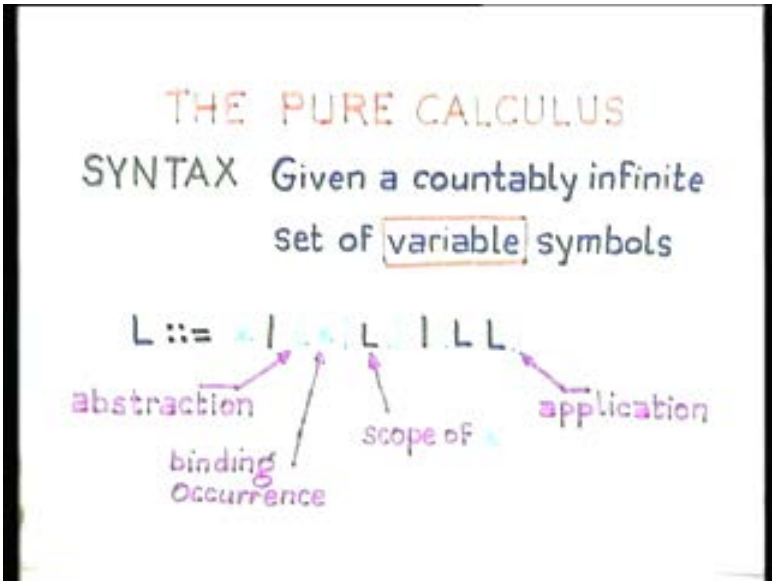
Principles of Programming Languages
Prof: S. Arun Kumar
Department of Computer Science and Engineering
Indian Institute of Technology
Delhi

Lecture no 28
Lecture Title: Data as Functions

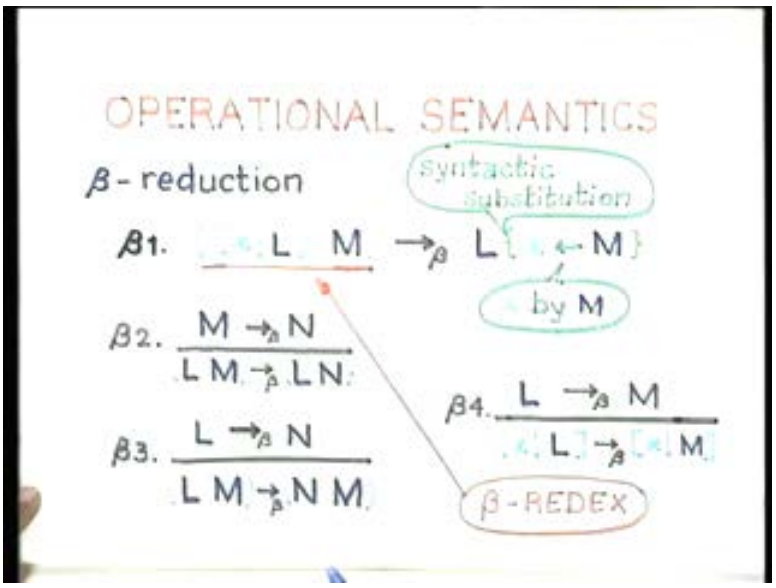
Welcome to lecture 28. We will briefly recapitulate whatever we have done in the λ calculus and then I will talk about data as functions. Let us just briefly go through the pure calculus. We have the syntax given a count-ably infinite set of variable symbols. Every variable symbol is a λ term. The λ abstraction is a λ term and the λ application is also a λ term.

(Refer Slide Time: 00:48)



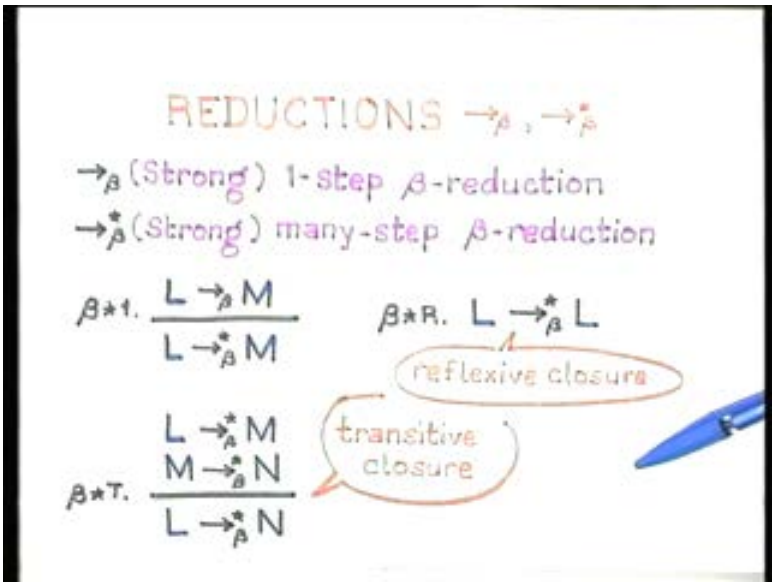
We have the usual notion of free variables. The closed λ terms which have no free variables are called combinators. We will be looking at some important combinators and the operational semantics of β reduction is as follows.

