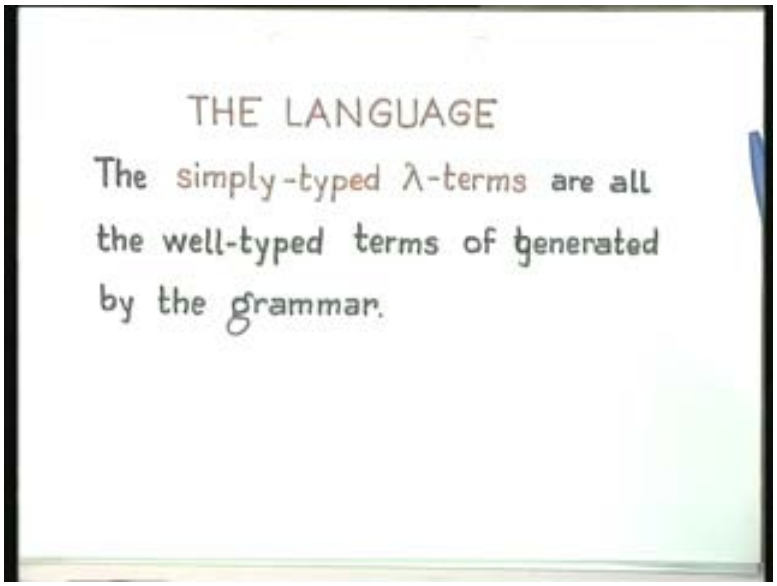
(Refer Slide Time: 22:03)



This keeps on expanding in this fashion infinitely and your tou has not yet been fixed to a base type int or bool as specified in our language type such that every type that is valid is something which will have a last thing which is a base type. You have not yet been able to infer what the base type tou is which you are getting every time in every unfolding of this equation. Therefore this inference is going to go on infinitely. So without an infinite proof one cannot infer a type for data. And of course even after an infinite proof I do not know what the type is. This is impossible, here it is really to be read as something that whatever may be its type it is not inferable in a finite proof.

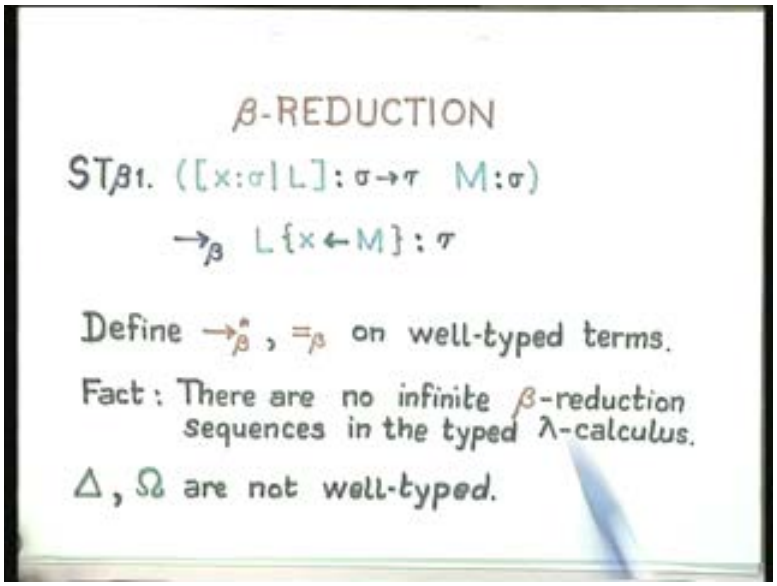
The point about unification of course is that I went about it rather in a simplistic manner but the unification algorithm is deterministic and it will clearly point out that this is impossible, equating these two is actually impossible. Any unification algorithm will be able to point that out. So it won't even go out through an unfolding and it wont even look upon that as a recursive definition which is to be unfolded. It will look upon that as an equation which is to be solved by a most general unifier and it is impossible to get the most general unifier because the disagreement sets are such that one is the subset of another. So when the disagreement sets are like that then you know that you are not going to be able to solve it and that is how a unifier would actually look at it. So it is actually compile time feasible to detect that delta cannot have a type assigned it is not necessary to go through an infinite unfolding process. But logically speaking from the point of view of rules what it really means is that you cannot infer in a finite proof the type of delta. That is how the simply typed lambda calculus goes. Therefore we will just look upon the language now that the grammar for the types itself puts the appropriate restriction, the inference rules put the restriction on the kind of terms which can be executed.

