

**Principles of Programming Languages**  
**Dr. S. Arun Kumar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture - 29**  
**Data and Fixed Points**

Welcome to lecture 29. We will just briefly review what we did last time and continue with fixed points. I showed you some important combinators yesterday K, S, omega, B and there is another important combinator W about which we shall learn. We were looking at data structuring in the lambda calculus as how to represent data as functions. Here the truth value is true and false, of course we are still in an **un-typed world**, we will talk about numeral representations but you can ask for example whether a numeral is true or not.

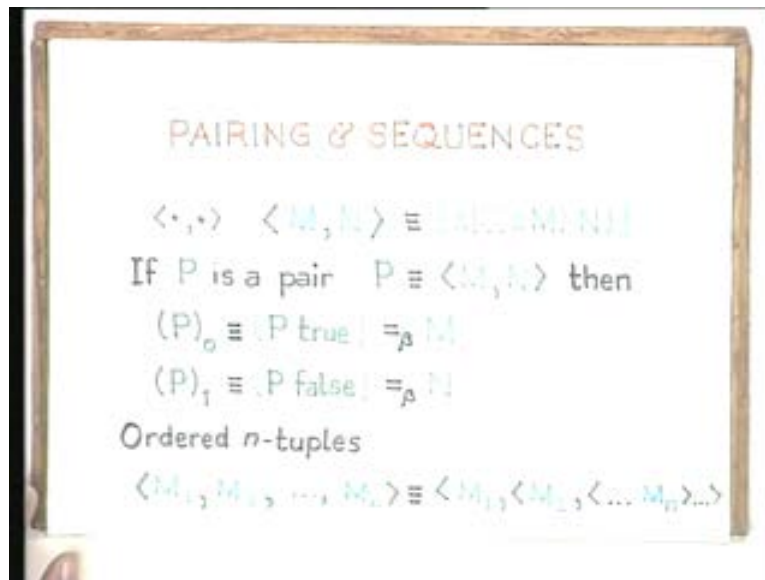
(Refer Slide Time: 00:50)



It is syntactically possible not that it should be allowed and you might actually get some answer but that is the problem with the un-typed world. Then we also had the pairing function again with its constructor and deconstructor. You can of course apply the deconstructor if you have already applied the constructor then you are guaranteed that you will get the components which were used for the construction of the pair but you cannot apply a deconstructor and then apply constructors and expect to get the same term back.

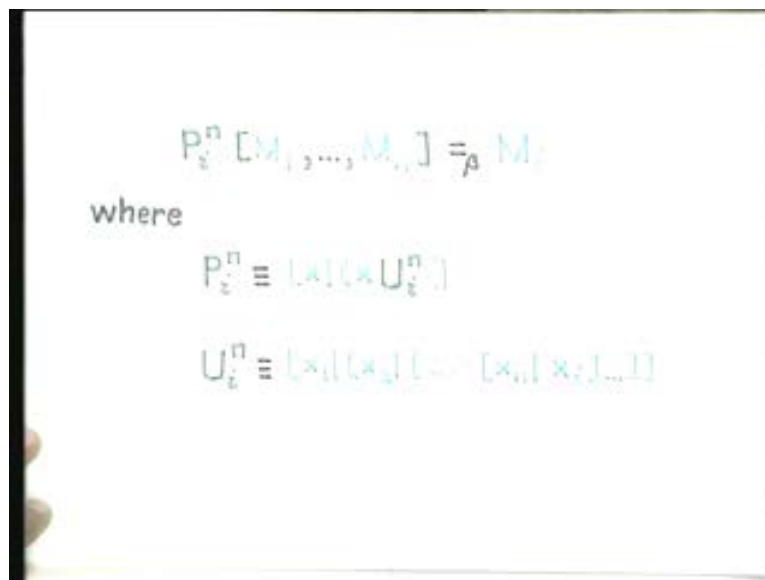


(Refer Slide Time: 1:30)



Hence using pairing inductively we might define  $n$ -tuples for both the construction and the deconstruction. We can also define sequences in this form for which this is the constructor and you will have destructors which look like this. So you want to be able to project out the  $i$ th component and get this and if you use this combinator  $U_i$  in which has  $n$  abstractions then you put that into this and make this application then you will get the  $i$ th component of the sequence.

(Refer Slide Time: 2:38)





And of course the last and most important thing is how you represent the naturals in the lambda calculus. We will look at the representation of the numerals. The representation of numerals is actually a very important feature of the lambda calculus. But the only thing is that there are lots of different representations of numerals.

As I said Church's original agenda for the lambda calculus it was able to show what exactly is computable and what is not. There was a growing body of work called recursive function theory which was really a set of functions which people thought could be mechanizable at that time and the natural numbers formed an important part of that. So, to represent the naturals this is not Church's original notation, original representation, his original representation is a little more complicated and it does not satisfy some nice properties.

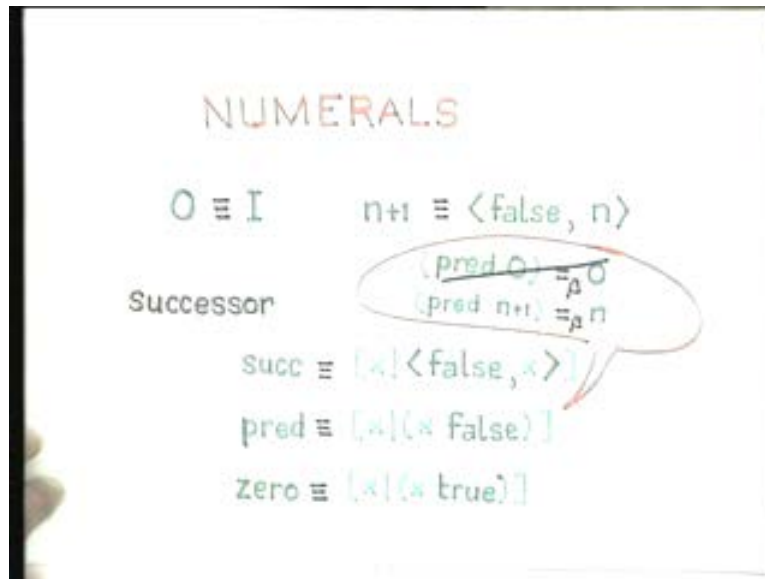
So you just take the  $i$  combinator the identity function itself as 0 and this  $n$  plus 1 I am writing  $n$  plus 1 where I could have written  $n'$  may be but this whole thing is not supposed to be an addition but it is supposed to be a single combinatory. This single combinator is just the ordered pair false with the representation of  $n$ . Since it is an inductive definition essentially what it means is that a natural number, the 0 is represented by the combinatory  $I$  and any other number is actually an ordered pair of false so there will be some  $n$  falses and then  $I$  except that they all be nested within the pairing functions. So it is a pair containing false and another thing which is a pair containing false and something else which is a pair containing false and something else so on and so forth.