(Refer Slide Time: 01:18)



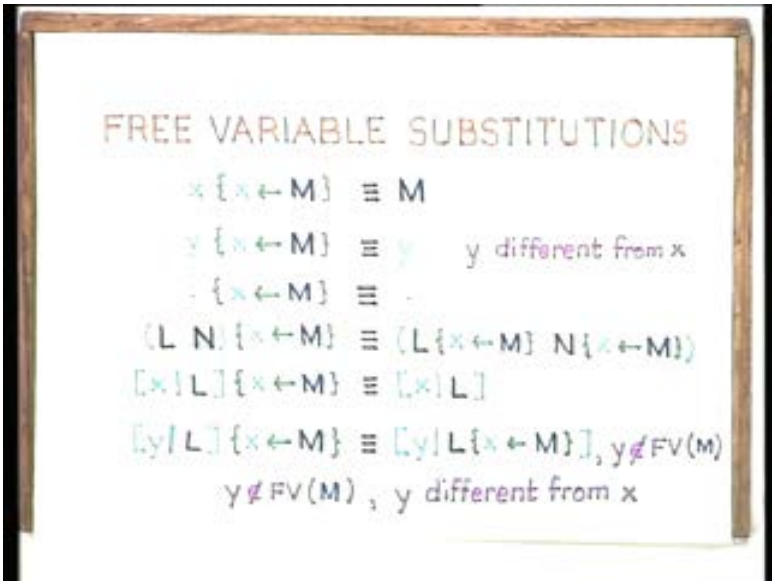
The main rule is that of finding an application and replacing the bound variable in the body by the operand which simulates function application and there is the notion of syntactic substitution which we have rigorously defined before.

(Refer Slide Time: 01:48)



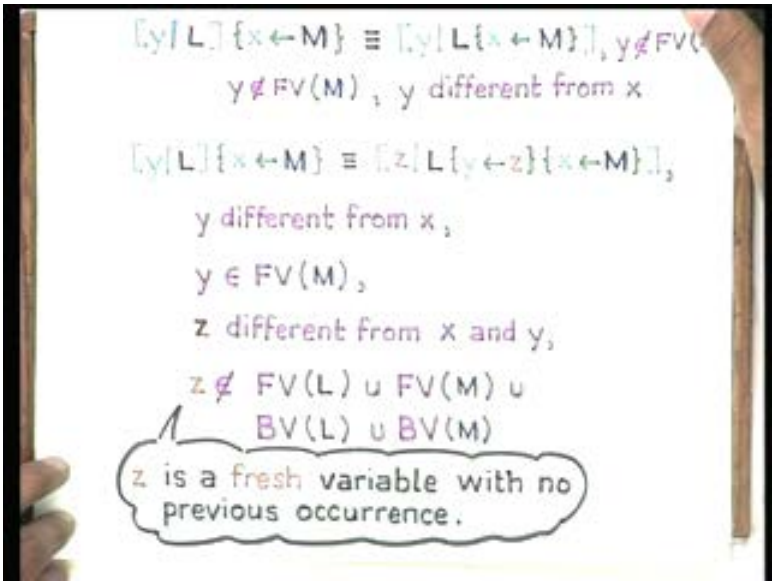
We can take the reflexive transitive closure of the β reduction and obtain a many step β reduction by these rules, just to make clear what our notion of free variable substitutions are. As I said for our β redex we require the notion of free variable substitutions which somehow has to be very carefully defined.

(Refer Slide Time: 02:16)

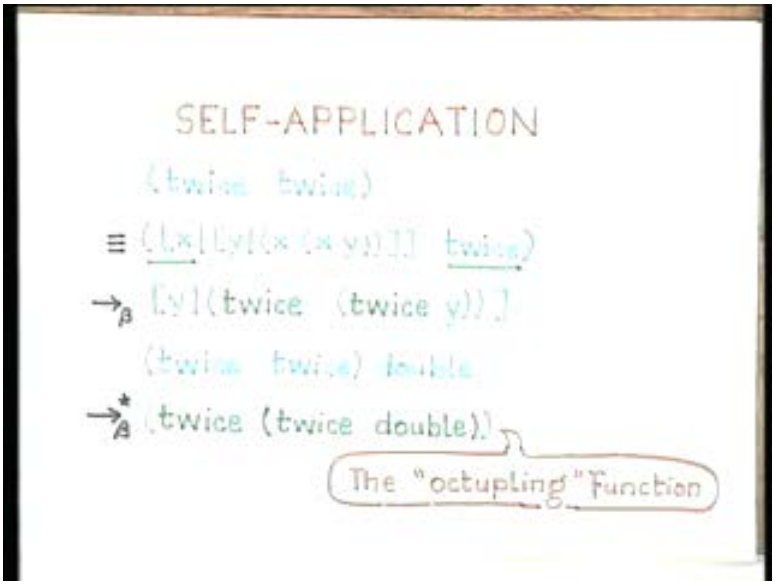


We will take this as a definition, though it is not completely algorithmic. It is possible to give this definition entirely in terms of free variables without worrying about these bound variables and then make it algorithmic but then that is harder to understand. We will follow the simpler route here.

(Refer Slide Time: 02:54)

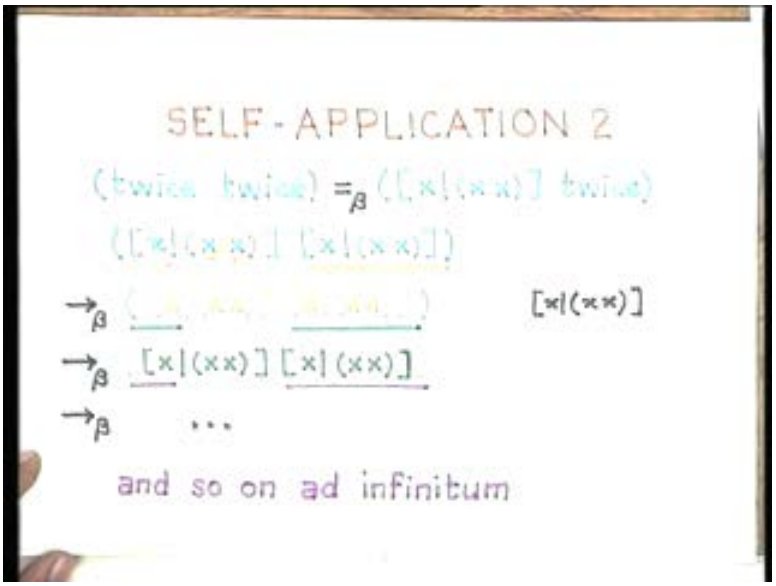


(Refer Slide Time: 03:12)



We also looked at some strange kinds of things which normally do not happen with functions. For example, we saw that functions may be applied to themselves something that is otherwise unheard of and actually it does make sense for a function to be applied to itself. On the other hand it also does not make any sense for certain functions to be applied to themselves because they lead to infinite-non terminating computations and so one has to be careful.