(Refer Slide Time: 24:30)



So what it means is that if you put in a typing inferences engine with a unification algorithm in your compiler the compiler will just throw out all those terms and say impossible, the type cannot be determined. Therefore it will not even permit execution; it will not generate code to execute. So we will just look upon the simply typed lambda terms as all the well-typed terms generated by the two level grammar type and the lambda calculus. Of course we just have to complete a few formalities. We have to define what beta reduction is and we should define beta reduction in such a way that it is well-typed.

(Refer Slide Time: 25:18)



If now this portion is not part of the language but this is what the compiler has inferred whatever is in light blue is really part of the language of the simply typed lambda calculus but by an application of the rules of inference assuming that L is a term of type tou what your type inference system will produce is that this lambda abstraction is of type sigma arrow tou and then only if the argument that you give to this application is of type sigma will it actually perform a beta reduction. So whatever is in dark blue here is really something that is actually not part of the language but something that is part of your type inferencing system that is again a part of your compiler or translator for the language.

So the actual terms are those that are given in light blue. So, given an application of the form lambda x: sigma bar L applied to M and assuming that the type inferencing system can assign these types to them through the application of rules T1 to T3 then a beta reduction is possible which will give you a term of type tou. And what we can do is we can carry these definitions forward just as we did before. You can define a many step beta reduction, equality on beta.

Now, when you do equality on beta it is guaranteed that you can never equalize two terms which do not have the same type. After all they should both be beta reducible to a common term so all the terms in your beta reduction should have the same type, all the steps in your beta reduction should have the same type. That is implicitly guaranteed once you go through a type inferencing system. And the interesting thing is that since delta and omega are not well-typed and they are thrown out by a type inferencing system you are also not going to get these horrible infinite beta computations.

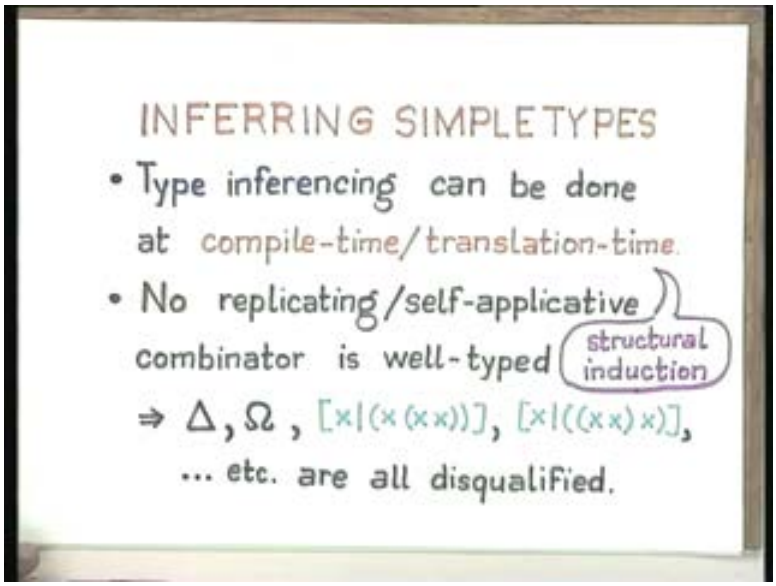
Actually there is a caveat there that does not mean that with a type inferencing system you can guarantee that every program terminates, you cannot guarantee that every program is an algorithm. It is only guaranteed for the typed lambda calculus with base types which are not actually applied some things like numbers. The moment you bring in numbers we can really sit together and design a really lousy definition which will run for ever. But the thing is if you remove functions or numbers and you look at only the simply typed lambda calculus with this beta reduction with all these replicating terms out there are going to be no infinite beta reductions.

