

**Principles of Programming Languages**  
**Dr. S. Arun Kumar**  
**Department of Computer Science & Engineering**  
**Indian Institute of Technology, Delhi**  
**Lecture - 38**  
**Meanings**

Let us look at abstracts namely functional and procedural abstracts and today I will try to define the meanings of abstracts. The age old fundamental question still remains for which we will only partly answer today. What's in a name? We have an abstract.

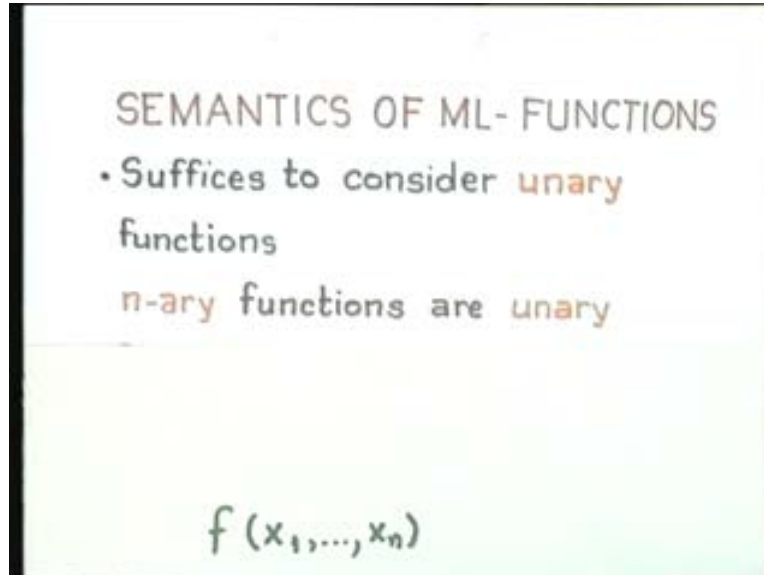
We still have not answered the question what really is an abstract. So today we will answer it. I know a lot of people who read Agatha Christie books who after the initial chapter when the murder is committed you go to the last chapter where the murder is solved and then they start reading the rest of the book. So let's do the same thing. So we have done the initial chapter of this chilling suspense and now there is this chilling suspense.

Thus the next question is who did it?

In our case it's rather who had done it, in our case we say what's in an abstract, what is an abstract?

And amazingly the answer is that it is an applied lambda abstraction, applied or pure depends on the rest of the application and we will define the notion of a closure and we will try to define the meaning of abstracts. Therefore any abstract especially parameterized ones are really going to be just lambda abstractions. So the distinction between an abstract and an abstraction then becomes negligible. In functional settings these things are always easier to define so we will look at the semantics of ML like functions.

(Refer Slide Time: 4:00)

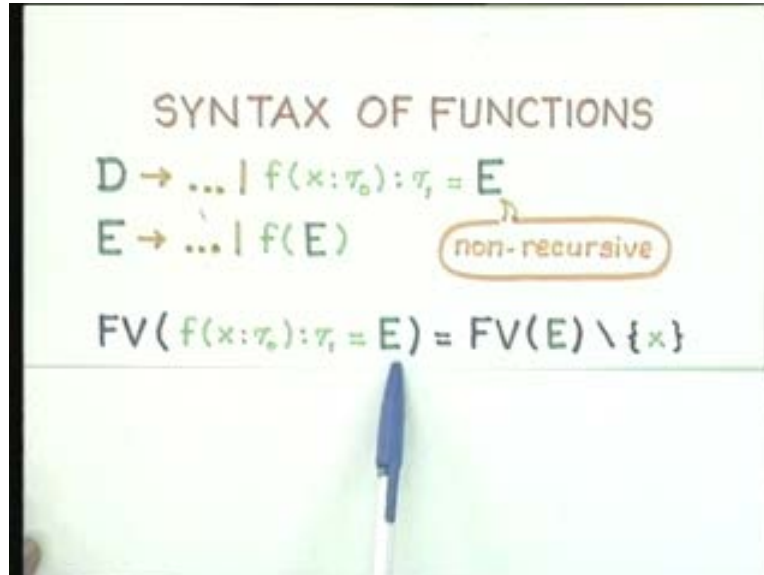


So let us look at functions. Whatever we did we only considered mostly values and I will make the assumption that it suffices to consider unary functions and there is a view point in which every function is a unary function not obtained by [c.....3:18] it is just that any n-ary function can be regarded as a unary function applied to an n tuple construction. In fact in that sense the conventional mathematical notation for a function is also really that. This is really  $f$  as a unary function applied to an n tuple so we do not lose too much by considering all functions to be essentially unary functions so there is an important aspect of a n tuple construction before you actually apply the function but even otherwise whether carried or not carried we can think of all functions as being unary.

Then what happens is with this it also makes it more convenient and less messy to talk about functions, their parameters and so on. So let us look at the syntax of functions. Now that we have done something about type checking what I will do is I will define the syntax. So essentially there are just two constructs we have to worry about. One is function declarations so  $f$  is a function symbol with a parameter  $x$  of type  $\text{tau}_0$  and  $E$  is an expression of type  $\text{tau}_1$ . And if you remove this type information then essentially it is like a ML declaration of a function.

And of course even in ML if the type inferencing has to succeed always you might have to put at least some type information somewhere in order to make sure that these operations inside  $E$  are meaningful. So we will we will assume that we have we have a fully typed information. Then of course the function call is just  $f$  applied to some expression  $E$ . This is a sort of ML syntax. So we will mainly consider non recursive functions. The extension to recursive functions is quite simple if you clean up the syntax a bit.