And of course what you require is the successor function in our Peano Arithmetic and this was a successor function which is just defined in terms of pairing. So, given any  $x$  the successor of  $x$  is just the ordered pair formed by false and  $x$ . And we can actually take the predecessor function and the predecessor function is just for any  $x$  its predecessor is obtained by applying that  $x$  to the combinator false. Of course 0 should not have a predecessor but then there is no question of undefinedness and so on and so forth in this case so the predecessor of 0 is taken to be as 0 itself. So if you actually apply  $i$  to false you will get false so this is not.

This is one of those expected things in an untyped world. It is expected in the sense that you can get unexpected results when you do something you are not supposed to do like finding the predecessor of 0. But however, the predecessor of  $n$  plus 1 would be  $n$ . Then there is of course a 0 predicate which is just the natural number applied to true and you will be able to get both true and false answers from this.



(Refer Slide Time: 7:11)



Once you have this you can basically define all other functions and naturals using these so there is no problem about it. So essentially we have looked at data structuring. The most important data structuring facilities are you should be able to represent Booleans and numbers, real numbers are just ordered pairs of natural numbers, integers are just ordered pairs of a sign bit and a natural number so you know that a sign bit can be a Boolean or a natural number so integers, reals, floating points are all trivial.

Once you have represented numerals those things are trivial and of course we have looked at the basic data structuring facilities forming pairs, tuples and sequences. And essentially you have the power of the data structuring facilities in any programming language.

Let me tell you a story: once upon a time in a forest of lambda terms there was once a young combinatory called Y. So people ask why Y and they got back the reply because why why Y so people asked but why why why Y and they said well because why why why Y why Y and it went on and that is this young combinator looks like this. I have of course followed structured programming facilities and provided abstraction and so on and so forth so I have given a new name V to this lambda term and Y is  $\lambda X V V$  where V is this. So essentially if I remove this V and write out the full lambda term this combinator looks like this. So  $\lambda X \lambda Y X$  applied to Y applied to Y this whole thing applied to X.

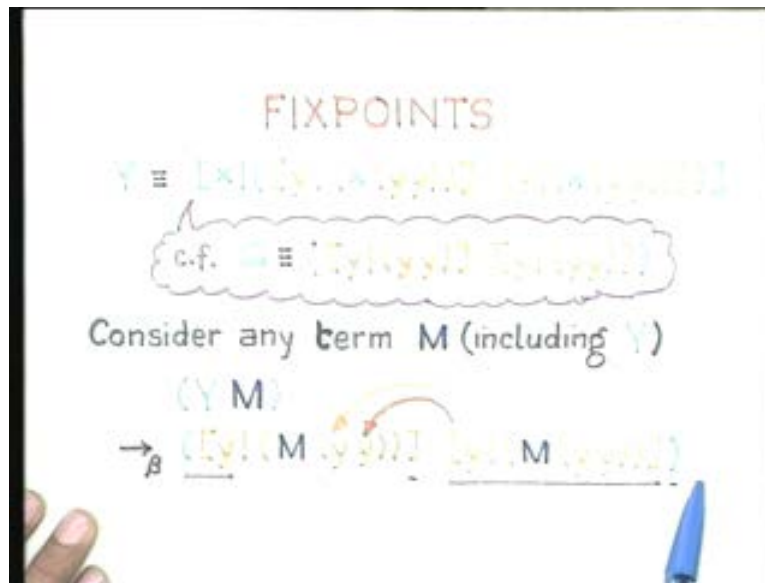
Now we have seen a similar combinatory, something that looked a bit similar which was the omega combinatory. So here is Y colorfully embellished so that you can distinguish all the terms. And the combinator omega that we had was somewhat similar to this except that none of these Xs were here and these abstractions over X were not there. And if you just throw your mind back to what we said about omega, omega keeps beta reducing to



itself or to an alpha version of itself. Omega is the ultimate in undefinedness of course it is an infinite computation which does not show any output it is really like an infinite loop.

So if you take this Y instead which is not quite that, if you perform this beta reduction you get something else you do not really get the effect of Y but you do get some form of infinite computation. So if you consider any term M and by saying any term M now you can include Y if you like the term M could be Y itself which is why why Y why Y Y why Y why Y and so on and so forth.

(Refer Slide Time: 12:46)

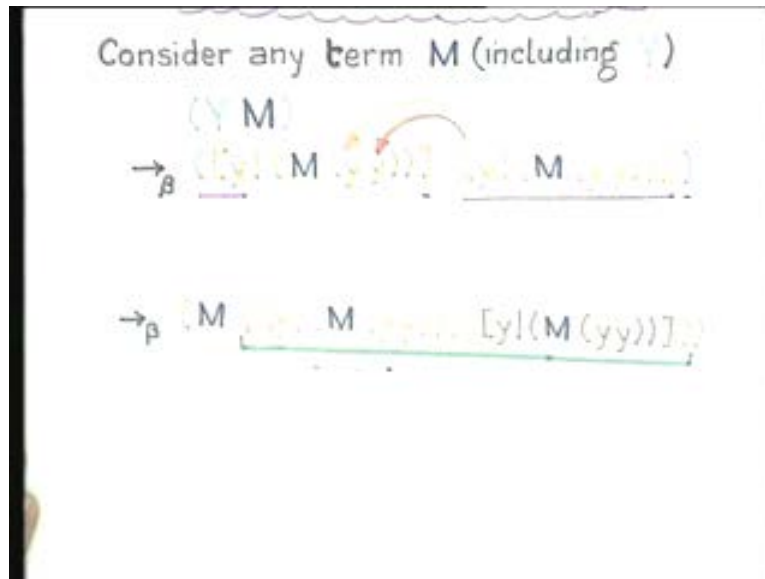


Now if I apply Y to M what happens, there will be a beta reduction so this left bracket comes here and then this M comes here and then there is a closed bracket the right parenthesis actually and then I can do this beta reduction, I do not do this application at all but what I do is I do the beta reduction over this X. then when I do this beta reduction over this X what it means is that this abstraction goes, this application remains so this blue left parenthesis is not this one but this one.