Infinite computations do come in the moment you apply it on to some other domain like numbers. But by just using this base types int and bool or whatever as patterns and not actually using any integers or Booleans just looking at the lambda terms without actually doing number computations, not bringing in Peano Arithmetic and things like that just use these int and bool as patterns for possible values just look at pure lambda terms there will be no infinite computations because you have outlawed all these kinds of terms which have the potential for replication.

In the typed lambda calculus what it means is that then beta reduction is strongly normalizable they are always guaranteed. Coupled with the fact beta reduction is Church-Rosser what it also means is that there are unique normal forms. If something is Church-Rosser and it has two different terms then they should be able to meet at a common term by the diamond property which means you will have a unique normal form. Because if

you have two different distinct normal forms which are not mutually alpha convertible then by the Church-Rosser property it says that there is a common computation which they should both meet. So you cannot have two distinct normal forms. These are some of the nice properties that come out of type checking and type inferencing.

(Refer Slide Time: 32:11)



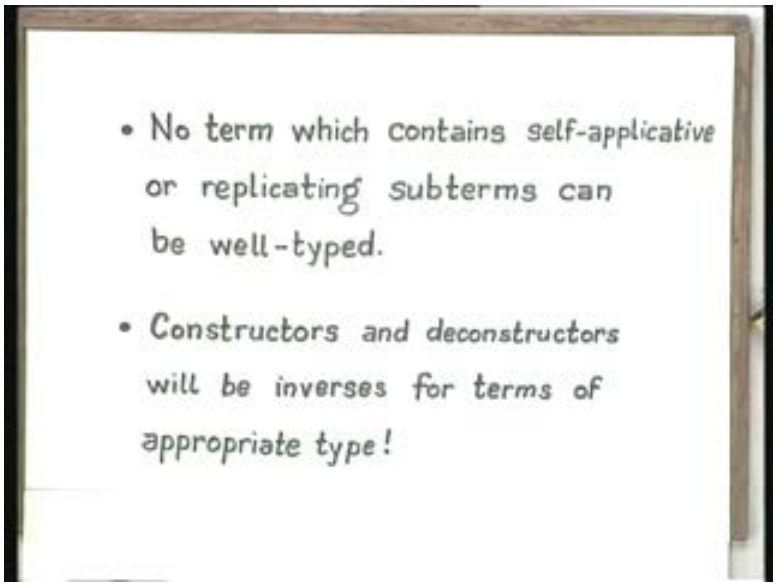
Let us look at what we have done. So one thing is that when you do simple types the type inferencing can be done entirely at compile time or translation time. In the case of interpreted languages it can be done at translation time before you actually perform any kind of executions. The second thing is that there are no replicating or self applicative combinators in this new language and therefore those horrible infinite computations are outlawed and the type inferencing is done by structural induction which in practical terms and in terms of a parser and so on means that you will do it through a recursive descent parsing technique. If your compiler has implemented by a recursive descent parsing technique then you will do it through that. If it is some other parser like a table driven parser or something then in the table you can also incorporate these rules for the appropriate productions.

Therefore all these horrible combinators which complicate life are removed on the basis that they are really meaningless things and they are really complicating life. since they cannot be typed the unification algorithm within the type inferencing system which is within the compiler will actually produce a failure so the type inferencing system will actually produce a failure because of that and the compiler can just throw these programs out without generating code.

Therefore the simply typed lambda calculus that we have looked at basically starts with the assumption that no term which contains either self application or a form of replication can be well-typed. Therefore anything that is built up on top of self applicative or replicating terms cannot be well-typed. After all by structural induction unless you can

type the innermost terms you cannot type the outermost terms. So if this omega and delta and so on are embedded deep inside a huge lambda term your recursive descent parser will go up to the omega lambda and keep coming up from the recursion and produce a failure. The other nice thing is once you have put in a typed discipline you cannot apply arbitrary deconstructors to arbitrary objects then your constructors and deconstructors will actually be inverses.