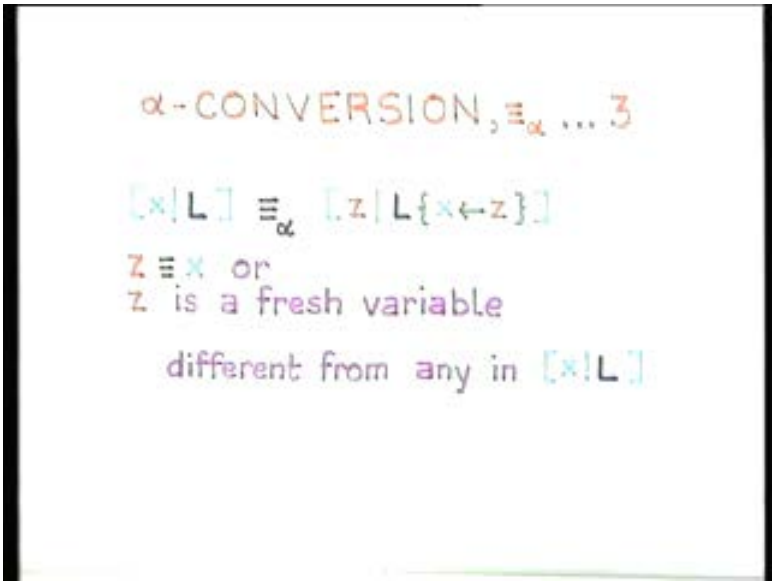
(Refer Slide Time: 03:53)



We will see that self application by itself has to be banned and then we also looked at α conversion which is essential somehow for readability and for doing a β conversion without

creating collisions or confusions. This definition of α conversion uses the definition of free variable substitution and also this α conversion includes the syntactic identity.

(Refer Slide Time: 05:03)

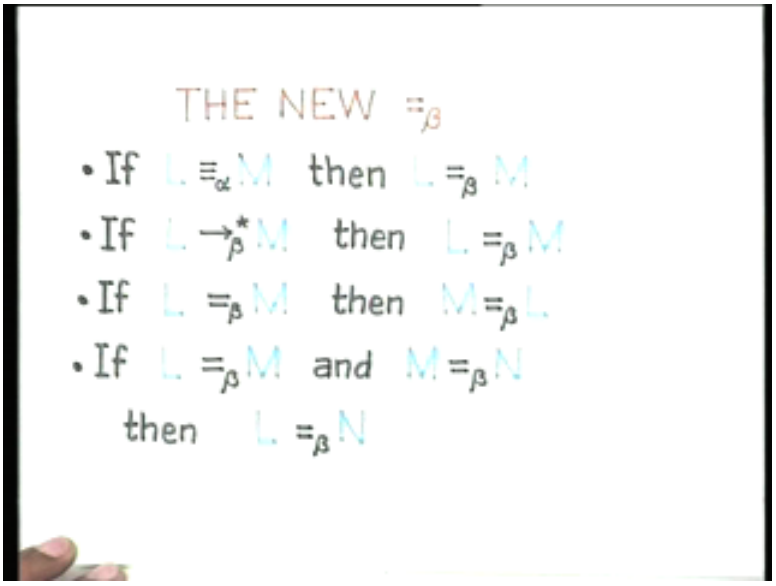


L is α convertible to itself. We will take z to be a fresh variable different from any in this. Again it is possible to redefine α conversion such that it is only necessary to look at free variables in certain bodies. This is the notion of α conversion. The α conversion basically tells you that two terms which are different only in terms of bound variables are essentially the same. They are almost syntactically identical and that you really cannot distinguish them on the basis of any meaning or operational semantics. We will include α conversion in our definition of β equality. The new β equality is defined as follows: if L is α convertible to M then L is β = N and if L goes in a many step β reduction to M then L is β = M.

If L is β = M then M is β = L.

If L is β = M and M is β = N then L is β = N.

(Refer Slide Time: 05:42)



These three were the original rules which really tell you about the reflexive transitive closure of a many step β reduction. This is a new one which says that every now and then while doing β reductions we might feel compelled to rename some bound variables therefore convert α to another term which is not syntactically identical but which is α equivalent and we will claim that they are both β equal. So, our reflexive transitive closure actually includes α convertibility too. Now we do not need to worry too much about α conversion which adds to the confusion and does not add anything new to the meanings of terms.

Now getting back to the title of this lecture namely data as functions it is important to realize that there are essentially several models of computation. When we get back to this data as functions if you go through an architecture course firstly, it tells you that there is really no difference between program and data. Program and data both have the same representation namely various kinds of bit representations and they have the same representation. So, it is a matter of interpretation whether some bit string is a program instruction or a piece of data.

The whole idea of the Von Neumann architecture which is what you study in most architecture courses is really that there is nothing fundamentally different between programs and data. Programs are not any different from data. It is a matter of interpretation. You can interpret certain bit strings either as data or as instructions. So, there is no difference between data and control and the important feature about any architecture course would be that all control and all programs are represented in the form of data in the sense that the machine instructions are all expressed in some bit strings; your jump instructions are bit strings and so they are all coded into data. All control is coded as data and data is coded as itself. The whole point is that depending on convenience very often you logically partition areas of memories into a data segment and a code segment and you interpret whatever is in the code segment as control and whatever is in the data segment as data. But it is largely a matter of interpretation and there is no essential difference between the two.