And instead of X I put M Y Y and then again  $\lambda y. \lambda x. x (y y) M Y Y$ . Now this is itself a beta reduction this is beta redex and I can apply beta reduction to it. And when I do that what I get is this. So this is the beta reduction so what it means is that each of these light [occ...13:15] colored wise will be replaced by this orange term. So when I do that replacement I get this M applied to this thing. The blue brackets are of course gone away due to the application so this **occo** bracket is this **occo** bracket and this bracket is this aqua bracket and these two **aqua** brackets are these two **aqua** brackets and each of these Ys has been replaced by this term which of course I have written in orange and red to distinguish that.



(Refer Slide Time: 13:54)



Now what is this term? This whole term (Refer Slide Time: 13:55) which has been underlined in green is just Y applied to M.

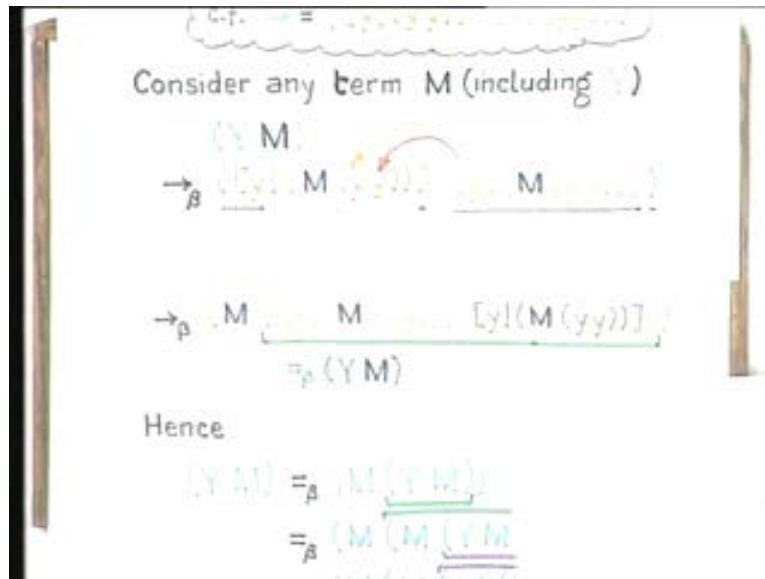
So what is the moral of the story?

You get that  $Y$  applied to  $M$  is beta equal to  $M$  applied to  $Y$  applied to  $M$  and that is beta equal to  $M$  applied to  $M$  applied to  $Y$  applied because I can taken this  $YM$  and do a beta reduction and get another copy of  $M$  applied to  $Y$  applied to  $M$  and so on and so forth. And of course instead of  $M$  I used  $Y$  then I get  $Y$  applied to  $Y$ ,  $Y$  applied to  $Y$  applied to  $Y$ ,  $Y$  applied to  $Y$  applied to  $Y$  applied to  $Y$  applied to  $Y$  applied to and so on and so forth.

The moral of the story is that this combinator  $Y$  has a peculiar property and that is that given any term  $M$  it automatically gives you a fixed point of  $M$ . If we think of these lambda terms as being functions then this equation is really equivalent to a some function which says, if I take this  $(Y\ M)$  as some  $X$  then what happens is that if  $x$  is equal to  $(Y\ M)$  then what I get is that  $x$  is equal to  $f(x)$  where  $f$  is  $M$  which by definition is a fixed point.



(Refer Slide Time: 14:15)



When is  $x$  a fixed point of a function  $f$ ?

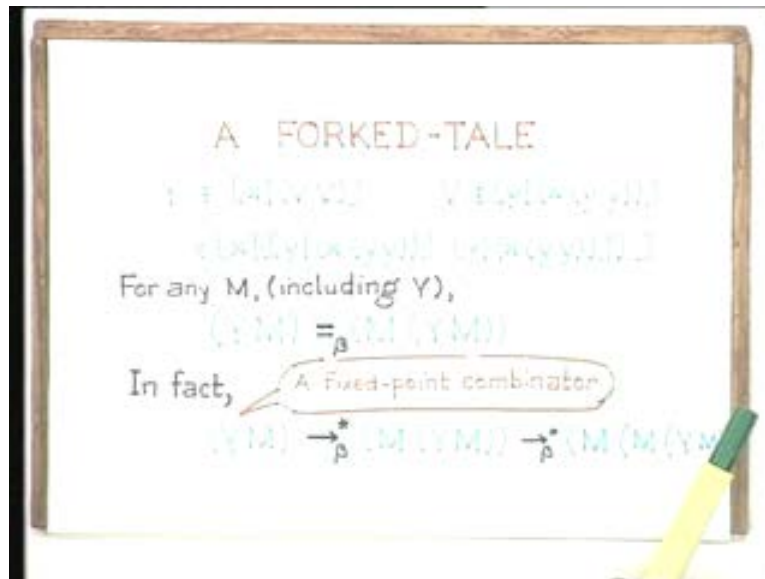
It is when  $x$  is equal to  $f(x)$ . So up to beta equality the  $Y$  when applied to any lambda term automatically gives me a fixed point of that lambda term. So what happens is that I can always find fixed points by just applying the combinator  $Y$  on the term. What I mean is I can get something that satisfies such an equation. And the property about fixed points of course going back to our normal mathematics is that if  $x$  is equal to  $f(x)$  then I can apply  $f$  on both sides and I get  $f(x)$  is equal to  $f(f(x))$ . But I already know that  $x$  is equal to  $f(x)$  so this equals this and I can go on adding infinite terms, in fact I can keep on applying  $f$  again and again and that is what we do and that is the property of a fixed point.

Here essentially the same effect has been obtained. This  $Y$  applied to  $M$  is  $X$ , this  $M$  is  $f$  then we get  $x$  is equal to  $f(x)$  is equal to  $f(f(x))$  is equal to equals  $f(f(f(x)))$  and so on and so forth. So what is the importance of fixed point operators? We have already seen fixed point operators somewhere.