(Refer Slide Time: 7:14)

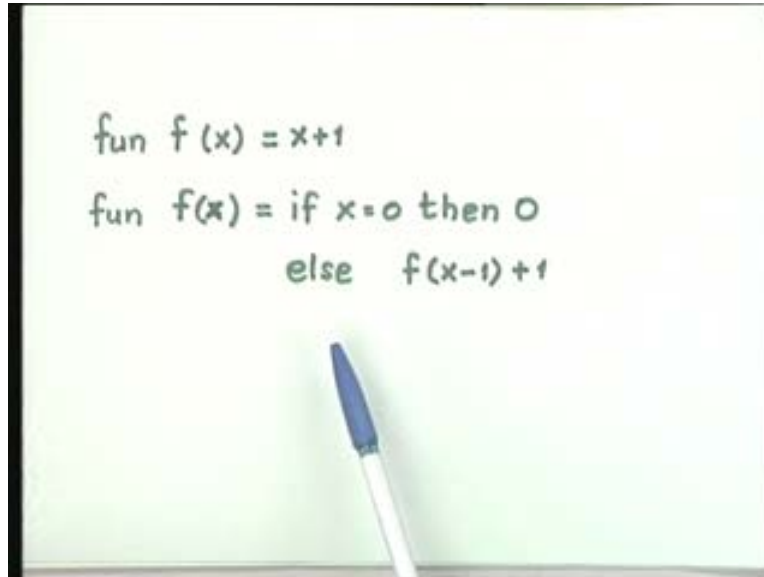


Then as with all declarations and expressions we have to look at what are the free variables of a declaration. So the free variables of a declaration are just all the free variables of  $E$  except of course  $x$  which is a parameter of  $f$ . Thus if it is non recursive then that's really what it is, if it is recursive what it means is that you will have to use some form of disambiguation to make sure that  $f$  is not included in the free variables of  $E$ . Remember that functions are also values and there is a great unity operating all that. So, if you have some way of recognizing that it is a recursive definition then what it means is that you will have to remove  $F$  also from the free variables of  $E$ .

In the case of ML what happens is the question of whether if  $f$  occurs in  $E$  the question of whether it is recursive or that  $f$  actually refers to some previous declaration of  $f$  is resolved in ML by a very simple mechanism. It takes the innermost reference and decides on that. So it looks through this and if there is an  $f$  here then it assumes that it must be a recursive definition. Hence, ML has a very simplistic view of that in which case in actual ML programs this distinction between non-recursive and recursive is not very syntactic it is rather implied being direct. But many languages like Camel for example which is really a variant of ML explicitly requires that if you intend a function to be recursive then.

You should put a reserved word called "rec" in front of the definition of the function. Thus for example if you have a definition like this;  $f(x)$  is equal to if  $x$  is equal to 0 then 0 else  $f(x - 1)$  plus 1 so if you have a definition like this in the case of ML it automatically takes this to be a recursive function. On the other hand, this might be a part of ML session in which you may have previously declared  $f$  to be of type explicit. Now what ML does is it just basically disregards this and all occurrences of  $f$  here refer to this most recent syntactic occurrence as part of the interpretation.

(Refer Slide Time: 10:00)

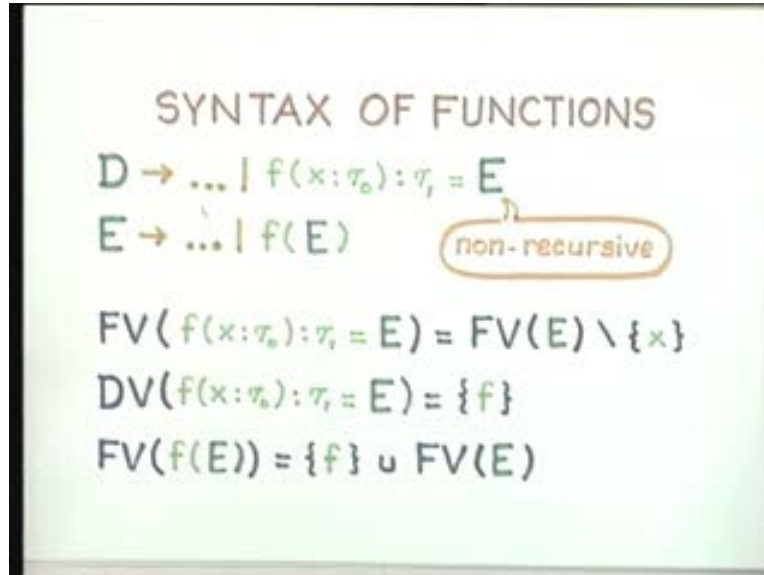


However, Camel is very much like ML has a very similar syntax and is based on the same kind of system, type inferencing and so on. It actually specifies that if you do not include a reserved word `rec` here so this “`rec`” is a reserved word which is supposed to indicate that a function is recursive. Then if you actually include this reserved word “`rec`” here only then Camel will take this `f` to be a recursive call to the same function. If this “`rec`” were not there then what would happen is Camel would be looking in the environment for a previous declaration of `f` and would use that as the meaning of this `f`.

In some sense in ML things are very implicit, even though the designers of ML were among the first to point out that there is a ambiguity in programming languages in which most programming languages do not use this reserved word called “`rec`” or some kind of a syntactic tile which makes it clear whether the function or a procedure is recursive. In spite of that they actually have this implicit binding mechanism which binds this `f` to this whereas in Camel it actually has to be made explicit by means of this reserved word `rec`.

In that sense Camel is somewhat more explicit and perhaps better because we do not really know what kinds of ambiguities the lack of this recursive keyword can create. We already know how it can create problems in Pascal functions for example. For the present let us assume that these are non-recursive things.

