Principles of Programming Languages
Dr. S. Arun Kumar
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture - 35
Contexts

Welcome to lecture 35. So, contrary to what I said last time it won't be actually possible to start on abstraction today so I will complete whatever I had to do on type checking in the context of contexts and type inferencing and type checking with contexts. It is something that we had not specified last time. We have just assumed somehow that information was available in some form especially for the "while language".

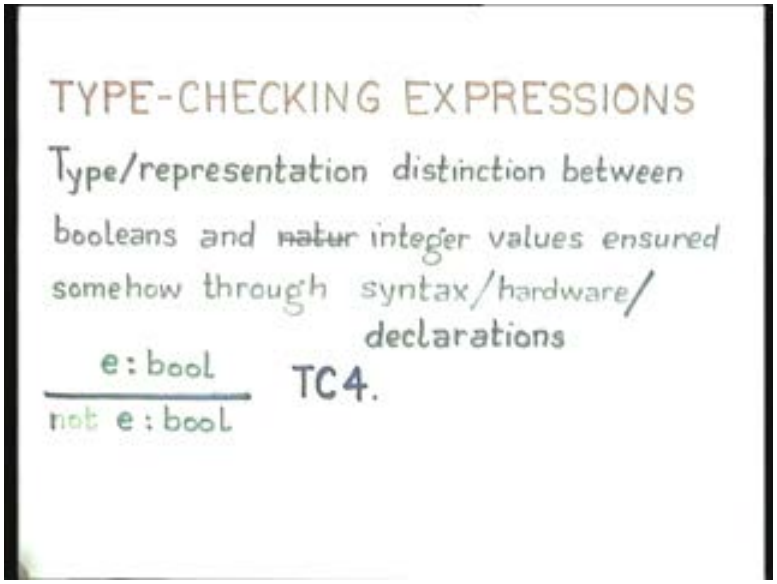
The implicit assumption is that just as in the run time case we assume that there was a notion of a state, we are never starting with an undefined state and the notion of state that we assumed there in the run time system was that there was a variable to value mapping available for every variable. Since there are no declarations in the while language as we have defined it the implicit assumption is that there is a variable to type binding implicitly available at the start and given that it is always there.

(Refer Slide Time: 1:41)

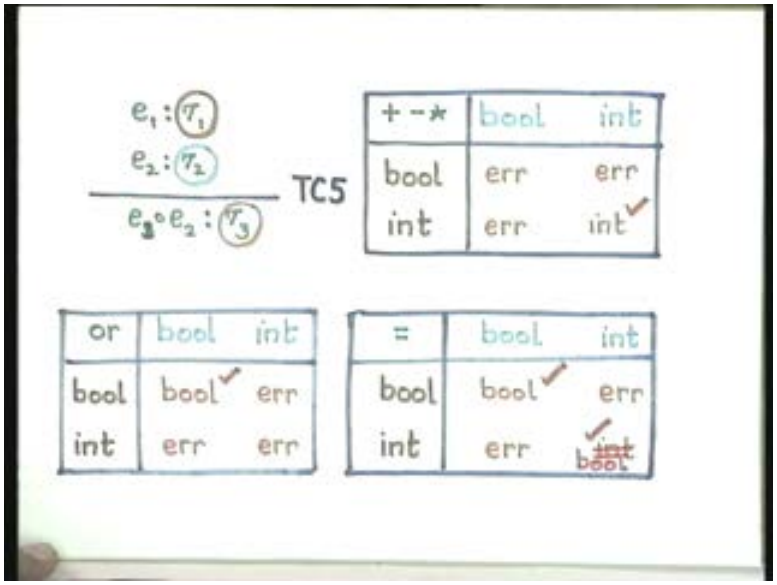
Since there are no declarations we were forced to assume that there is an infinite collection of variables, all of them had some value assigned to them in the run time environment. Similarly now we will assume that in the case of "while" we have an infinite collection of variables and they all have some type associated with them, may be one of the base types. So with those assumptions in mind we actually gave these simple rules for type checking and one reason for giving such type checking rules is that they are very elementary and lets dispose off expressions and get on to context get on to contexts

so that we get to know how exactly type inferencing is done, how type inferencing and type checking are related and so on.