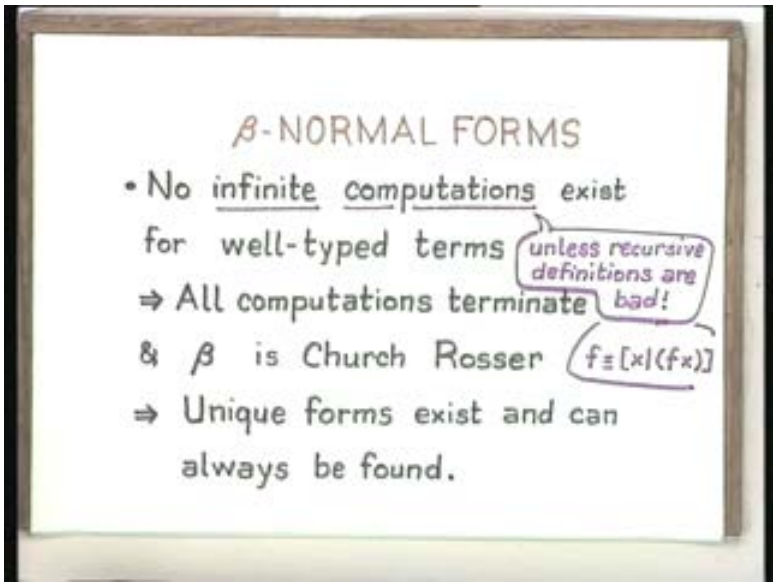
(Refer Slide Time: 34:02)



So a deconstructor will be applicable only if its argument is something of a corresponding constructor type and the constructor will be applicable to whatever only provided the elementary objects from which it constructs are of the appropriate types. Then once you have constructed these things you can keep applying these constructors and deconstructors as long as your compiler allows then you are guaranteed that they are meaningful. Now let us look at beta normal forms.

In beta normal forms there are no infinite computations unless you actually give really lousy definitions. All computations at least of the pure simply typed lambda calculus have a guarantee to terminate, computations of the applied lambda calculus in fact any calculus which incorporates natural numbers as a data type means that it has a potential for infinite computations. So only in the simply typed lambda calculus is it guaranteed that there are going to be no infinite computations. The moment you give these definitions over an applied term the moment you give bad recursive definitions you are going to get infinite computations. Then since beta is Church-Rosser unique normal forms exist and you can always find them for the pure simply typed lambda calculus. Now let us look at what is the simply typed lambda calculus term look like. What are some of the meaningful terms?

(Refer Slide Time: 36:02)



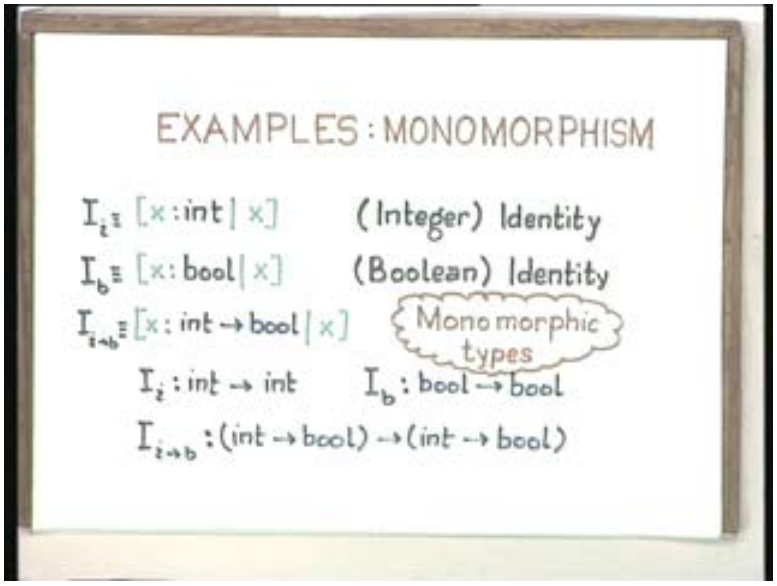
Let us look at some of the simple combinators. We cannot use terms like omega and delta but we can use some simple things like K, you could use the identity you could use S etc. So let us look at the identity function over these two base types integers and bool so the identity function in the simply typed lambda calculus over integers will look like this and your type inferencing system when it goes down it collects this information that x is of type int and then when it gets down to this free variable x it gets the information from the context that it is of type int then when it comes up again since it is a lambda abstraction it pronounces this to be of type int arrow int. So the identity function is an integer identity function. Basically what it does is if you give an integer argument to it will return back the same integer argument.