(Refer Slide Time: 12:23)



The only difference in our semantics if it becomes recursive is that you have to have a  $y$  combinator or some form of recursion combinator just like we found for “while loops” otherwise the difference between recursive and non-recursive is really a question of identifying what are the free variables and what are the defined variables.

If  $E$  is non recursive then can  $f$  be a free variable?

If  $f$  occurs in  $E$  it is yes.

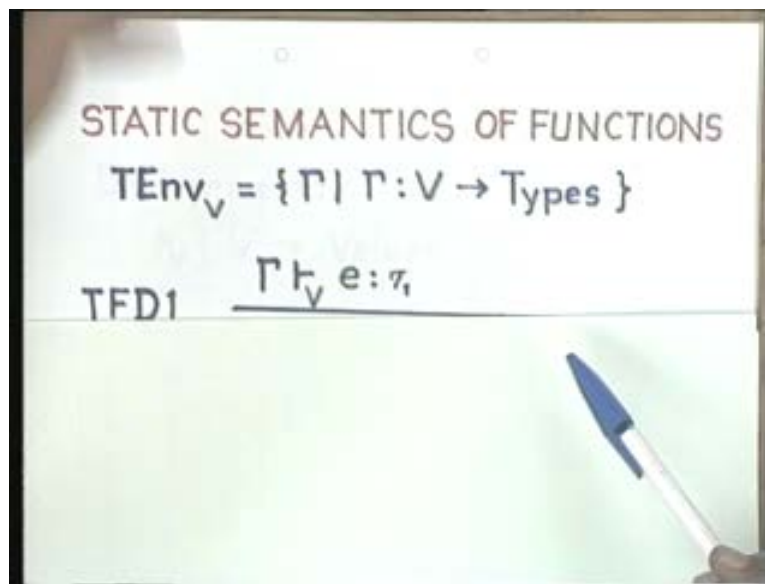
Let us be explicit, let us take Camel like syntax where if you intend it to be recursive then you give this keyword “rec” otherwise you do not give the keyword “rec”. So if you do not give this keyword “rec” here then what it means is that the elaboration of this expression means that your environment the environment in which this expression is going to be evaluated should already have some definition for  $f$ . If it does not have a definition for  $f$  what a Camel interpreter will do is it will say that  $f$  is an undefined variable. This is the non-recursive case.

In the recursive case this reserved word “rec” will actually help it to identify that therefore within the scope of this definition all occurrences of  $f$  are the same as this occurrence of  $f$  and since  $f$  is a defined variable here it is not a free variable. All this is not explicit in ML it is implicit. It is not necessary to have the same type of  $x$  in both the functions. But if we do not give this recursive “rec” then this  $f$  of  $x$  is going to be bound to the first function and if the type of  $x$  here and the  $x$  here are different then you are going to be at a type error. Therefore, if you do not intend it to be recursive and you actually intend this  $f$  to refer to something that is already available in the environment then the type checking problem has to be satisfactorily addressed.

Hence essentially all the variables in  $E$  except  $x$  are free. Thus the way I am looking at it now is that the free variables of  $\text{rec } f \ x \ \text{of type } \tau_0$  the whole function of type  $\tau_1$  is equal

to  $E$  is equal to all the free variables of  $E$  minus  $x$  and  $f$ . And if  $f$  were not recursive then any occurrence of  $f$  in  $E$  refers to some previous binding so that  $f$  is a free variable of  $E$ , it is as simple as that. Therefore what it means is that if you did not have the keyword the reserved word “rec” when  $f$  could be both a free variable and a defined variable of this definition so the defined variable of this definition is just the symbol  $f$ . This is as far as definitions are concerned. And in the case of a function call the free variables are called to  $f$  with an actual parameter which is an expression  $E$  which is just  $f$  union the free variables of  $E$ .

(Refer Slide Time: 19:36)



Firstly we have to look at the static semantics or functions of these definitions. As usual a type environment over a set of variables or a set of identifiers is just some identifier to type binding. Previously the types we had were just base types like integer and bool and so on and so forth. But of course after having spent so much of time on the typed lambda calculus it is clear that now our types actually expand out. In addition to the base types you have all the type constructions that are possible either in the simply typed lambda calculus depending upon what your language is or the polymorphic lambda calculus depending upon what your language is. Essentially it includes something more than just the base type. It includes all kinds of functions that you can create on base types.

And if you are considering a polymorphic case it also includes all type variables and type constructors including the “for all quantifier” on type variables. So whatever can be defined from the base types using the typed language of the polymorphic lambda calculus is included in this. And of course since our functional language is an applied language is an applied lambda calculus essentially all the types of the underlined application and so on and the higher types created by the lambda abstractions on those applications are all included in this type. That is how you get higher order functions from a simple application.

Let us just go through the type definitions. If  $e$  is of type  $\tau_1$  then this definition of this function  $f$  with  $x$  as a parameter of type  $\tau_0$  the type checking rule is just this, you should make sure that the body of the definition is of the same type  $\tau_1$  as given in the declaration. And then this definition creates a little type environment which associates with  $f$  the type  $\tau_0 \rightarrow \tau_1$ . Essentially the function takes an argument of type  $\tau_0$  and gives you a result of type  $\tau_1$  whatever  $\tau_0$  and  $\tau_1$  might be, they might be polymorphic types, they might be monotypes, they might just be base types or whatever that is representable.