In the case of the λ calculus, Church showed that programs and data are no different or functions and data are no different and that all data are functions and that is a reversal part of the fundamentals of architecture. There is no difference between programs and data or control and data. In this case it is functions and data and all data can be represented as functions.

So far we have not only looked at the pure λ calculus but also what might be called the pure untyped λ calculus. Firstly, we must realize that in theory it is not really necessary to have to apply the λ calculus to have an applied λ calculus. In theory it means that our application like taking the pure λ syntax and applying it on to Peano arithmetic is not essential. You can get rid of Peano arithmetic completely. Since all data are going to be represent-able as functions, Peano arithmetic can also be represented as functions. So, it is possible to work with just the pure λ calculus. You do not need to apply it. It is a matter of taste whether you want to apply it or not but largely if you look at functional programming languages they are all applied λ calculi but they are all applied λ calculi for reasons of efficiency.

The trend in the last 50 years of the existence of computers has been that you should be able to exploit speed of hardware. Hardware is very fast and anything that is programmed in hardware is likely to be very fast. Instead of actually coding up all data structures as functions it is more efficient to use the underlying data structuring capabilities of your machine hardware. You want the power of the λ calculus which means higher order functions but you do not want to work with a pure λ calculus simply because even though the pure λ calculus can represent all the data you require, it is going to be extremely slow. So, you apply it on to an existing virtual machine whose operations are likely to be very fast.

The underlying hardware is going to be very fast and it is getting faster everyday. So, it makes a lot of sense not to try to code everything in the pure λ calculus but to use the underlying data representations of the underlying hardware and code only what is very difficult to do with the underlying data representations. Remember that the whole Von Neumann thesis is not that you cannot write higher order functions in hardware but it is just that it is extremely hard and complex to do it. So, you exploit the hardware to the hilt, for example, the integer arithmetic is very fast on hardware. No amount of simulation using list is going to get you that speed. Whatever is programmed in hardware or at most firmware is likely to be 5 to 10 times faster than the best algorithms you can write in software.

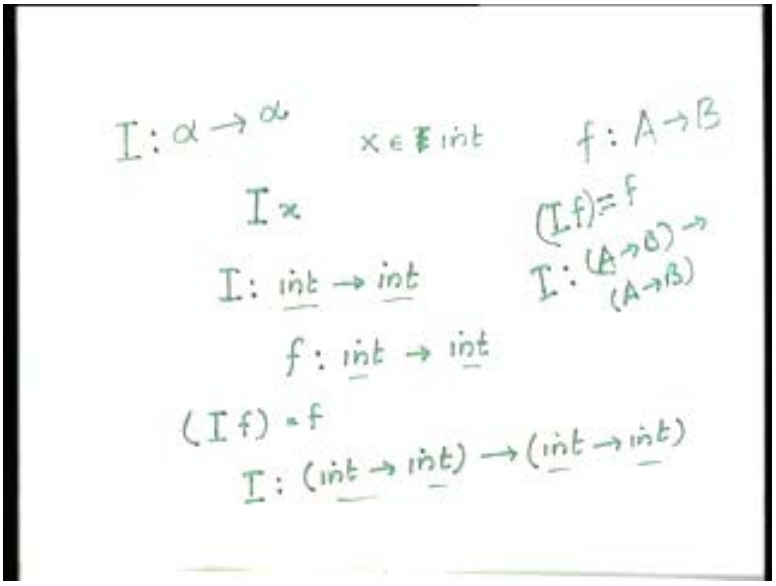
It is a good pragmatic reason to use only the applied λ calculi which is why all functional languages are applied λ calculi. They provide this excellent structuring facility for higher order functions but even though the pure calculus can structure data just as well as data itself, it makes a lot of sense to apply it on to an existing virtual machine and get the benefit of the structuring mechanism of the λ calculus coupled with the speed of the underlying virtual machine. That is really the main reason why all functional programming languages can just be regarded as applied λ calculi. But as a matter of academic interest it is a good idea to know that it goes in parallel with what you learn in an architecture course. In an architecture course essentially you learn the representation of all data and functions as data. If an architecture course has to be summarized in one sentence it is just a representation of all data and functions as data. It is a good idea to look at

the pure λ calculus theoretically at least to be able to program all the data themselves as functions and so it provides a parallel.

First, let us go through some of the important combinators that we will see. Then we can look at numbers and data structuring facilities. The three most important combinators are these. What is a combinator? A combinator is a closed λ expression. It has no free variables. Closed λ expressions provide the capability as they have the status of full programs to which data can be supplied and they can be executed. This I is the identity function. You apply it to any object and you get that object. Since we are talking about that n type λ calculus, that object may be a value object or it might be another function. This combinator 'I' is polymorphic in the sense that there is nothing specified about this type of X. Whether this type of X is a value object or a function object or a higher order function you can apply this 'I' to any function and you should get back the function. Let us suppose that you are using this 'I' in an applied λ calculus in which you have a function from some domain A to some co-domain B. In the function f A $\rightarrow$ B when I is applied to f then 'I' acquires the status of having a type and A $\rightarrow$ B goes to A $\rightarrow$ B.

If you look at an object X say in integers and you apply I to X then 'I' is a function of the type integers to integers. On the other hand if you took a function f from integers to integers and you apply 'I' to f you get back f. Then 'I' really has the status of having the type integers to integers. That means it takes a function 'integers to integers' and gives you another function which is integers to integers and that is actually the same. You can take any function of any type and 'I' applied to that function will give you back the same function. In each case if you have a function f which goes from some type A to some type B then 'I' applied to f is equal to f and therefore I has the status of being from A applied to B $\rightarrow$ A applied to B. $I : (A \rightarrow B) \rightarrow (A \rightarrow B)$. 'I' is polymorphic in the sense that depending on the argument, it has a varying personality. If the argument is a plain integer then 'I' has this personality of a function from integers to integers. If it is a plain Boolean then 'I' has a personality which is like Booleans to Booleans. If 'I' is applied to a function from integers to integers or integers to Booleans then 'I' takes a personality which is of a type of integers to integers to integers to integers or integers to Booleans to integers to Booleans.