(Refer Slide Time: 2:33)



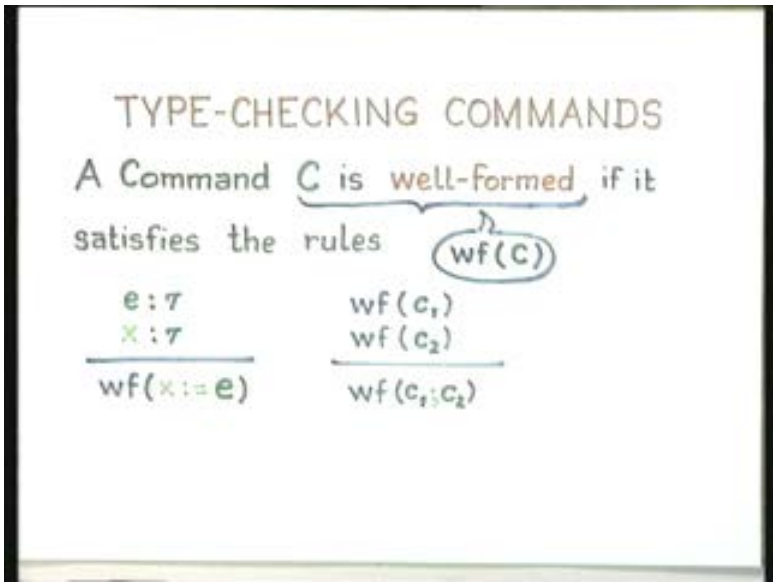
(Refer Slide Time: 3:37)



Therefore we assume somehow that a context or in type information is available for the basic elements in the language namely variables and if there are constants then constants too are available and we had these basic type checking rules so that if you can somehow infer that e_1 is of type tou_1 and e_2 is of type tou_2 then for any binary operation e_1 binary operation e_2 is of type tou_3 where tou_1, tou_2 and tou_3 are listed through these tables. As far as commands were concerned commands do not have a type. However, commands are

made up of expressions so we have to type check commands in a way which guarantees syntactically that a command is somehow well formed out of the right kinds of expressions. So with commands since they are just state transformers we just assume a predicate called well-formed.

(Refer Slide Time: 4:18)

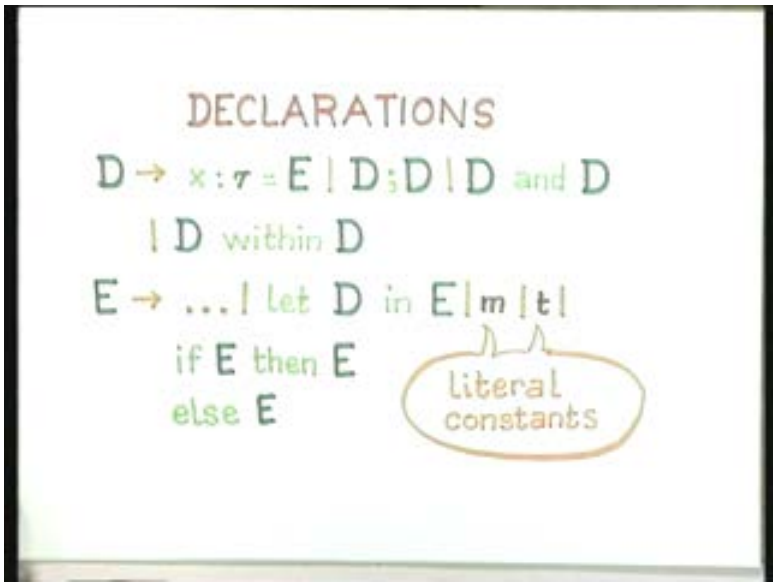


So the type checking rules for the assignment command that you should assign to a variable only a value expression of the same type and that you recursively go through the commands based on the structure of the command and you type check complex commands in terms of the type checking rules for the simpler commands. Therefore, if any of these components does not type check is not well formed for some reason then the whole command is ill formed, then the whole command does not type check.