(Refer Slide Time: 19:36)

STATIC SEMANTICS OF FUNCTIONS

$$TEnv_V = \{ \Gamma \mid \Gamma : V \rightarrow \text{Types} \}$$

TFD1 
$$\frac{\Gamma \vdash_V e : \tau_1}{\Gamma \vdash_V f(x : \tau_0) : \tau_1 = e : \{f : \tau_0 \rightarrow \tau_1\}}$$

TFC 
$$\frac{\Gamma \vdash_V e : \tau_0}{\Gamma \vdash_V f(e) : \tau_1}, \Gamma(f) = \tau_0 \rightarrow \tau_1$$

And then in the case of a function call of course the type checking is that, if your static type environment gives you the type of  $f$  as being something of the form type  $\tau_0$  arrow  $\tau_1$  then this function call type checks only provided the argument to the function call is of type  $\tau_0$ . So, if the argument to the function call is of type  $\tau_0$  then this function call is of type  $\tau_1$  and it type checks. So, essentially that's the type checking for function calls.

You just have to evaluate the type of argument. remember that all this is done at translation time, it has got nothing to do with executions, it is all done at translation time so essentially you look at the expression from the static environment that you have already created when you are looking at this expression you evaluate the type of this expression and if this expression has a type  $\tau_0$  and your environment assuming a declaration before used strategy your environment already has created for  $f$  the type binding  $\tau_0$  arrow  $\tau_1$  then this function call type checks but otherwise it does not.

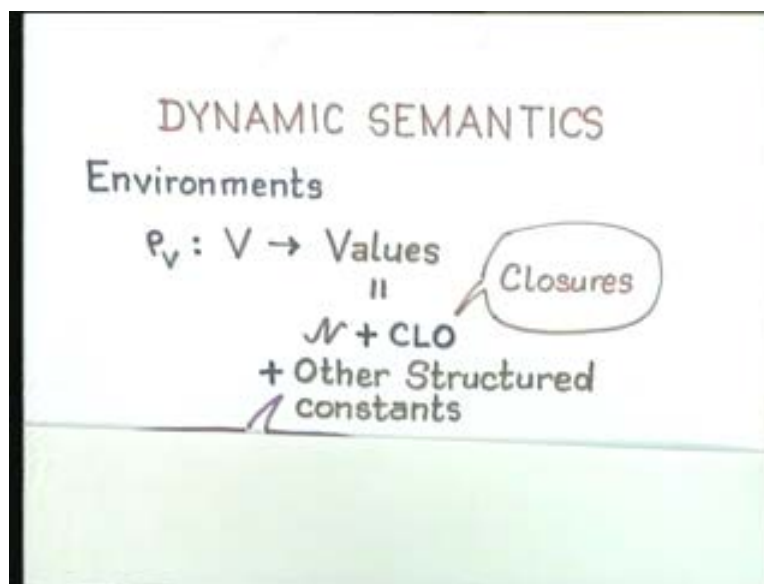
Why have I written that as a side condition?

Side conditions are usually of this form that in the main premises and the hypothesis we only use what is directly relevant to the syntax of that phrase. Side conditions are all the

information that could somehow be gathered or have been gathered earlier through some definitions or some declarations or through applications of other rules.

Given the type environment  $\gamma$  as an assumption this is a piece of information that is inside  $\gamma$  and is not part of the current syntactic phrase of our rules. So the current syntactic phrase is just  $f$  applied to  $e$  which has one sub expression  $e$  so the premise has to be in terms of just that  $e$ . So given this assumption  $\gamma$  if you can prove this is the provability symbol that  $e$  has type  $\tau_{e_0}$ ,  $e$  green, remember my coding that green is a part of the programming language, blue is something else some other pieces of information, derivations inferred information and so on and so forth.

(Refer Slide Time: 27:28)



Given this type context  $\gamma$  if you have inferred that this expression  $e$  is of type  $\tau_{e_0}$  then you can infer that this call  $f(e_1)$  is of type  $\tau_{e_1}$  provided inside this  $\gamma$  you have this binding for  $f$  that it is of type  $\tau_{e_0} \rightarrow \tau_{e_1}$  that's how all our rules are being going and that's how side conditions are given. The premises and the conclusion of any rule consist of just the syntactic subexpressions of the phrase that is under consideration. So let us look at the dynamic semantics.

Of course in a dynamic semantics what we often do is we transform the syntactic phrase. It is part of the symbol manipulation of the run time. So, in the dynamic semantics we have the notion of a run time environment  $\rho$  which of course is a binding from identifiers to values and now values take on a new meaning.

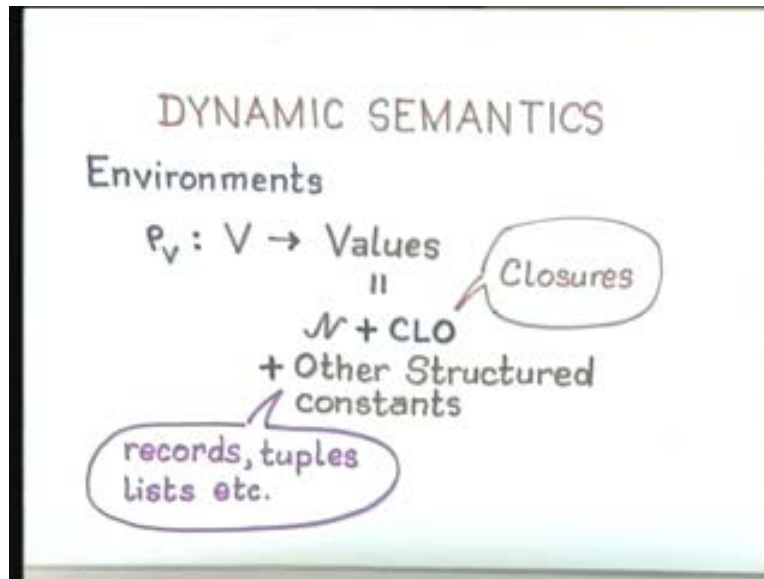
Values are not just values in the base types of your language, values include also values of higher types constructed through various functions. So this value is now a loaded word, as I said functions are also values, functions of functions are also values and all higher order functions are values. And of course looking at it the other way from the lambda calculus everything is a function whether it's a value or function. So these values



include all these values of higher types that you might have starting from the construction of your base types.