Essentially this forked-tale actually is summarized in this. So  $Y$  is the fixed point combinator such that you can keep getting this sequence and add infinite term without any problem and we have seen other fixed points also.



(Refer Slide Time: 18:08)



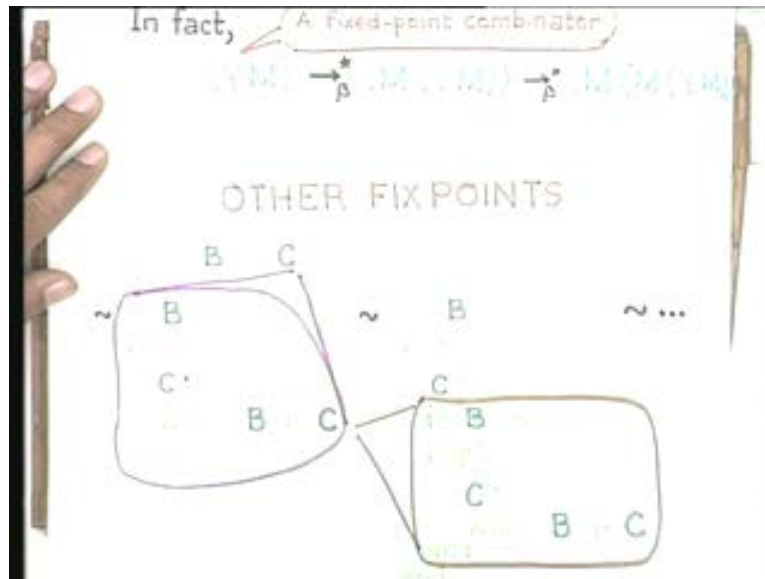
Thus let us look at some other fixed points. Instead of looking at mathematics we should actually look at programming. We have this standard while loop for example in our imperative programming language and that is semantically equivalent to this. In our operational semantics that is what we defined. This while loop is like some combinator Y applied to something which includes B and C and the effect of that is to unfold the while loop. The M in this case is that something which contains B and C and now I can unfold this “while” again and I can get an semantically equivalent construction like this and so on and so forth and I can keep unfolding that while loop again.

Each inside while loop can be expanded out into this and which is exactly what happens in this case. So if you look at this then that is really what you are doing. This context consisting of B and C is the M here, the while is just that Y applied to M and the effect is to expand out the inner most while by creating a new copy of the body. So just like by beta reduction you can go in this fashion what we are doing in our operational semantics, in our implementations and so on is that we are gradually unfolding the while loop one at a time for each execution of the program. You are unfolding the while loop in terms of an “if then”. This Y actually provides the required unfolding that you want how much ever you like.

In the case of “for loops” for example it is a finite number of iterations which means it is a finite number of unfoldings but in the case of while loops it is actually an indefinite number of unfoldings.



(Refer Slide Time: 201:15)



So you take this inner most body so this while loop is actually something of the form  $Y$  applied to  $M$  and it can be unfolded in each case to give you more applications of  $M$  onto itself. And if you look at other fixed point equations for example data and so on we actually have a similar situation. So take an alphabet  $A$  and take the set of all finite sequences or finite strings on this alphabet.

What is the set of finite strings on this alphabet?

It is the empty string and prefixing any string of this set by a letter of the alphabet which is like a one step unfolding of  $A$  to the power star. So this epsilon union  $A$  prefixed is actually like your  $M$  and the star is like  $Y$  applied to  $A$  and the result of that is to provide unfoldings of this form.

Now this A star in turn can be unfolded into this fashion and so on and so forth. You take in fact anything which has a fixed point this is what is going to be happen. if you take the collection of all finite and infinite strings on this set A it has a basis which consists all the finite strings and then prefixing of all strings in the same set A infinity and this is like a finite unfolding, you can go on in this fashion forever. You can think of this as a collection of equations which are all satisfied by any function which has a fixed point.

How do you get that fixed point?

You get it by applying the Y combinator to any function. So the Y combinator is actually quite interesting. Let us go back to this while loop.



(Refer Slide Time: 23:12)

The image shows a whiteboard with the title "FIXPOINTS IN DATA" written in red. Below the title, the following equations are written in black ink:

$$A^* = \{\epsilon\} \cup AA^* = \{\epsilon\} \cup A(\{\epsilon\} \cup AA^*)$$

A blue arrow points from the  $\{\epsilon\} \cup AA^*$  term in the first equation to the next line, which is:

$$= \dots \text{ ad infinitum}$$

The set  $\{ax \mid a \in A, x \in A^*\}$  is circled in green. Below this, the next equation is:

$$A^\infty = A^* \cup AA^\infty = A^* \cup A(A^* \cup AA^\infty)$$

Finally, the last line is:

$$= \dots \text{ ad nauseum}$$

One of the things that I brushed aside when I was defining the while loop semantics was that I said it is not structurally inductive. Actually what we can do is we can define a structurally inductive semantics and that is very simple. What do I do?

I have something consisting of the set of states, I have some set capital sigma which is the set of all possible states and I can take any mathematical domain including the set of states and apply a lambda calculus on it in the sense that I can define an applied lambda calculus of the algebra of states.

I can take any mathematical domain and I can do lambda abstractions and so on. The lambda calculus is sufficiently independent to allow for such applications. We have already seen how we get an applied lambda calculus on natural number like Peano Arithmetic and so on. Do the same thing for states then what it means is that in particular you can apply the Y combinator on any function on states and you can get a fixed point and that is what it guarantees.