Similarly, for the rest of the complex commands the one extra type checking rule we require is that you actually have Boolean conditions and not arbitrary expressions. So now let's actually look at it in the context of a functional programming language with declarations. If we have an imperative command structure the problem will become too complicated. Essentially this is to illustrate the use of contexts where you do not have an infinite collection of variables you have only finite collections of variables available at any instant and there are declarations which specify the types in which case how type inferencing is done.

So if you were to extend this language to include a language of commands essentially the type checking or type inferencing for the declarations and expressions would proceed in the same way and in addition the commands would have to be well-formed by appropriate rules. So we will not worry about commands so we will just look at a simple functional language which is like ML if you like that follows a similar syntax.

(Refer Slide Time: 6:24)



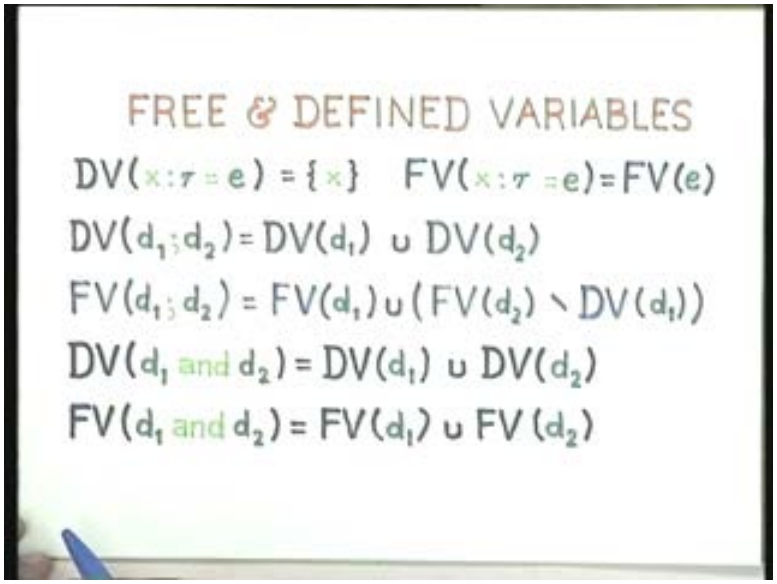
Therefore we have these declarations, you can have a declaration of this form, in ML it would be val x equals something. In ML also you don't always need to give the type tou but that's because ML is not a type checking system and more than a type checking system it's a type inferencing system. So, if the type of E is clear then what ML will do is it will automatically assign that type to this variable x. Therefore right now we will assume that declarations are all well typed which means we are actually imposing Pascal like declaration discipline in the language for the present moment and then you can of course compose declarations in the usual fashion and just for variety I have added an extra form of an ML type expression the "if then else" which is an important form and from the "if then else" it is also easy to derive options where you have options on the structure, options on pattern matching and so on.

So we have this expression with a declaration in it and what holds for this expression with the declaration in it also holds in case you extend this to commands for commands with declarations in them. So Pascal like declarations are very easily taken care of and then of course you have these literal constants. In the case of declarations the moment you bring in declarations you are bringing in bound variables or free variables, the notion of free variables becomes important and we have this definition of free and defined variables. So the base case is something of this form, the only defined variable here is x and every variable that's free in e is free in this entire declaration and when you compose declarations you get similar structurally defined sets of defined variables and free variables.

So the question is whether x can occur in e in the first declaration. Now, the way declarations work in ML is that the x that occurs in e is a different x previously declared in some outer scope from the x that is now being re-declared, that x is still free so it is still a free variable of this declaration. If you have taken all x's in e let us assume you give them a different color and in the free variable set it will be of that color whereas in

the defined variable set it will be of this color. So these natural static scoping rules apply and any free occurrence of x in this refers to an x declared in some outer scope and it derives its type also just as it derives value from that outer scope it also derives its type from that outer scope where the declaration for x occurs. And d_1 semicolon d_2 similarly has all these defined variables.