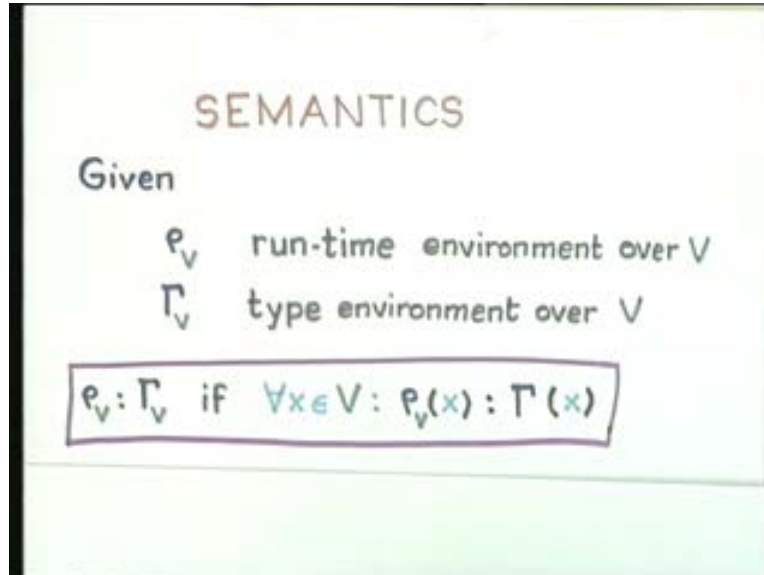
If you look at values originally we had just this, if you go back to some of the age old semantics that we gave we had only the natural numbers. But of course we showed how you could construct other structured constants from this and we also include what are called closures.

(Refer Slide Time: 28:11)



The other structured constants could be records, tuples, lists etc and these base values could be something other than int2 it could have int and bool and real and character and so on and so forth. But in addition we will have this thing called closures. A closure is really nothing more than a lambda abstraction. So let us look at the semantics now. So, given a run time environment rho over a collection of identifiers V and a type environment V as you might also have to do run time type checking in certain cases now we will look at the semantics in terms of both a run time environment and a static environment or a context gamma and both the run time environment and the context are over the same set of variables. So a basic constraint which has to be satisfied by these two environments is this that the run time environment should somehow be type consistent with the static environment.

(Refer Slide Time: 29:23)



Unfortunately there is a plethora of colons here and colons are a very highly overloaded operator. This colon here is a new symbol which essentially says that the run time environment over  $V$  is type checked by the context gamma over  $V$  provided this colon is a colon that I always use (Refer Slide Time: 29:59) when I write a logical predicate with quantifiers, it is something that separates the variables inside a quantifier bound to a quantifier. This colon is the colon of the type. So what we are saying here is this. The run time environment is type checked by the type environment gamma provided every variable in the run time environment has a value which is given by rho  $V$  applied to that variable and that value should have the same type that gamma gives it.

This is the constraint that we are always going to work under. This ensures that things also run time type check if necessary. So this new colon is really an extension of this old colon. These dark blue colons are really essentially the same. This colon is over particular variables or values, this colon is over collections of variables or values, it is just an extension from single variables or values to collections of variables and values. So these two dark blue colons, the left one is actually an extension of the right one. (Refer Slide Time: 31:35) This light blue colon of course is just a part of my symbolism for writing predicates and there is nothing more to it. This is the basic constraint under which we can define the semantics. So let us look at function definitions.

What happens at execution time?

I have removed this  $V$  as a subscript given a run time environment rho implicitly over some collection of identifiers  $V$  such that it type checks with the type environment gamma which is again over the same set of identifiers  $V$ , this is a function definition which goes so this is a function declaration. So it creates a new little environment and that new little environment is a run time environment. All identifiers have actual values and what is the value of the function  $f$  in this new little environment, it is a lambda abstraction over some expression containing  $e$  such that this huge expression is of type

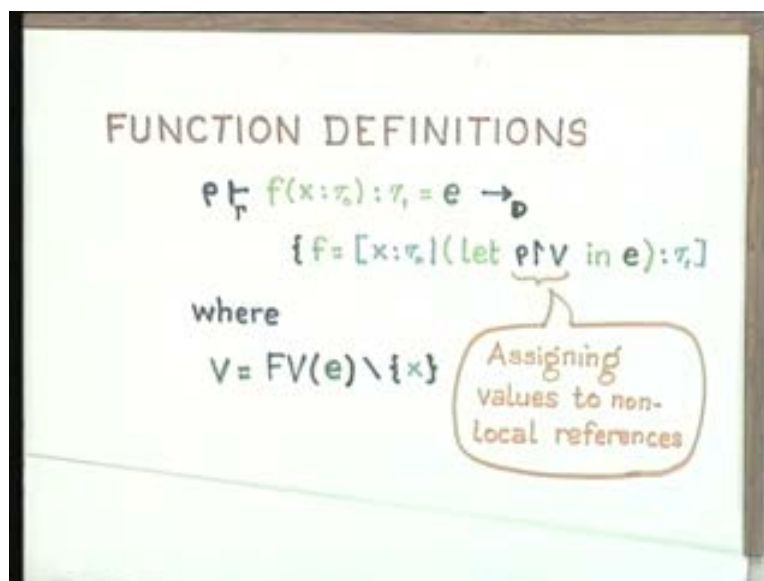
$\text{tou}_1$ . Therefore one constraint of course is that this  $\text{tou}_1$  and  $\text{tou}_0$  are specified here. Now what does this huge expression does?