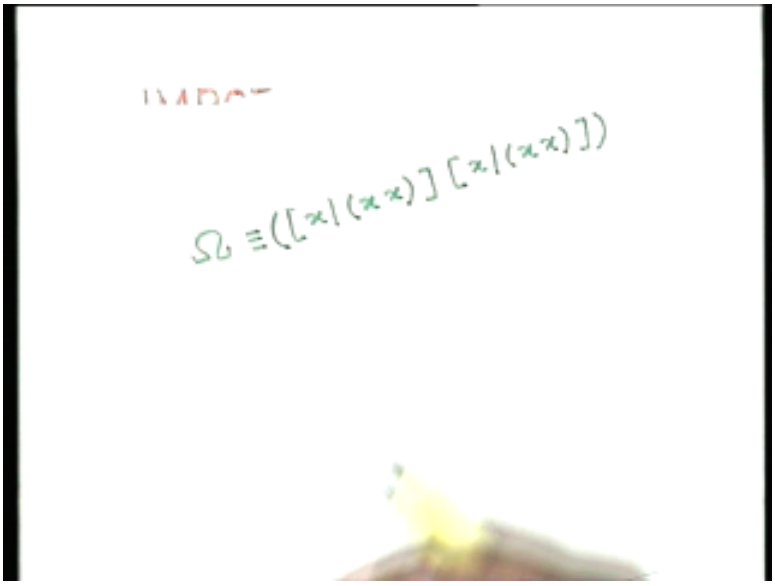
(Refer Slide Time: 21:52)



In general instead of looking at these various types formed by constants just as I can talk of value variables, I can talk of type variables, '$I : \alpha \to \alpha$'. These types are what you see in your ML interpreter. When you define a function it determines the type, if it can determine the type by means of a constant that is if there is a distinguishing constant then you can look upon types themselves as a language whose constants are things like int, bool, real etc. The functions are arrow types formed of these. In the case of a combinator like I, it is saying that you take any type α. So, α is a variable over the types and I is a function then of type $\alpha \to \alpha$.

We will talk about polymorphism in a type λ calculus in a little bit of detail. A large part of it still is a matter of fairly current research but there is quite a bit of material which is already quite well known especially ML type inferencing etc. Similarly, you can assign most of the other combinators a type in that way. Twice is also polymorphic in a similar fashion in the sense that I can assign type variables to the type of twice and then I can try to solve for those type variables. Solving entirely for type variables is expressing one type variable in terms of the other.

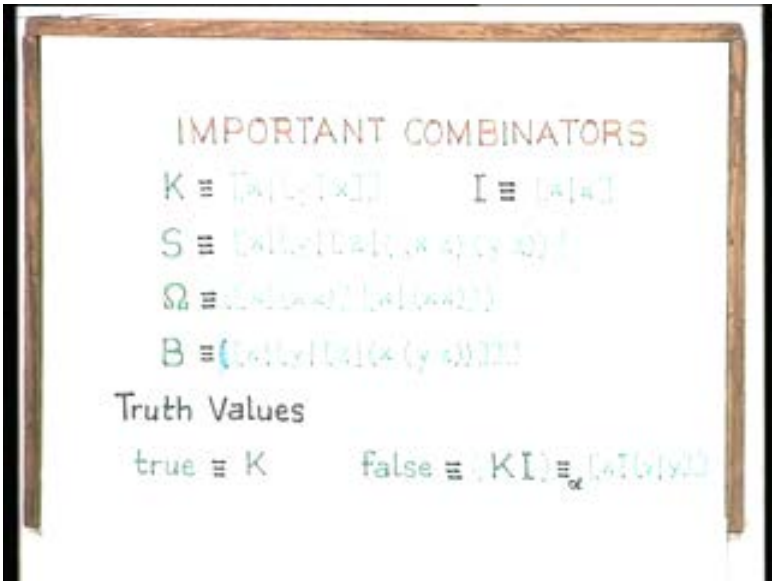
(Refer Slide Time: 24:24)



If I cannot do that as in the case of Ω (omega) combinator that I showed twice self application is polymorphic whereas the Ω combinatory, defined this way; $\Omega \equiv ([x|(xx)][x|(xx)])$ is not polymorphic. It is possible to do a type inferencing for twice but it is not really possible to do a type inferencing for Ω. It is possible to assign some type variables and some type expression to twice but it is not possible to do that for Ω and that is what distinguished just genuine self application from merely polymorphic application. Let us look at combinators now. Just keep in mind that we are starting at the beginning. In the beginning there is nothing. You are extracting functions, values and everything from essentially a homogenous mass of nothingness.

Here is a combinator K. If you look at the combinator K, basically given two arguments X and Y it returns the first argument. Then there is this combinator S which is actually very important. For the three arguments X, Y and Z it is of the form X applied to Z the whole thing applied to Y applied to Z. Its importance is really that with S and K you can program all other combinators meaning, they are the absolute primitives that you require. With just these two constructs essentially all functions that are programmable can be programmed.

(Refer Slide Time: 27:30)



It is a matter of study after you have finished your theory of computation course but till then you just keep in mind that S and K are very important. This combinator just tells you about how to compose functions together. Function composition is again polymorphic in the sense that regardless of the domain in which you are, you can always compose unary functions. This is a function composition. Then let us look at actual representation of data.

Firstly, let us look at truth values. The truth value 'true' is just the combinator K. Given two functions X and Y it chooses the first one and that is true. False is just, choosing the second one given two functions X and Y. I have given two objects X and Y. If you choose the second one and that is false you can see the similarity. Here it is functional but in your architecture it is really data based. You have only two possible values of data from which data is all formed 0 and 1. One of them is false and the other is true. It really does not matter which one you take.