(Refer Slide Time: 10:32)

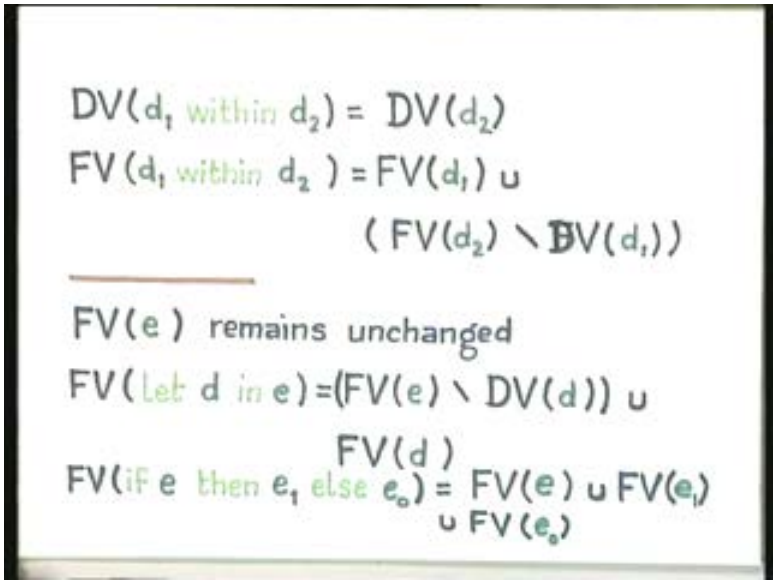


The free variables of d_1 semicolon d_2 are that, anything that's free in d_1 is free in d_1 semicolon d_2 and whatever is declared in d_1 could be used in d_2 so anything that's free in d_2 except that those that have been declared in d_1 this union constitutes the free variables of this declaration. Similar things apply here for d_1 and d_2. Of course there is a disjointness condition for d_1 and d_2 the same name cannot be declared in both d one and d two that disjointness condition is made clear in rules of inference.

In the case of the nested declarations a nested declaration really refers to the declaration d_2 which might be very complex so you might require additional names to abstract out some of the sub expressions and give them new names. So you have the declaration d_1 to aid essentially what is required is the declaration d_2. The only defined variables in this declaration are those that are defined in d two and the free variables in this are all those variables that are free in d_1 and as usual the variables that are declared in d one might be used in the declaration d_2 so the free variables of d_2 excepting those that have been declared in d_1 this union (Refer Slide Time: 12:23) actually constitutes the free variables of this declaration. For all other expressions that we have already seen before the notion of free variables remains unchanged and in the case of this let expression we need to specify free variables and declare defined variables in declarations in order to enable a definition for the let construct essentially. So, all these machineries are precisely for this so the free variables in this are exactly the free variables in e.

Of course the free variables in e could include some variables that have been declared in d so remove those because they get bound union the declaration d involves expressions which contain variables previously declared in the scope in which this let construct occurs or in some outer scope in which this let construct occurs so the free variables of d are also free in this expression.

(Refer Slide Time: 13:28)



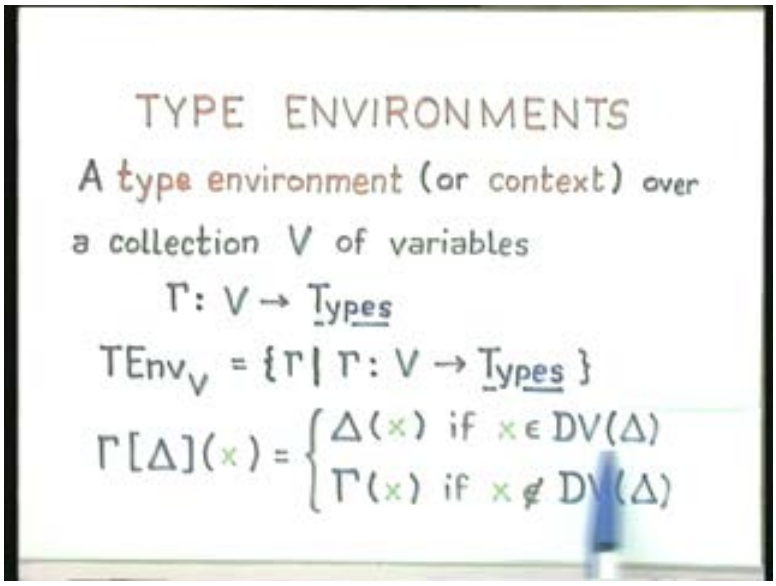
Here of course it is very trivial it is just the union of all the free variables that occur in each of these expressions. Now we are going away from infinite collections of variables. Since we have declarations we can restrict ourselves to actually the variables or the names or the identifiers that you actually have in your program. You don't need to assume an assignment of types or of values to the infinite collections of identifiers that you have. So we have the notion of a type environment or what we have called in the lambda calculus a context. So a type environment or a context over some finite collection V of variables now this would be a finite collection is just a variable to types binding. This "types" is meant to denote whatever types that are generated through some context free grammar in the language of types.