But the point about this is supposing you give a Boolean argument to it then it will throw it out because it is of a wrong type and it does not satisfy the conditions of the beta reduction. Remember that even if you use the lambda calculus representation of numbers and Booleans you will have to include type information on all the bound variables there. So you cannot give a Boolean argument to this function and expect to get any answer. It will not type check, the beta reduction cannot be enabled because it expects an argument of type integer and you are giving it an argument of type Boolean and it will throw it out. The good thing about that is you cannot for example ask questions like whether zero is true.

So what does it mean?
Supposing you want an identity function on Booleans what it means is you will have to have a different combinator like this. What applies to the identity function also applies to other complicated functions but the point is that they should all be well-typed. But now what if I want an identity higher order function which takes a lower order function and returns me the same function?

In general, what if I want a combinator which transforms one higher order function into another higher order function? Like derivatives for example the derivatives are higher order function which takes a function and gives you another function.

(Refer Slide Time: 40:29)

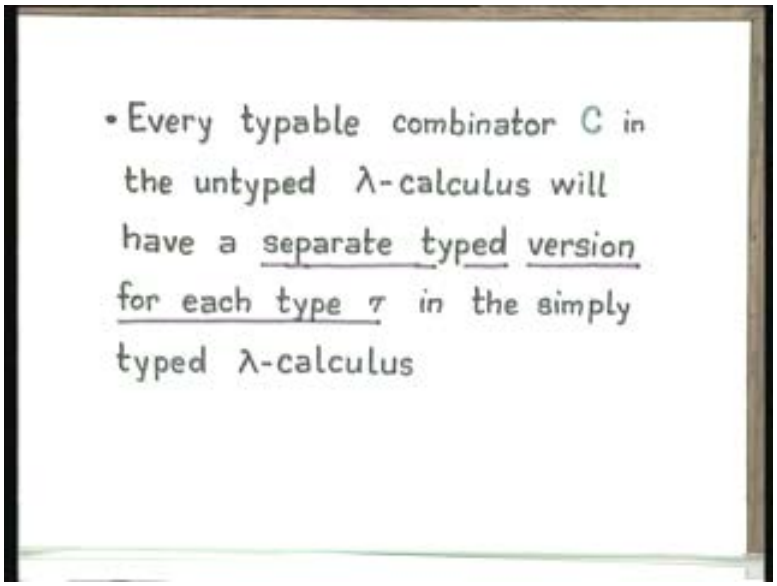


So, here I am taking identity as an example but it could be any of those functions. The point is that they are not going to be type checked unless the arguments and the functions are of appropriate type. supposing you take a higher order function for which you want a higher order identity function but you cannot take any arbitrary higher order function, let us say you take a higher order function of type int arrow int then you have to have a special identity function for int arrow int if you have a function from int to bool then you require a special identity function from int to bool.

Therefore, for example this combinator accepts only arguments of the form which have the type it arrow bool. This is a higher order higher order function which accepts another function whose type is int arrow bool and gives you back the same function. So if you want int arrow int you will require another combinator if you want int arrow bool arrow int arrow bool then you will require yet another combinator and so on and so forth.