(Refer Slide Time: 25:46)

$$\begin{array}{c} \text{INDUCTIVE} \\ \hline \frac{\langle B, \sigma \rangle \rightarrow_B^* \langle f, \sigma \rangle}{w_{B,C}(\sigma) \rightarrow \sigma} \\ \\ \frac{\begin{array}{l} \langle B, \sigma \rangle \rightarrow_B^* \langle t, \sigma \rangle \\ \langle C, \sigma \rangle \rightarrow_C \sigma' \\ w_{B,C}(\sigma') \rightarrow \sigma'' \end{array}}{w_{B,C}(\sigma) \rightarrow \sigma''} \end{array}$$

You take any lambda term and you apply the Y combinator on it you get a fixed point and see how this while loop works means that you just have to find a fixed point on a function on states. That function on states is defined by the Boolean condition and the body of the while loop. So here is the semantics. There is some function W which is defined as follows: If the Boolean for example starting in some state sigma evaluates to false then this WBC on sigma evaluates to sigma. And if B sigma goes to true and C sigma goes to some state sigma prime and WBC applied to this sigma prime goes to some sigma double prime then WBC on sigma goes to sigma double prime. This WBC we know exists now because you can always define an applied lambda calculus on the functions of states, you can always take the state space and on top of it have the applied lambda calculus, the pure lambda calculus sitting on the algebra of the function of states. It is some function which satisfies these properties.



(Refer Slide Time: 27:18)

$$\begin{array}{c}
 \text{INDUCTIVE SE} \\
 \hline
 \frac{\langle B, \sigma \rangle \rightarrow_B^* \langle f, \sigma \rangle}{w_{B,C}(\sigma) \rightarrow \sigma} \\
 \\
 \frac{\begin{array}{l} \langle B, \sigma \rangle \rightarrow_B^* \langle t, \sigma \rangle \quad \text{---} \\ \langle C, \sigma \rangle \rightarrow_C \sigma' \\ w_{B,C}(\sigma') \rightarrow \sigma'' \end{array}}{w_{B,C}(\sigma) \rightarrow \sigma''}
 \end{array}$$

Note that these arrow marks for WBC do not have a label. If I have to give them a label that label will be like the label that I defined for the applied lambda calculus on Peano Arithmetic. Each reduction is either a beta reduction or a Peano reduction. In this case this arrow would be either a beta reduction or some reduction defined on the set of states. So I do not have subscripts on all arrows which deal with this WBC because this WBC is not green, please note this point.

WBC is the highest level of abstraction you can go to and it is a function on states. it takes one state to another state and it is a function on states which depends somehow on B and C. it is a function on states constructed from behavior of B and C. the only way you can get a semantics of the while loop is that it should somehow depend on the Boolean and the body. By putting the pure lambda calculus on top of whatever may be the calculus or whatever may be the reduction mechanism for states is that we still can get a Y combinatory.

We still have the Y combinator of the lambda calculus and you can see that there are no constraints at all on that term M. That term M could be either pure or applied it does not matter. The point is that whether this term is pure or applied we are not looking inside this term M. So we are not using the structure of M or the properties of M in anyway. These beta reductions, the Y just allows that whatever may be this it might be just one tone of garbage too, I mean, what it means is that it allows for replication of that term.

The behavior of the Y combinator is completely independent of this term M. this term M can be a pure lambda term, it can be an applied lambda term, it can be in any mathematical domain and I am not looking into it, I am not manipulating M in any other way, all I am doing is this Y just creates new copies for the application of M to whatever is the rest. So M is actually just a data object and therefore it is not a function and what I



will get is an applied lambda term which I cannot interpret. But if this M is actually a function in my domain in whatever domain of interest that I have then what I get is an application of this function.

I am assuming a unary function so it means an application of this function repeatedly satisfying this fixed point equation. So this WBC is really constructed through the Y combinator and what does this WBC do is it just gives you these beta reductions straight away except that you might also mix beta reductions with the reductions in this algebra of states. So the only constraints we are imposing on this function WBC is that if B in this state sigma as false then WBC should do nothing and just return that state. If B in this state is true then what should WBC do? It should iterate the body C once and then create whatever is the effect of WBC on the new state. This is iterating the body C once, and on that new state find out whatever WBC can do and that is what WBC can do original state sigma that you started of with.

(Refer Slide Time: 31:26)

INDUCTIVE SEMANTICS

$$\begin{array}{c}
 \frac{\langle B, \sigma \rangle \rightarrow_B^* \langle f, \sigma \rangle}{w_{B,C}(\sigma) \rightarrow \sigma} \\
 \\
 \begin{array}{c}
 \langle B, \sigma \rangle \rightarrow_B^* \langle t, \sigma \rangle \\
 \langle C, \sigma \rangle \rightarrow_C \sigma' \\
 w_{B,C}(\sigma') \rightarrow \sigma''
 \end{array}
 \quad
 \frac{w_{B,C}(\sigma) \rightarrow \sigma' \quad \langle B \wedge C, \sigma \rangle \rightarrow_C \sigma'}{w_{B,C}(\sigma) \rightarrow \sigma''}
 \end{array}$$

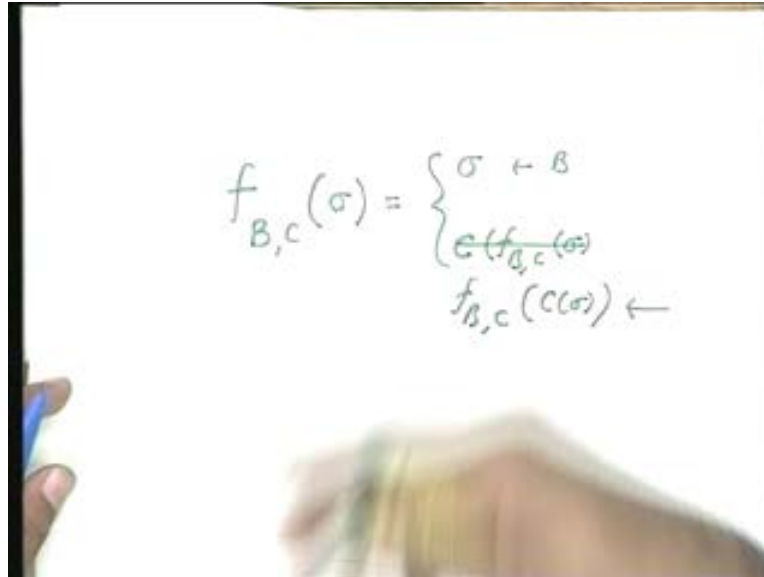
Remember that long time back when I spoke about transition systems I was saying that almost anything on earth can be represented as a transition system including functions. After all why should not we have functions also represented? Instead of representing functions as equations I will represent them through this arrow mark, as a transition. So the successor function applied on N goes to a transition which is N plus 1. So this WBC is actually a function and what is this function?