When we say variables they are not really value variables they could be identifiers which denotes functions a higher order function but for the present we will just restrict ourselves to this as to whatever "types" means. So, in the type environment a given set of variables is just a set of all possible contexts that you can define. There are no type variables there are only value variables but we have a set of types generated by some context free grammar.

In fact for most of the examples here this word "types" will denote only the base types, type variables are really required when you are dealing with polymorphism but we have already dealt with polymorphism and we are doing a backward integration into lower levels so lets restrict ourselves to this for the moment. Hence, just as in the case of our

environments we require the notion of a temporary updation on this static environment. Static environment is another word that is used interchangeably with context and type environment, a temporary updation of the type environment. Given that capital gamma and delta are both contexts over some specified sets of variables this updation of gamma by delta is such that for every identifier x the type of x is given by delta if x is a defined variable of delta.

(Refer Slide Time: 17:00)



If x is somehow a defined variable of delta what I have done is I have stealthily extended the notion of defined variables from syntax to the contexts. so essentially if you think of a context as a collection of this form (Refer Slide Time:17:41) may be z of type int arrow int or some such thing if you consider a context to be a collection of this form then the defined variables are x, y, z etc. So I have actually done that without explicitly mentioning it so we have the concept of what are all the defined variables in a context so this updation is such that if x does not occur in the defined variables of delta then the type of x is whatever is given by gamma.

Gamma is a context, delta is a context and gamma updated with delta is a new context and how is this new context defined? It's defined in terms of the contexts gamma and delta as follows: for any identifier x in the collections of variables over which gamma and delta are defined the type of x is what you are interested in a context and in the type of x whatever is the type of x in delta if x occurs as a defined variable in delta. If x does not occur as a defined variable in delta then the type of x is whatever in the context gamma. So this takes into account redefinition of a variable.

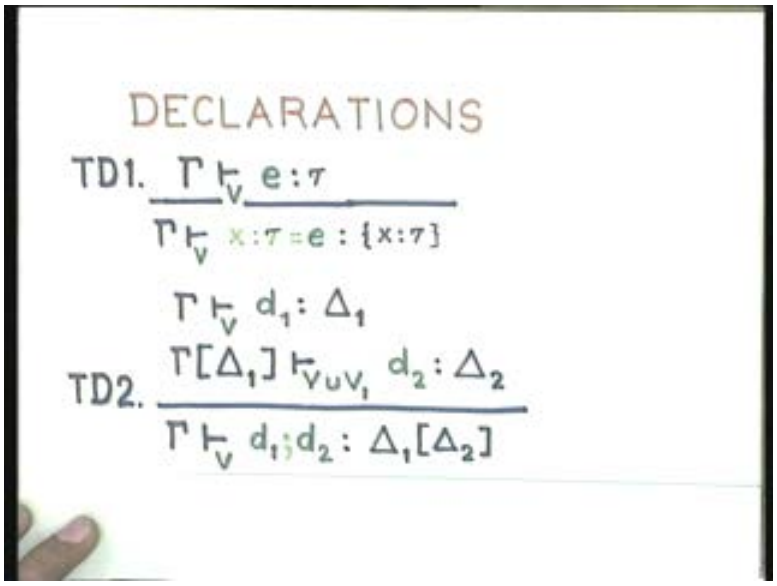
So how do we process declarations for type checking?
We have the rules of inference which are very similar to the rules that we had for dynamic semantics in the presence of run time environments. We are considering a collection of variables v and essentially this context gamma has this collection of

variables v as defined variables. With this assumption of the types of the variables given in gamma if you can somehow infer that the expression e is of type tou then this declaration of x is well typed and what it means is that this declaration creates a new little context just like we had new little environments this creates a new little context which consists of a type binding type for x and that type binding type for x is tou.