What are the kinds of variables that are there in  $e$ ?

Here  $e$  could have bound variables namely  $x$  since we are considering only unary functions  $e$  could have  $x$  as a bound variable,  $e$  could be a ML like expression so it could have let and local and so on and so forth and more and more declarations in it which are all bound variables.

Let us assume for simplicity now that  $e$  is non recursive so it has no occurrence of  $f$  then what are the other identifiers that could be there in  $e$ ? Those are all the non-local references. So  $e$  has to be evaluated assuming that all those non-local references have values given currently by the environment  $\rho$ . Hence now I want to look at this function declaration in isolation. I am saying that this function has a value which is given by lambda  $x$  which is the argument of this function so essentially parameters are going to be bound variables in a lambda abstraction. So if you have several parameters then there are going to be that many lambda abstractions either in a single lambda abstraction over a tuple or that many carried lambda abstractions. The two things are isomorphic.

(Refer Slide Time: 36:57)

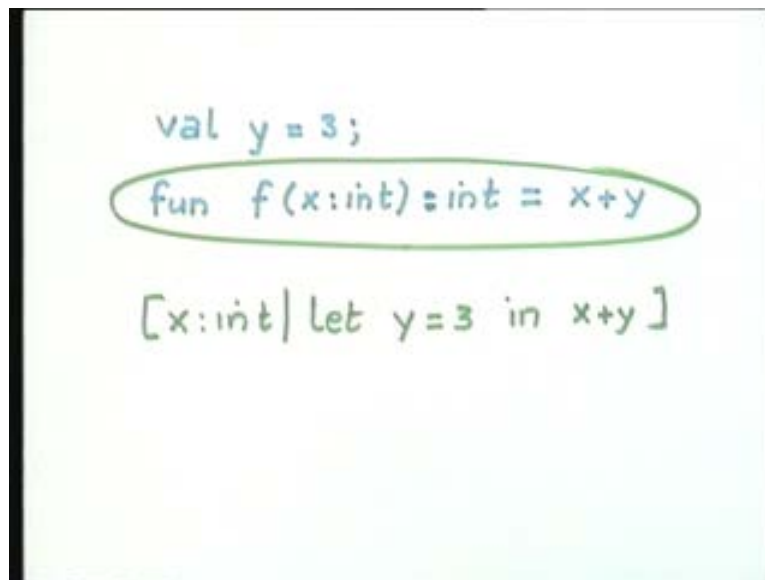


But since we are considering unary functions for the present for simplicity so it's a lambda abstraction over  $x$  and the expression that it has cannot just be  $e$  because  $e$  has too many free variables which are essentially non-local references. So  $e$  could have all kinds of bound variables in it in addition it could have some non-local references which have values in the current environment given by  $\rho$ . Essentially if I want to specify this function declaration in isolation that means if I want to remove it from its current ML session, this is a function declaration that's part of a larger ML program.

And now if I want to take this function declaration out of that ML session and put it in a separate ML program as a stand alone ML program by itself then I should in the current ML session extract the values of all the free identifiers that occur in  $e$  assign them the same values and then I am ready to take this definition out and look at it in isolation and this is what that has. For all the free identifiers in  $e$  extract their values as given in the current environment and create fresh “let” declarations which give you those values in the current environment.

So essentially what I am saying is if I have a simple ML session in which I first defined  $\text{val } y$  is equal to 3 and then now I define a function  $f\ x$  of type integer and let us say this is also of type integer and be completely explicit and this is equal to just  $x$  plus  $y$  for simplicity I am not considering any non-local references then what I am saying is what is the meaning of this function, this function has a meaning in this current environment and what is the meaning of this function, the meaning of this function is the same as replacing this by a new lambda abstraction  $x$  of type int such that “let” the only non-local reference here is  $y$  let  $y$  is equal to 3 in  $x$  plus  $y$ . So essentially that is what happens.

(Refer Slide Time: 38:55)



```
val y = 3;  
fun f(x:int) : int = x + y  
[x:int] let y = 3 in x + y]
```

It is a closure because now there are going to be no free variables in this lambda abstraction. The lambda abstraction that you now get is a completely closed lambda expression with no free variables. That's why it is called a closure.

The reason we have to look at all abstractions as closures also has a pragmatic basis and that is when you look at a function or procedure as an abstraction then what you are trying to say is that it is something that stands by itself. I can take that function from you for the same purpose for which you defined that abstraction for a similar computation and I can call that which means I can pull it out of your program or from a library of programs and use it somehow. The only problem that is created is then there are non-local references.

How do I resolve those non-local references?