It is actually the function which somehow takes B and C creates a new function so it is really a function of this form.



It is a function  $f_{B,C}$  on sigma so this function is sigma if B is false and whatever is the effect of C applied to  $f_{B,C}$  when sigma is true or rather it is  $f_{B,C}$  applied to C applied to sigma whatever the function C might be. From the functions of B and C I somehow combined and applied a Y combinator onto them and I get this function WBC.

(Refer Slide Time: 33:18)



$$f_{B,C}(\sigma) = \begin{cases} \sigma \leftarrow B \\ e(f_{B,C}(\sigma)) \\ f_{B,C}(C(\sigma)) \leftarrow \end{cases}$$

In order to be consistent with our transition system semantics I have defined this WBC also through reductions but otherwise it is a function. Now we have a purely structurally inductive definition for the while loop which is not really operational. Here in the second row these two sigma primes are the same and these two sigma double primes are also the same. If B in the original state sigma evaluates to true in our original while language there were no side effects.

But if supposing you had side effects and so on what it means is that I would just transform this sigma to some tau and work with that tau and that is the only change that will happen in the rule, it is not going to affect the semantics of the while at all in any way. So, if B evaluates to true in this state sigma (Refer Slide Time: 34:41) and one execution of the body of the C actually gives me sigma prime this actually should be possibly a many step evaluation. And if WBC is the function on this new state sigma prime, the new sigma prime is obtained after one execution of the body of the while loop, if it gives me a final state sigma double prime then WBC on the original state sigma also gives me sigma double prime.

On the other hand, if the Boolean in the original state evaluated to false then WBC just leaves the state unchanged so this is really like the “if then”. So the while loop is really the Y combinator applied to an “if then” which is really how our operational semantics works. So there is this function made up of the function Booleans B and the body C which is really an “if then” function with some sort of a parameters **whole** for it and the



while loop is just the Y combinator applied to this. Therefore it is possible to define a purely structurally inductive semantics for the “while loop” something that we had left really undefined.

We had given it an operational flavor in terms of implementations but if you want a purely structurally inductive definition it is possible, not that it is very easy to understand I can see that it is quite hard to understand but anyway if you ever do a course on semantics of programming languages you will have to deal with fixed points a large numbers of them. And it so happens that in the lambda calculus this Y was a combinator defined originally by church and it does not satisfy one very important property.

Hence for any term M Y applied to M is beta equal to this M applied to Y applied to M. but Y applied to M does not beta reduce to M applied to Y applied to M. Why it does not beta reduce? Why does not Y apply to M, why does not it beta reduce to?

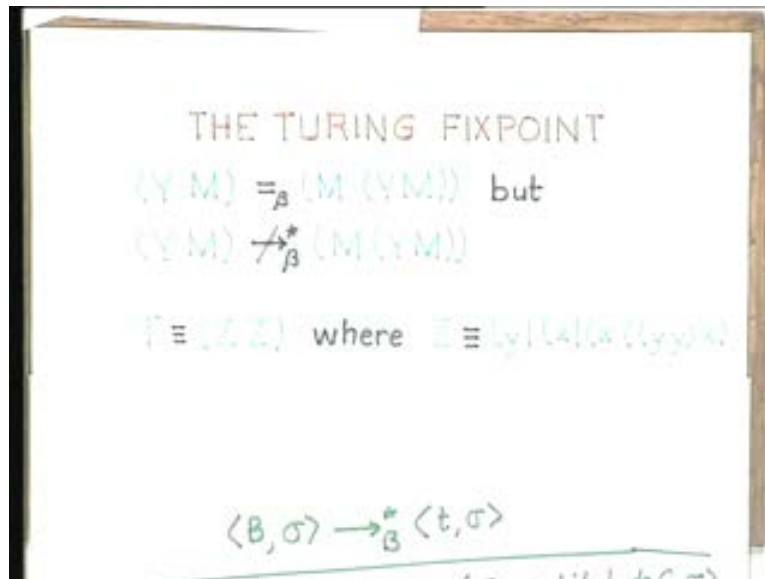
So what have I done here, (Refer Slide Time: 38:49) I have taken Y applied to M and with one beta reduction I got this step, with another beta reduction I got this step but I inferred the beta equality from the fact that this is equal to Y applied to M, it is going backward. But Y applied to M does not itself reduce to M applied to Y applied to M, it is just that beta equality since it is defined in a fashion where you can either go forwards over reductions or backwards over reductions it does not matter then I would consider the two terms equal. It is because beta equality is defined in that fashion that Y applied to M is actually equal to M applied to Y applied to M. But Y applied to M does not beta reduce to M applied to Y applied to M. And in fact a nice property that we would like to have is really that, is it possible to find a combinator which will satisfy the property that whenever it is applied to a term if beta reduces to this expansion. This is what really happens in our implementations.

If you want to accurately model implementations what you are doing is you are you are actually going through this beta reduction you are not going through an equality. You are using the equality may be to reason but the actual execution of a “while loop” program is that you reduce that “while loop” program, if you go back to our original operational semantics how did we define the original operational semantics in the while loop?

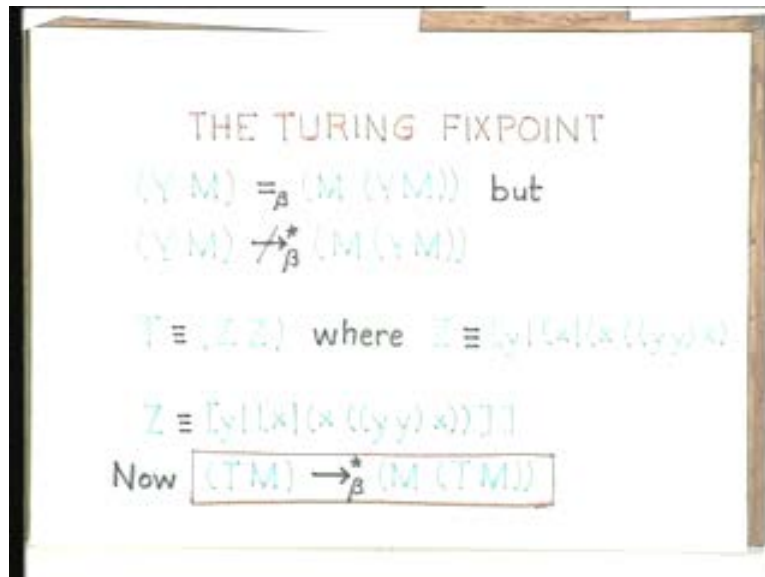
Especially let us forget about the case when B is false. When B is true we wrote it as if B, sigma and when this goes to true sigma then we said that the while B do C sigma goes to C; while B do C sigma. So what we have defined here is the reduction, this transition is like a reduction which actually does the unfolding and this is an accurate rendering of how your loop is actually implemented by the code generation procedure in your compiler. This is actually the way it is implemented so what we would like to know is whether it is possible to have a fixed point combinator which actually does this unfolding which for example Church's Y combinator does not and it turns out that Turing actually defined one which does, yeah, and you can actually check it out at home.