So the type of x is tou and it creates this new thing. So this rule actually does not do any inferencing it does only type checking. So it creates this little environment only if e has somehow been inferred to be of type tou otherwise this rule is not applicable which means that your declaration does not type check. The same kind of default rules that we had for the dynamic environments also apply here, what is not specified is not allowed. Hence if e were of some other type some type other than tou then what it means is that this rule does not apply because this declaration does not type check.

Therefore, no new little context is created and essentially the translation stops right there, no further translation takes place. Remember that this is something that happens at translation time as opposed to our run time environment. In the run time environment if none of the rules apply then what you can say is that essentially the execution stops there, it aborts and it does not give you any results. Similarly the translation stops here if something does not type check.

(Refer Slide Time: 23:18)



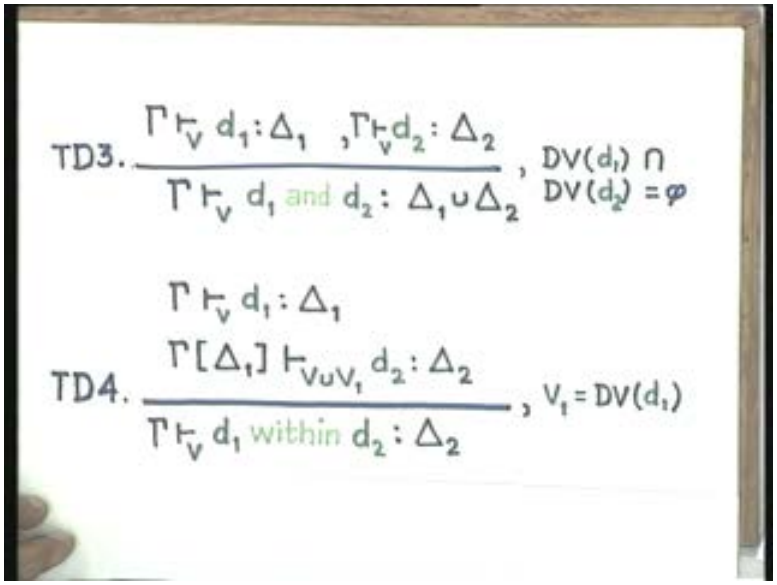
Declarations produce little type environments. In the context gamma which means somehow all the free variables in d one have type bindings in gamma. Assume that all the free variables in d_1 have type bindings in gamma then since each of those declarations in d_1 may be something like this for example each of them create a collection of bindings.

Assume that we can somehow create a collection of little type environment which creates the bindings for all the defined variables in d_1. Since the defined variables of d_1 could be

free in d_2 in this updated environment where delta one is the little environment created by processing d_1, it might contain some new variables v_1 so in this updated type environment if you can infer that d_2 creates a little context delta two then in the original context gamma these declarations $d_1;d_2$ actually creates the little environment $delta_1$ updated by $delta_2$. Note that this allows for the fact that the same variable might occur in both d_1 and d_2. This is not allowed in languages like Pascal but it's certainly allowed in ML. So what happens is that within this in the context of this composite declaration the most recent declaration of that identifier is what is used throughout and the same thing also happens in the run time environment.

The value of the most recent declaration of the identifier is what takes effect. This is how contexts are created and now for the other two declaration mechanisms we have this. Here we have this disjointness condition on this side that the declared variables of d_1 and d_2 should be disjoint and given that they are going to be disjoint both d_1 and d_2 are type evaluated in the same context gamma within which they occur and if each of them produces a new type environment $delta_1$ and $delta_2$ respectively then this composite declaration produces a new type environment $delta_1$ union $delta_2$ and this disjointness condition ensures that $delta_1$ and $delta_2$ are also disjoint and so the same variable does not occur in both.