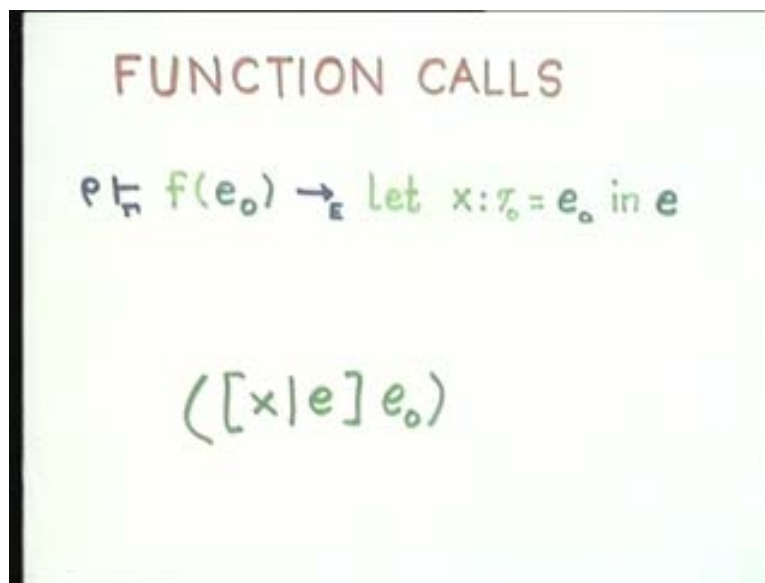
If you had non-local references in your function I extract all the values that are expected at that time and use them only then I am guaranteed that it will work right and that's essentially what I am doing by this closure. Therefore an abstraction is really that, I mean an abstract is really an abstraction in that sense. And that is in fact the purpose of an abstraction too. It is something that should stand out in isolation.

Unfortunately we have a lot of environmental variables and so on and so forth which prevent us from taking it directly. So in order to give an abstraction a meaning in isolation you have to also include the information that of whatever is relevant from the environment should go into the meaning of the abstraction and that's really all that we have to do. So let us look at function calls. That's the function definition then as a closure of function definition it stands out by itself as the lambda abstraction.

What is a function call?

A function call is just this. If a function definition is a lambda abstraction then a function call is just an application for that lambda abstraction so it's a beta reduction. And of course in a ML like language we don't directly have beta reduction but we have beta reductions in another form that is in the form of "lets". So if you look at "let" carefully it is really like an application. If you look at the semantics of "let" what we gave as the semantics was that you create a new little environment where this x has this value and then in that environment you evaluate e which means that all free occurrences of x in e will take that value from the new little environment that is created. But that's another way of saying forget about this x replace x completely by the expression that it denotes and keep the environment the same do not create a new little environment.

(Refer Slide Time: 44:46)



So our semantics of "let" could have been alternately written out as a pure form of beta reduction with no notion of an environment. In a purely functional language we could

actually do that. Every variable can be replaced by its body and often out of context. But in imperative language because of the possibility of side effects and because of the fact that there is a huge amount of environment that really needs to be maintained it is necessary to keep that environment information somehow.

But otherwise in a pure ML like language that we are defining we could have defined without an environment by just using the notion of a substitution which makes a “let” construct in ML just a beta reduction. A “let” construction in ML is just a beta redex so let  $x$  is equal to  $e_0$  in  $e$  is just  $\lambda x e_0$  applied to  $e_0$ . So this let actually is just like an applied lambda abstraction assume that there are no local references in  $e$  then this would be just and we wouldn't require any environments, you could actually do syntactic replacements. But we require environments because there are re-declarations of variables and so on and so forth so it is more convenient to use. But essentially a “let” construct is just a lambda is just a beta redex so a function call is just a beta reduction. Since a function is just a lambda abstraction a function applied to some argument is just the lambda abstraction applied to that argument it is just a beta reduction. So the kind of mechanism that we have been looking at so far really is what is known as call-by-value.

(Refer Slide Time: 45:43)

$$\text{CALL-BY-VALUE}$$

$$\frac{p \vdash_T e_0 \rightarrow_E e'_0}{p \vdash_T f(e_0) \rightarrow_E f(e'_0)}$$

$$p \vdash_T f(m) \rightarrow_E \text{let } x:\tau_0 = m \text{ in } e$$

where  $m:\tau_0, \quad p(f) = [x:\tau_0 \mid e:\tau_1]$

Therefore essentially what we are doing in call-by-value, I am redoing that function call I can rewrite that function call more explicitly in this fashion so that it models our further discussion on parameter passing. The function is called with an expression  $e_0$  then you evaluate  $e_0$  in the expression language. So essentially you don't touch the function till you have evaluated  $e_0$  completely and  $e_0$  has finally got reduced to some value  $m$ .

Now here again this  $m$  could be any higher order value, it could be a function value or whatever, it could be anything. But I am using  $m$  in order to keep it uniform with whatever we have done before as a hard value that is somehow available. Then the application of  $f/e_0$  is just the application of  $f$  on  $m$  which means that it is just this “let”

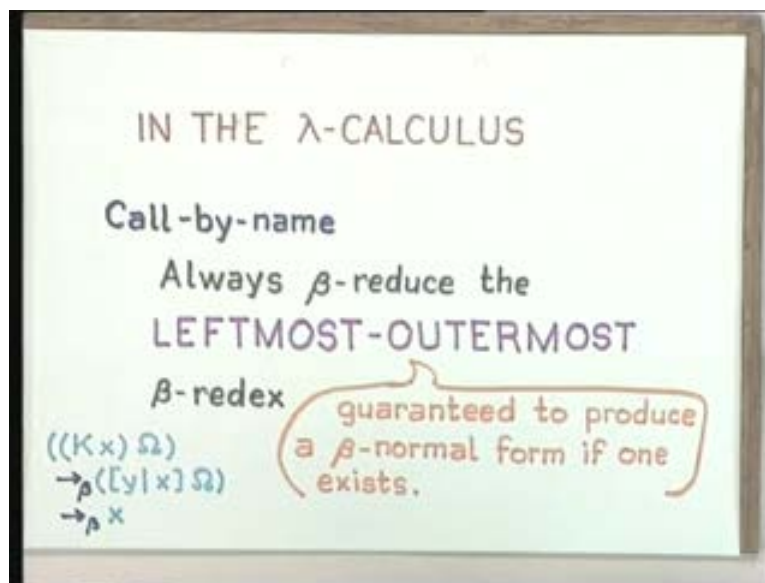
construct where  $f$  is just this lambda abstraction and of course here again I have to worry about those non-local references and so on and so forth so there is a “let” inside “let”. Do not take this  $e$  very seriously as this  $e$  might have non-local references which means again you will have to write this kind of an expression (Refer Slide Time: 47:22) so may be I will just give it a different name. I will just give it the name  $e_1$  where  $e_1$  is equal to let rho over the non-local references in  $e$  where of course the function definition is of the form  $f(x)$ ;  $\text{tou}_0$  is equal to  $e$ .