(Refer Slide Time: 42:07)



(Refer Slide Time: 42:17)



Therefore the Turing fixed point combinator  $T$  is defined in this fashion  $\lambda Y, \lambda X, X$  applied to  $Y$  and  $Y$  applied to  $X$  and you can do beta reduction to see that it actually directly reduces in this fashion. It actually does the unfolding that you require for your implementations.

The whole point is that you have to be careful in order to check such a thing. You have to be careful that you are not using the fact that you had named something before as something, you are not using that but that is what we did in the case of Church's

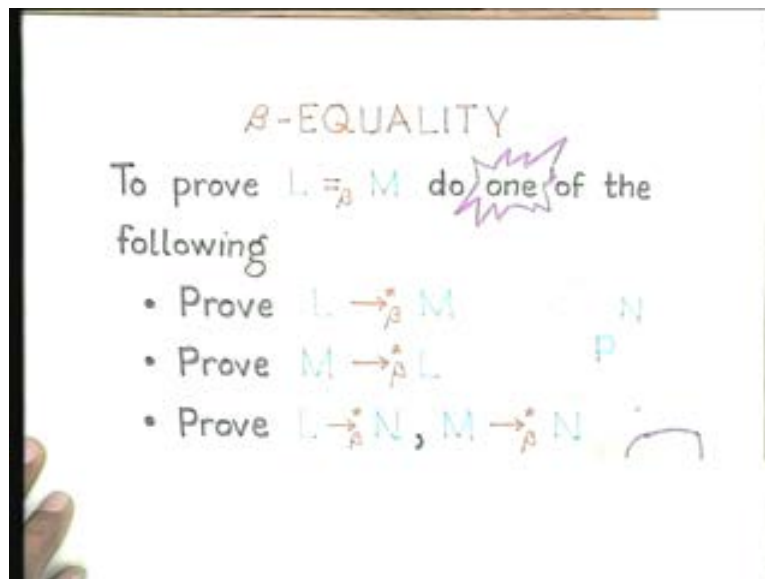


combinators. It does not directly reduce, you have to use the rules for many step beta reduction very rigorously.

Now let us look at the beta equality. Some of the things we have done over the past few lectures is that considering the way beta equality is been defined, two terms are beta equal. In order prove the two terms are beta equal you can do one of the following. one thing is directly reduce the left hand side term to the right hand side term,  $L$  to  $M$  and other possibility is you can directly reduce the right hand side to the left hand side and then they both would be beta equal. The third possibility is that you reduce each of them to some common term.

Let us look at this beta equality from the point of view of our new beta equality which includes alpha conversion. In all these many step beta reductions you might also have some alpha conversions in between. You can intersperse them any where you like. So we will assume that we will not explicitly mention alpha conversion but we will say that it is syntactically equal. So strictly speaking what I would say is in order to prove  $L$  is beta equal to  $M$  by the third method it means that you reduce them both independently to some two terms  $N$  and  $N'$  which are mutually alpha convertible if you like.

(Refer Slide Time: 44:44)



But I will consider the alpha equivalence to be the same as syntactic identity. So essentially what it boils down to is that you can reduce them both to some common term and then when you reduce them both to some common term you can claim that  $L$  is beta equal to  $M$ . There is actually a fourth possibility and that comes from the fact that the operational semantics of the lambda calculus is non-deterministic.

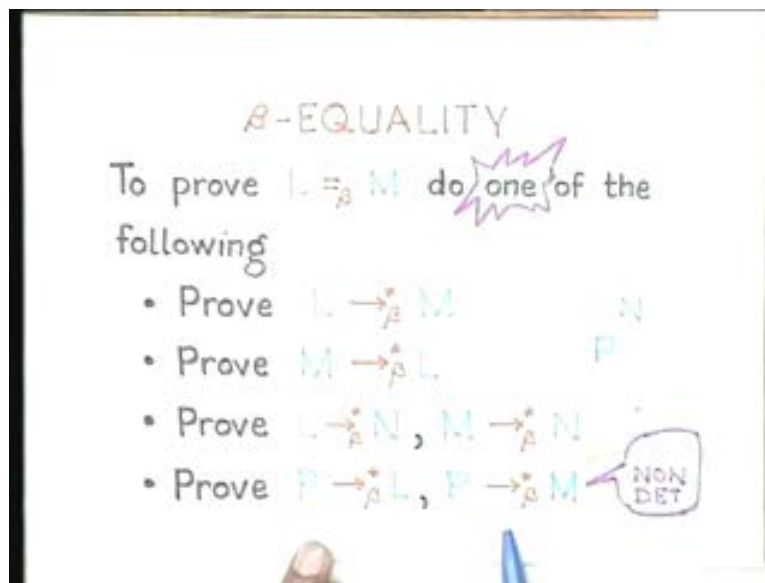
What I can do is I can find a term  $P$  which reduces by one way to  $L$  and by another way to  $M$ . Since the operational semantics of the lambda calculus is non-deterministic it



actually allows for this possibility. This is something that we do not very often use because we are all too concerned with proving things by reductions. But there is also a method of proving things by abstractions so that you find a common abstraction from which both of these will reduce to these terms may be in two different ways.

**In the non-deterministic operational semantics is there any guarantee?** What I am saying is, that is something that we have to prove. How do you know it will always work? You are saying that if it is not true then we will not be able to use it which is true. But there is nothing in the development that we have said so far which actually guarantees that you will find this N. What guarantee you have that you will be actually able to find a common M. Sometimes there might not be a common M always so the fourth one does not necessarily reduce to the third one.