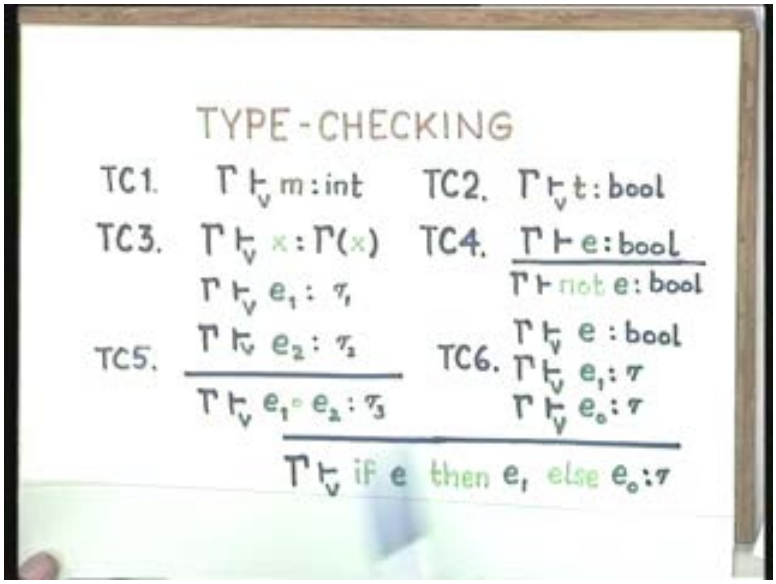
(Refer Slide Time: 26:46)



Finally for the "within construct" as I said d_1 is used in order to essentially create the environment d_2 the type environment defined by d_2 and of course the defined variables of d_1 might occur within d_2 as free variables. So we first process d_1 in the context gamma and let's assume that it creates a little environment $delta_1$ with a new collection of variables v_1. Now in this updated type environment gamma $delta_1$ process d_2 to obtain a new little type environment $delta_2$ and the net result of this declaration is to create this new type environment $delta_2$.

So now we have seen how actually declarations create contexts and now the type checking becomes quite easy. Thus we can look at type checking with contexts. Hence if you assume you have given context gamma which means the collection of variable to identifier to type bindings then that is the context we will carry throughout. In the case of these integers and so on and so forth assume that there is some syntactic mechanism by which even without explicitly specifying it may be you can infer the type or in other cases you might actually require an explicit declaration of this form. So in either case any constants that are their types are obtained by pattern matching of the forms. For any free variable its type is really whatever is defined in the context and for these Boolean variables we have the same old rules except that they are all in the presence of a context.

(Refer Slide Time: 29:29)



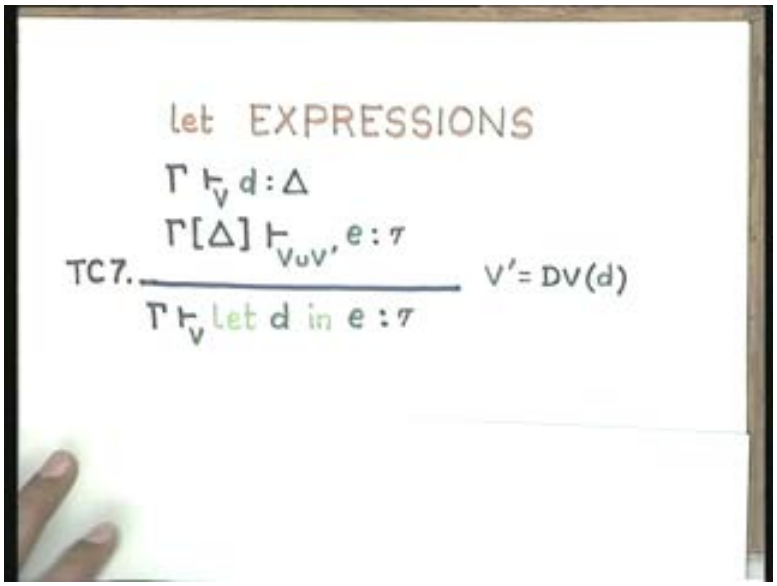
Therefore we have this trivial type checking rules; e_1 is of type tou_1 e_2 is of type tou_2 then e_1 circle e_2 is of type tou_3 where again of course the type tou_1, tou_2 and tou_3 are as defined by these tables here for integers and booleans. So what it means is you carry forward these tables for type checking here and lastly the ”if then else” should ensure that an ML type inferencing actually means that, in the case of commands as I said well formedness is enough, there is no question of command having a type. But in the case of “if then else” used in an expression language since the whole expression denotes a value it denotes a value of a certain type which means both arms of this conditional should have the same type. The result of this expression is going to be either the result of evaluating e one or the result of evaluating e_0 depending on whether e is true or false.