Therefore here what are you doing in call-by-value is you are evaluating the argument completely and the function call is applied only after you have got a perfect value. So if the evaluation of this argument is in infinite computation what it means is that you can never return from this function call which is something we all have experienced. Thus that brings us essentially to what is known as to various parameter transmission mechanisms application in the lambda calculus, in functional languages and also other imperative languages.

Therefore what happens in the lambda calculus?

In the lambda calculus you have two mechanisms one is called the call-by-name. These are the two principle mechanisms which have applications also in other programming languages.

(Refer Slide Time: 49:01)

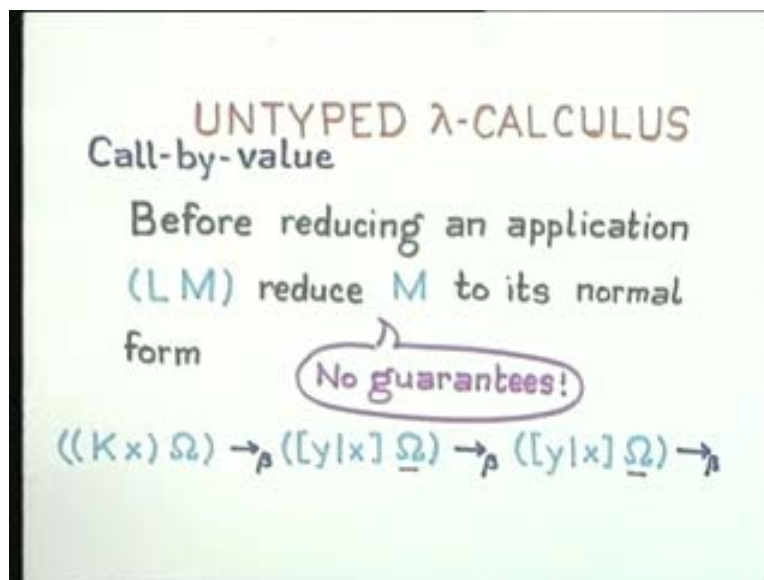


Hence what happens in the lambda calculus also happens in other programming languages to some extent. So, if you remember we considered this infinite computation and we said that because of replicating operators in the untyped lambda calculus you can get infinite computations. Even though by taking a different reduction route you could get a normal form.



Now there is a theorem in the untyped lambda calculus that if you take the leftmost outermost beta redex always and reduce it and having reduced that you will get a new lambda expression. Again choose the leftmost outermost beta redex. That means don't go deep inside to choose the redex. Look at the outermost levels of parenthesis to find the redex from left to right, read from left to right and do this. If you do a leftmost outermost reduction always which is deterministic remember that the lambda calculus operation semantics is not deterministic. But now if I make it deterministic by choosing always the leftmost outermost beta redex to be reduced then there is a theorem in the lambda calculus which I am not going to do is that if there is a beta normal form then this order of reduction is guaranteed to produce it. This is clearly a case of leftmost outermost beta reduction and this is called call-by-name.

(Refer Slide Time: 50:58)



Whereas if you have a call-by-value then you evaluate an operand completely and try to reduce it completely before actually doing a beta redex, you go deep inside. So essentially while you are reading the lambda expression you look at all the operands of beta redexes and try to reduce them.

An operand of a beta redex could itself contain beta redexes inside which means you go inside and try to reduce the operands within there and so on and so forth. So you are actually going inwards but you are not going too deep in, in the sense that you are not going deep inside the lambda abstractions you are only going deep inside the operands in an attempt to first reduce the operands to their normal forms and then apply the beta reduction.

Therefore, given an application of this form you won't go deep into  $L$  you will go deep into  $M$  you will want to reduce  $M$ . In order to reduce  $M$  you find out whether there are any beta redexes in which case you take their operands and try to reduce them and so on and so forth. So you always look at the operand of an application and try to reduce it first.



That is really what we did in call-by-value. Given an application you take the operand and try to reduce it first and then do the application. Unfortunately in the untyped lambda calculus this can give you an infinite computation. So it is not guaranteed to produce normal forms even if they exist.

We use call-by-value can converge faster. Once you have typed lambda calculi there are no infinite computations, normal forms are guaranteed so now call by name and call-by-value should both yield the same results. But in a typed lambda calculus that is a polymorphic type you can have operators which are replicating. So, if you have a replicating operator  $L$  applied to an operand  $M$  then a call-by-name can produce lots of copies of  $M$  which means that you will have to reduce each of those copies of  $M$  individually later.

On the other hand, if you decide to reduce  $M$  first itself and then you do the application even if  $L$  is a replicating operator you still have a normal form without doing extra reductions on copies of  $M$ . So it is also the easiest thing to implement a call-by-value reduction and it converges faster especially in the presence of replicating operators. That is why an untyped lambda calculus is completely of no use to man or beast because its semantics requires that you will have to always use a call-by-name implementation if you want to get normal forms. Therefore most functional languages use call-by-value. So we will start parameter passing in this context next time and we will look at other parameter mechanisms in imperative languages.