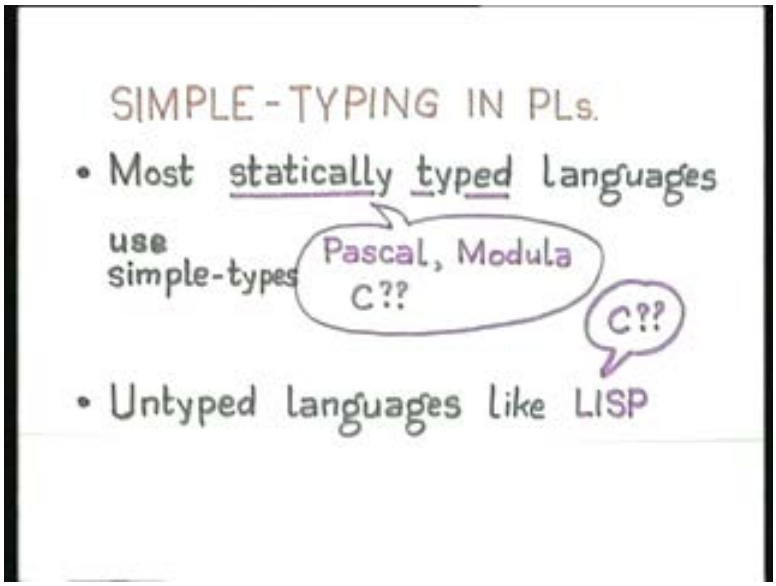
There is infinite number of such types because it is a context free grammar on the language of types. Starting from even from a finite set of base types I can construct an infinite number of types. This means that even for simple function like the identity function I require an infinite number of identity functions in order that my type checking actually works. what we looked on as in the untyped lambda calculus if you look at all the combinators which can some how be ascribed types with the type system so for every combinator C for which you can ascribe a type there are actually infinite number of typed versions of that combinator and only the appropriate typed version should be applied to the appropriate argument.

(Refer Slide Time: 41:42)



If you had a combinator C like the combinator K in the untyped lambda calculus for all types sigma and tou for which K is a valid combinator to be applied on type sigma and tou you will require new combinators one for each sigma tou combination. So each combinatory of the untyped lambda calculus is going to be multiplied an infinite number of times to cater to each of the infinite number of types that are now generated. Hence what happens is that in most programming languages the statically typed languages like Pascal and Modula actually use this simple typing scheme for their functions and procedures and people claim that C uses it but C has a lot of dangerous things which also do not use it. For example this returning void in a function is not something that is really statically typable, it is actually an untyped form which is why you can do a lot of manipulation of types using those voids and using pointers in C. But for whatever is actually declared C does use the simple typing scheme that we have seen in the lambda calculus.

(Refer Slide Time: 43:42)



Therefore languages like LISP do allow integers and so on and so forth as types but mostly LISP is really an untyped language. If you remove all the data types from LISP and look upon pure LISP as a version of the lambda calculus then it is really untyped. There is absolutely no type checking mechanism, there is no type inferencing mechanism and there is no worry about whether you are applying some combinator to some argument which you should not be applying. So it is mostly untyped, once you have the typed data the underlying data type is well-typed then that typing often works for most of our run time environment. As long as the values are from those data types the typing works mainly because of the representation.

The representations are nicely ensured on the machine which ensures that you get reasonable values. And what holds is LISP also holds for scheme and C with its void construct is actually going into the untyped territory of the lambda calculus. So most of these untyped languages simply do not bother about typing though it is an important way of catching bugs at a very early stage and it is becoming more and more important. Though I have not yet spoken about name functions and procedures I have spoken about unnamed functions and unnamed blocks so far but essentially if you take all those blocks and give them a name you get your named functions and procedures and of course you should allow parameterization.