We will just quickly look at the data structuring capability. We have to be able to define various data structuring facilities. What are the data structuring facilities? They could be Cartesian products, probably disjoint unions (but let us not think about disjoint unions now) and sequences. We need to construct sequences. So, if you have got pairs, tuples and sequences you have essentially got everything else that is required theoretically but is not required pragmatically. As far as functions are concerned you have got them. It is the whole purpose of the λ calculus.

We will define a pairing function. I am following a convention now. I am using dark green for combinators constructed from λ expressions. Do not get confused when I use a green square bracket. Do not confuse it with the blue square bracket which is part of the λ calculus language.
Here, this is a pairing function which takes two arguments and this pairing function is just defined by this combinator. In all our data structuring facilities we had essentially two kinds of operations. One is the constructor operations, which allows you to construct complex data from simpler data and the other is the deconstruction; how you get the simpler components from a complex piece of data.

This is the constructor for pairing and this is the deconstructor. I will call the deconstructor this and it means that if I have the λ expression P which is actually supposed to denote a pair of elements which presumably was formed through this construction operation and not anything other than P applied to true will give me the first component of this construction and P applied to false will give me the second component. You can verify them by actually doing the λ application. Remember that this bracket is my representation of the deconstructor operation. It is in green. This blue parentheses represent actual λ application so that means you take the combinator P to which you apply the combinator true and see what you get through β reductions and of course our β reductions now include also α conversions wherever necessary.

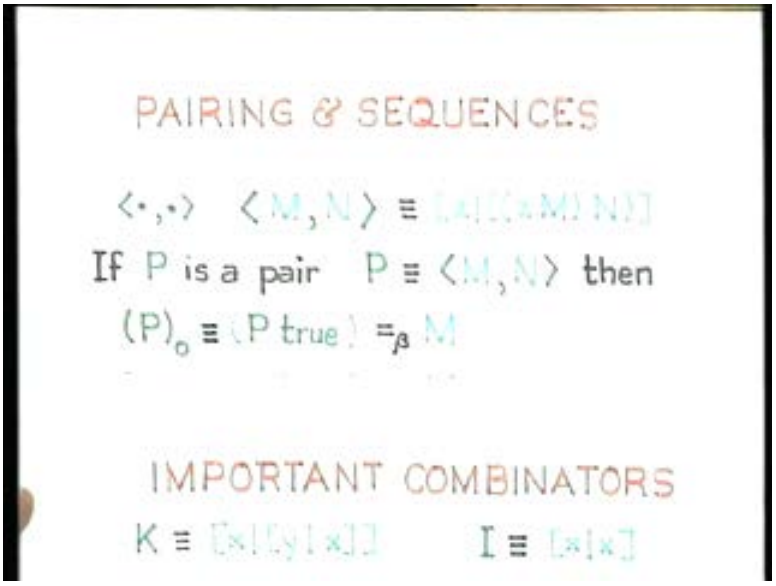
If true anyway is the combinator K, which given two arguments, gives you the first argument. For example, take this. P could be any λ term and you could apply this deconstruction operation on any λ term. It is just that if you apply it on any λ term then the chances are you will not be able to interpret what you are getting in some reasonable fashion. But if P had been constructed through two terms M and N using this constructor then when you apply true here you get true applied to M and the whole thing applied to N. True applied to M true is just K; K applied to M applied to N means that it will give you the first component and it will give you M.

Similarly, P applied to false will give you the second component and it will give you N. But remember that since we are in an un-typed world you can actually construct any arbitrary λ expression and say that this somehow represents a pair though you may not have constructed it through this fashion. You can apply that combinator to true and get some result which does not make any sense to you.

This is sort of natural. Here again we actually had an inkling of that in this. If you say true has the same representation as K then can I apply K to a pair and claim that I am applying true to some pair and getting some truth value. The type of value you get is largely a matter of interpretation. I used to have these FORTRAN programs and then run them through the Pascal compiler just to see what kinds of errors come. It is really a matter of interpretation meaning you can send any file into a Pascal compiler and what you get is anybody's guess. Similarly, you can apply any combinator to any combinator. After all the language syntax does not disallow you from doing that but what you get is undeterminable.

As far as these combinators are concerned, the fixing of the types and the fixing of the kind of values is largely a matter of interpretation. I can take cosmic ray data and try to execute it. I get a whole sequence of bits, 0's and 1's so why can I not regard it as a program and try to execute it. Some glitches will take place somewhere and something will happen but it is a matter of interpretation whether you get something meaningful or not. In an un-typed world, which is what your underlying hardware is, there are no types and there is no distinction between programs and data. So, what prevents me from executing data? Similarly, in the un-typed world of the λ calculus what prevents you from applying some strange combinators to other strange combinators? What you get is unknown.

(Refer Slide Time: 36:20)



I can interpret the construction and deconstruction operations only under certain conditions. For example; what prevents me from just incidentally forming an expression which has this syntax. Why should I interpret it as a pair at all? Similarly, there is nothing that prevents me from doing this application to any arbitrary λ term and seeing what I get. What I get is not necessarily the first element of a pair. I would get the first element of the pair only if that λ term was obtained by this pairing constructor. This is a fact in the un-typed world. Whether it is program or data it does not matter. In any un-typed world there is a problem of interpretation that is these operations are not serjective. They do not go back and forth necessarily always, but they go back and forth only if you go through the construction operation and then the deconstruction operation.

That is one of the problems of the un-typed world meaning you can take an arbitrary λ term and deconstruct it first and then try to do the construction and see whether you get the original λ term. If your arbitrary λ term was not of this form then you are not going to get the original λ term. If your arbitrary λ term was of this form then you will get it but if your arbitrary λ term was not of this form then you are not going to get it because when you do the deconstruction you will get some strange λ term M and a strange λ term N and then you will apply this construction and you will get a λ term of this form which is not necessarily going to be β equivalent to what you started out with.