(Refer Slide Time: 48:00)



What we are saying here is I look at the structure of L and M and I make a guess about P which is a lambda abstraction within application may be such that through a sequence of beta reductions I can get L through another sequence of beta reductions starting from P because P might have more than one beta redex. Therefore, if L and M are distinct then P is something which either has more than one beta redex initially or somewhere along the line through a common beta reduction it expands to a term which has more than one beta redex and by applying them differently by going through the beta reductions in two different ways I might be able to get L and M, it is a feasibility, it is a feasible solution but there are no guarantees that all of them will reduce.

So all I am saying is it is just like, given a theorem to be proved there are many directions you can follow but some of them might lead you to the conclusion and the others may not so these two are not equivalent and there is no guarantee that there are going to be

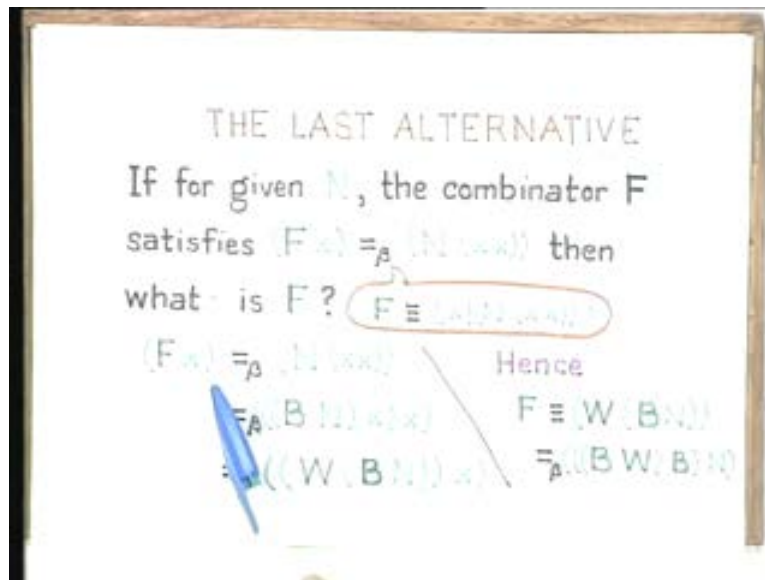


equivalent. But this last kind of alternative is something that we have not actually used in our manipulation so far but it is certainly a feasible method.

For example, I can take this combinator so I can ask if for a given  $N$  the combinator  $F$  satisfies this beta equality then what is  $F$ ?

Well, there is one way of looking at  $F$ , I will just say that  $F$  is this combinator (Refer Slide Time: 49:56) one of the most natural things. But then I decide I do not want this horrible thing and I will look for something else, I will look for a property therefore I will look for this.

(Refer Slide Time: 50:07)



So if  $f$  applied to  $x$  is beta equal to this then this is a form of self application. So, if I look at this structure it is like  $N$  is applied to  $x$  is applied to  $x$  which is like a composition of two functions so I take the composition operation  $B$  and I redistribute the parenthesis in this fashion. Now I have a guarantee from the structure of  $B$  and by lambda reduction that I can actually get this by a many step reduction from this.

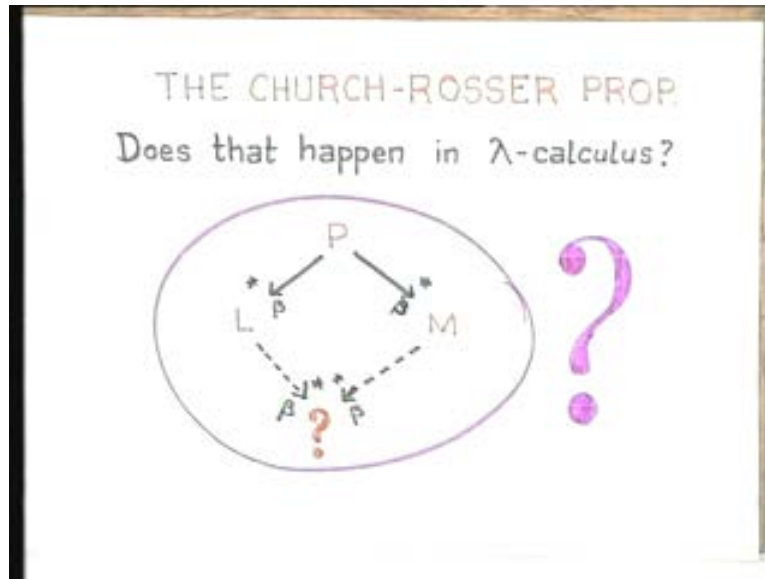
Now there is one important combinator which I did not mention at that time which is the  $W$  combinator which is just a diagonalization,  $W$  is just  $\lambda x. \lambda y. (X \text{ applied to } Y)Y$ . Then if I look at this  $W$  then I can go backwards. I know that from  $W$  this way I can get this by a beta reduction. So now I have got two structures which are of the same form a single  $x$  at the last and  $f$  applied something so this is something applied to  $X$  so I can claim that  $f$  is  $W$  applied to  $B$  applied to  $N$ . But then this is again like a function composition so I can look upon this as  $BW$  applied to  $B$  applied to  $N$ . So this is going backwards by the alternative four.

And the important question that he has raised is really the last important thing. Is it really true that given a lambda term  $P$  if it reduces in two different ways because of non-



deterministic term to  $L$  and  $M$  is it true that they will both eventually reduce to some common term. We are too used to seeing such things being taken for granted in Mathematics.

(Refer Slide Time: 52:52)



There are as yet no guarantees that this property would actually hold in the lambda calculus. This is a typical school Mathematics problem, you can have two different solutions, it shows non-deterministic reductions, it shows different applications of different formulae, one uses the fact that a square minus  $b$  square is equal to  $a$  plus  $b$  into a minus  $b$ . The other uses the fact that  $(a$  plus  $b)$  square is equal to  $a$  square plus  $b$  square plus  $2ab$  and you get the same answer. We are used to getting the same answer that we take it for granted. But how do you know it is actually going to happen and that is an important property. And unless this is guaranteed this does not really model programming in any way and we have to somehow guarantee this.