So what we started out in the simply typed lambda calculus was that its self application is really meaningless and no self respecting mathematician will use self application. But long time ago we actually looked at this combinator twice. And if you remember what we did was, a version of twice for the simply typed lambda calculus would be something like this, for the bound variables I have to specify types so I have done that otherwise I have made no other changes in the definition of twice. Because of the typing constraint since x is applied to y I have to give x a type of something arrow sigma but since x is applied to

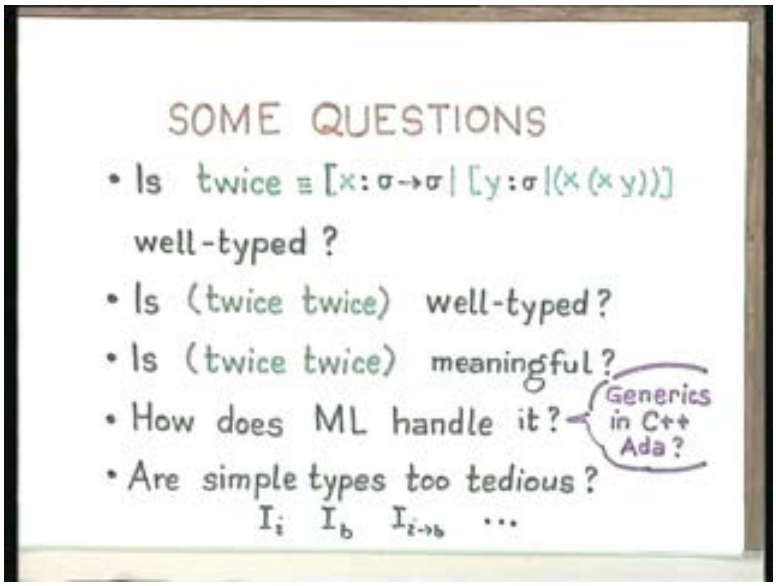
that the result of that I have given it a type sigma arrow sigma and via type sigma so that it is well-typed. So this is actually a well-typed expression.

Now the point is what about twice twice?
We had actually looked at this application also. The moment you put these type constraints on the simply typed lambda calculus twice twice is no longer well-typed. If you remember the fact that each of these sigma is something of the form arrow arrow arrow arrow arrow which ends up in a bool or an int you at once find that twice applied to twice is not well-typed. But we actually applied it and we got some nice result.

The next question is we actually applied it and got some nice results so is twice applied to twice actually meaningful? Are we being too restrictive, is it becoming like a dictatorship to put in simple typing and will this allow things like this?
When we apply twice to twice we actually got some results. You remember, we got the octupling function or whatever twice is applied several times all that made perfect sense. So what it means is that it means that all self applications need not necessarily be meaningless. It is true, you cannot apply a function from real numbers to real numbers to itself, it is not going to type. But there are enough functions like twice which look meaningful.

What does twice do?
It just takes any function as an argument and for any argument that function might have twice applies that function twice. This is really all it does which is perfectly meaningful. After all given a real number x and f is a function from real numbers to real numbers f applied to x is perfectly meaningful there is no problem with that. And what all I am doing by specifying twice is that I am saying you take any arbitrary function f on real numbers and apply it twice on whatever argument you get. I do not care what function on real numbers you are taking but whatever it is you just apply it twice and then give me the result.

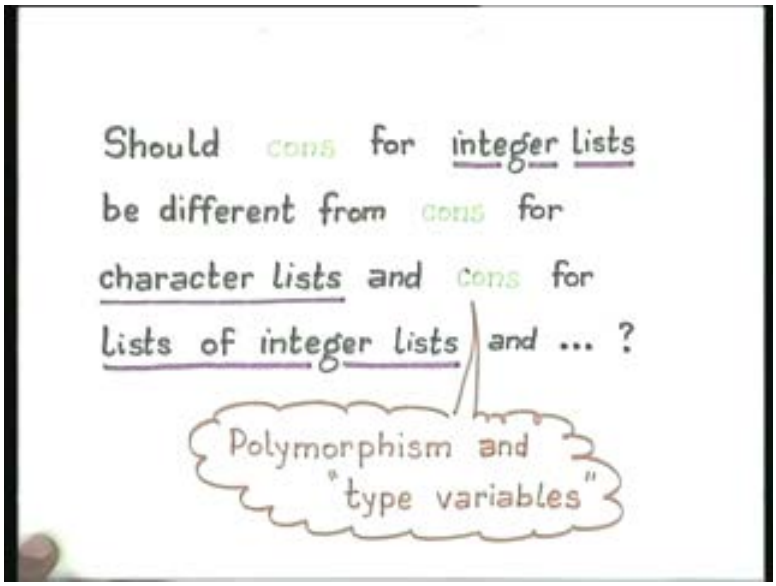
(Refer Slide Time: 50:21)



Therefore twice is a nice higher order function and it is actually in some sense type independent and there are lots of such functions. The important question that actually arises as part of all these is what is MLs view on types, secondly there is another important question, I said you are going to have infinite copies of the identity function.