One may think that when we started with true and false we expect something like what we do in the Boolean world. But the problem is that you are mixing up an un-typed world. For instance why should true + false = 1? We may think that when we take a function, apply true to it, take a function apply false and then in some way try to get it back, we always get back the same function. But it is not so because what is guaranteed is that if you go through the constructor first and then do the deconstruction then you will get back what you originally had. If you do a

deconstruction and then try to apply a construction then there is no guarantee what will happen. That is how the un-typed world is.

What Barronreck has proved way back in 1974 is that there is no perfect possible construction and deconstruction operation for pairs which will ensure that construction and deconstruction are somehow inversions of each other under all circumstances. It means that if you do a deconstruction and then do the construction from what you get out of the deconstruction you will not necessarily get what was original. It is a good idea to play around with these things to see what happens with your data structuring. Why should we play around only with Pascal compilers and FORTRAN programs? We can do that also with combinators. Apply the successor function to a sequence and see what you get. After all there is nothing that really prevents you from seeing what extremely complicated expressions you get. This is how the pairing goes and once you have the pairing it is very simple to construct tuples.

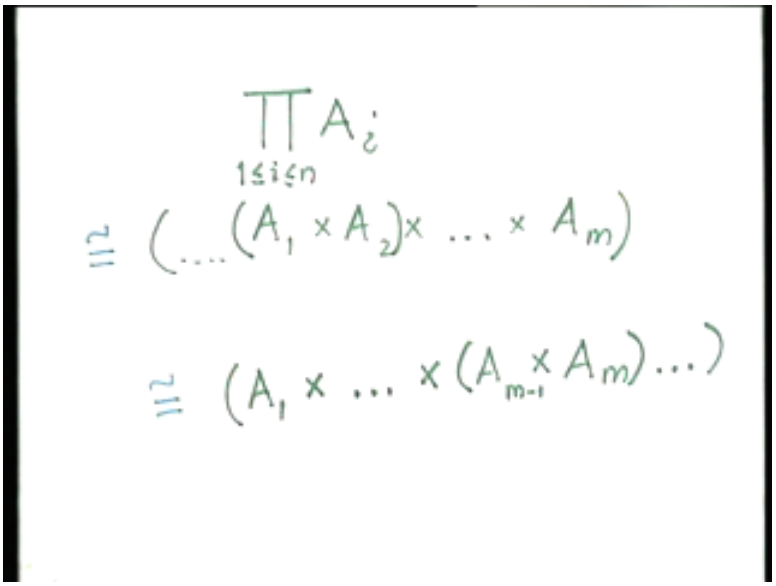
So, the pairing is through Cartesian products and tuples are also Cartesian products. Let me take a Cartesian product $(....(A_1 \text{ x } A_2) \text{ x}....\text{x } A_m)$.

I can look upon this Cartesian product which is obtained by a binary Cartesian product, take the Cartesian product of this with the next and I keep on bracketing.
The other possibility of course is that I can look upon this Cartesian product as looking at such Cartesian products so that I get $A_1 \text{ x } \ldots \text{x } (A_{m-1} \text{ x } A_m)...)$.

I can look upon an enary Cartesian product which I will just write like this $\prod A$ as being isomorphic to binary Cartesian products.

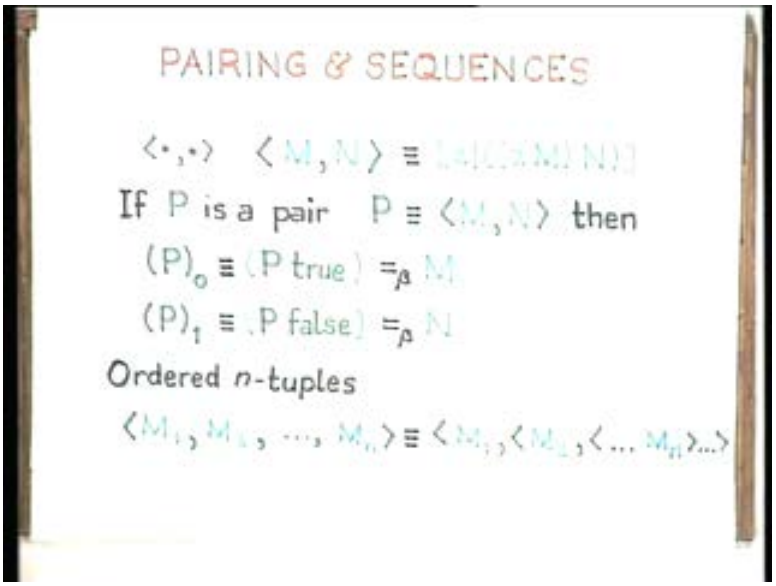
This is isomorphic to $(....(A_1 \text{ x } A_2) \text{ x}....\text{x } A_m)$ is isomorphic to $A_1 \text{ x } \ldots \text{x } (A_{m-1} \text{ x } A_m)...)$.

(Refer Slide Time: 42:15)



So, I can take an enary Cartesian product and either I can look upon it as binary Cartesian products done one way or binary Cartesian products done the other way or I can also have mixtures of these. I can choose some k and somehow do A1 to Ak and then Ak + 1 to Am and take it as a binary Cartesian product. So, it is isomorphic to a whole lot of possibilities. It is actually isomorphic to all possible ways of bracketing that you get. We will just take one of these. We will look upon a tuple as just obtained as an ordered pair. A tuple m1 to mn is an ordered pair whose first component is m1 and whose next component is n - 1 tuple whose first component is m2 and whose next component is n - 2 tuple etc.

(Refer Slide Time: 43:48)

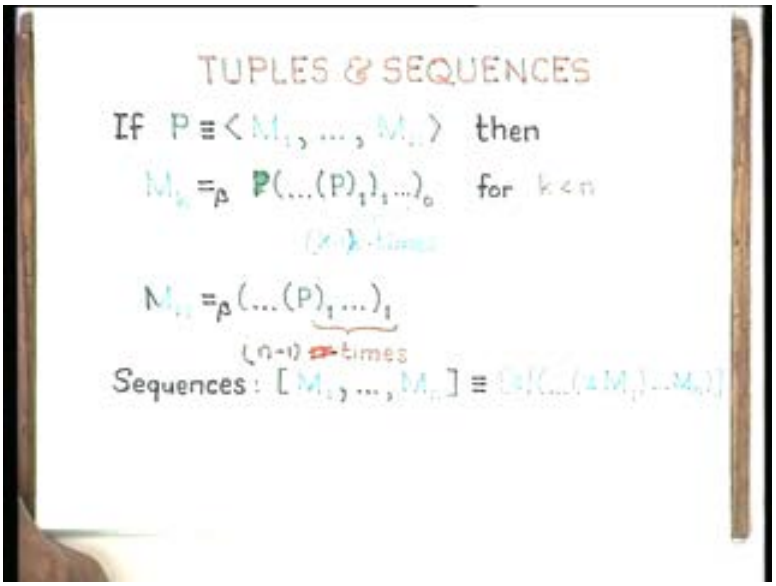


Since we have a pairing construction operation we know how to find tuples. Essentially, we are using one of these isomorphisms for the representation of tuples. Since our tuples are constructed from pairs that are deconstruction operations for tuples the projection functions for the tuples can also be constructed from the deconstructions for pairing. If you keep on deconstructing the tuple in a certain fashion you can get the I^{th} component of the tuple which essentially means that if p has been obtained by an explicit tuple construction mechanism then to get the k^{th} component of this tuple I do the paired deconstruction k - 1 times.

If I have a tuple of this form; $< M_1, < M_2, < \ldots M_{n-1}, M_n > \ldots >$ if I do a deconstruction for pairing n - k - 1 times then I have a tuple whose first element is M_k and whose next element is another tuple of n - k elements, it is a n - k tuple and now I take the first component of this and that is what actually happens here. You take the right hand component each time of this pair, do it k - 1times and then take the first component of what you get. For the n^{th} element you always take the right most element. So, you do this n - 1times and you always take the right component here. These are the deconstruction operations for tuples derived from the deconstruction operation for pairing. We had the first confusion with these parentheses because these

parentheses when in blue are λ applications and parentheses when in green are deconstruction operation for pairs as tuples. Now let us look at sequences. For sequences I have square brackets. Square brackets when blue are λ abstraction and square brackets when green are actually sequence constructions. If you actually look at your programming the tuple formation is really like your list constructions in ML or LISP etc. It is like having a cons operation.

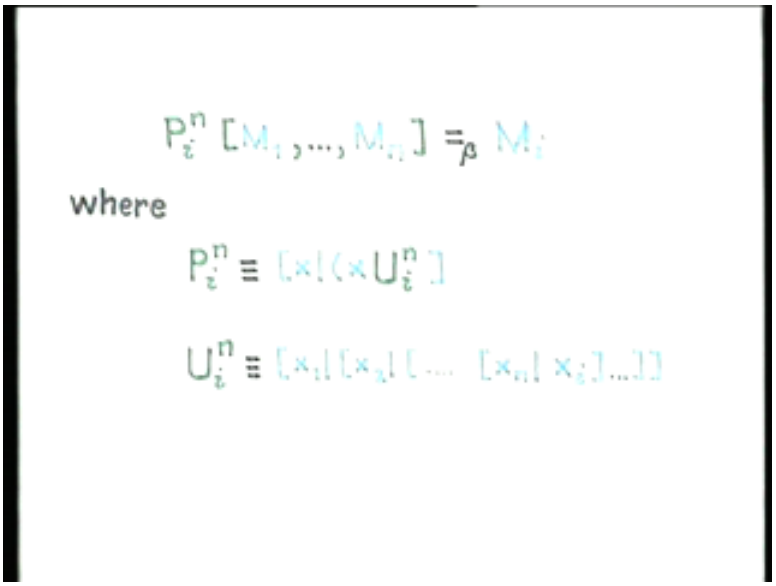
(Refer Slide Time: 48:09)



A cons operation is really like successive pairing. There is an alternative way of looking at them. You can look at sequences this way; whole sequences as just λ Z, Z applied to M1 applied to M 2…..up to Mn. Then you have this deconstruction operation. If you actually play around with it you will be able see that this is a deconstructor. This P, i, n is the deconstructor which extracts the I^{th} component of the sequence. If that sequence is n components long then the I^{th} component out of n is going to be this and we are in the un-typed world. So firstly, you can apply this P, i, n to any λ expression that you want; it may not even represent a sequence but you can still apply it and get something. You would not know what you apply and what you get after application. You could even take a sequence that is much shorter than n elements long and try to apply P, i, n where i is greater than the length of that sequence.

You will get some other λ term because a λ term applied to a λ term will give you a λ term but you would not know what it means again. Here again only when you go through the constructor operation and then do the deconstruction you are liable to get back your original component. If you go through the deconstructor operation and then go through the constructor you do not know what you will get and in the un-typed world there is no such thing as an error. Errors are a logical consequence of an interpretation. In your actual hardware you actually set some condition codes and detect certain patterns as errors. Otherwise if you just look at the bare hardware there is no such thing as an error. It is just bit patterns.

(Refer Slide Time: 49:22)



Similarly here, the concept of errors again is a matter of interpretation and you just get λ expressions. Errors are really higher level abstractions from the un-typed world, meaning, they are not part of the un-typed world. If you do erroneous applications you will get something but to interpret it as an error is a matter of your taste. It is a matter of what you consider right. For example; you may not like the idea of a deconstructor followed by a constructor application giving you actually something that is meaningful and then you call it an error. You will still get a λ term whether you like it or not.