Just imagine, just in order to give you back what you gave me I require an infinite number of copies which check what type it is send it to the appropriate copy and then send you back the same thing. So what about the code that is going to be written for something like the identity function? Regardless of the type of the argument the code is going to remain identical. There is no difference at all. essentially what the code says is take it and give it back, that is really all that the code says, take it and give it back even without looking at it but your simple typing scheme actually puts a restriction it says take it look at it and only if it is compatible with you send it back otherwise do not. So I will require an array of infinite number of programs which do nothing but take and give back. And we actually got this problem for whole lot of programming problems.

(Refer Slide Time: 51:39)



For example, what about the cons of integer lists?
Should the cons of integer lists be different from the cons for character lists? Should the cons for integer lists and character lists be really any different from cons for lists of integer lists or lists of character lists or lists of lists of integer lists, lists of lists of character lists and so on and so forth. Assuming that your base data type could have such an infinite collection then your simple typing only creates more problems, it creates tedium; you will have to create copies where only the type name is changed, whenever you get a new copy you will have to create a new program in which the type is changed. This is the problem with Pascal.

For example, if you define stacks of integers you cannot use that program for stacks of characters, you cannot use the program for stacks of strings, you cannot use that program for stacks of records of something or the other though the actual stack operations pop, push and empty are going to be identical in all these cases. And the reason in Pascal and Modula and so on you cannot do it is because they use a simply typed scheme. A simple typing scheme which requires an infinite number of copies where only the types of the bound variables have to be changed whereas in LISP you do not require this because it is untyped, it does not care what type you get and that is essentially the difference between the typing in ML and LISP or ML and scheme because ML and scheme are both statically scoped languages they are easy to compare. In scheme you can do cons of for any kind of type but that is because all types are regarded as being just the same type as being type lists.

The same cons is applicable to integer, integer star, integer lists, character star, character lists, integer list star, list of integer list and so on and so forth but that is because the cons in scheme is type lists and it is essentially like the untyped lambda calculus so it does not care what the argument is. In ML the cons is the same except that it is typed what is known as the polymorphic type. So you use the same code but now our base types are

type constants, what you require are type variables which are going to be instantiated on demand. So type variables are required. What we are saying is that if you look at the cons operation then for all types as long as they are types of the form something list for all types T such that an argument is of type T and another argument is of type T list it is possible to do a cons of the object of type T with the list of type T list and I require the same piece of code, I require only one copy of cons for that I do not require an infinite number of copies.

Therefore, we will look at the polymorphic lambda calculus where we actually move from the simply typed to the parametrically typed and this polymorphism is what is present in C + + and Aida as generics. For example, you define the stacks in Aida you use a typed variable which is not going to be instantiated and you write all the code for the stack operations pop, push, checking emptiness so on and the compiler compiles it you call this code for producing stacks of integers, stacks of characters, stacks of records whatever but at the call to this code the typed variable is initialized to integer, the typed variables are different from value variables so you have a notion of typed variables which are different from value variables which can be instantiated on a call. So you produce particular instances of the same code for the same type.

the simplest implementation of course is that instead of you writing the code for integer stacks, real stacks, character stacks and so on you separately you write it as a generic package in Aida or C + + and the compiler will produce code for whatever type you are demanding those operations to be used. It will actually replicate the code by changing the typed variable and putting the base type that you are entering there. In fact this is what most of the Aida compilers do they actually replicate the entire code for that call. But it is also possible to use reentrant code, use the same code with the typed variable instantiated which is what the ML does. You can use reentrant code without actually generating new code. So we will talk about polymorphism in the next class.