Therefore, essentially this whole expression is of some type tou which means that its eventual value that it returns in execution will be a value of type tou provided each of these arms e_1 and e_0 is of the same type tou, of course this has to be a Boolean otherwise it is not a condition at all. So, if this is a Boolean and each of the arms is of the same type which is not necessarily ensured in “if then else” in a typeless language, in an untyped language what can happen is that this e_1 and e_0 could be of different types and the result

of this if e then e_1 else e_0 may not be compile time determinable as a type. It could actually have different types and at run time after the evaluation of Booleans when you get the value you will also get its type. But in an ML type environment where we insist ML is strongly typed in the sense that at translation time the type of every expression is determined.

Then this expression must have a type and it can have that type only if both e_1 and e_0 are of the same type so that is how the "if then else" works here. As far as let expressions are concerned you have a declaration which creates a little type environment delta then you evaluate the type of this expression in the context of this new type environment delta. So you look at it in this updated context, determine the type of this expression where of course the side condition is that this delta might use new identifiers so this v prime is the set of all defined identifiers in the declaration d then in the context gamma the type of this let expression is really the type of this expression e.

(Refer Slide Time: 34:04)

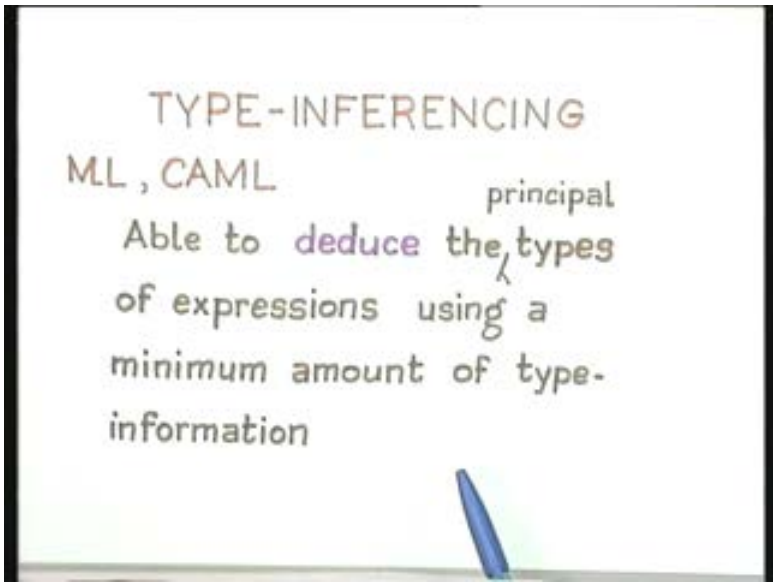


So, in a completely closed program just as you started with an empty run time environment you will start with an empty type environment and as you go through the declarations you will be building up contexts. As you go into inner and inner scopes your context will keep getting updated. As you exit scopes your context will also keep shrinking just like the run time environment keeps shrinking as you exit more and more calls till at the end where again you are left with an empty environment and that's at run time and this is at compile time which is the only difference.

Since it is at run time it specifies values and this is at compile time so it specifies only types. This is an interesting analogy between the structure of typing and the structure of the run time environment. There are other very interesting analogies too. Let's briefly look at typing. I initially said that typing is an important concept because it allows you to save on badly typed run time executions. So if you do type checking you can throw out a

program early in the game without even bothering to execute it. And it is been noticed that even experienced programmers may type mistakes and so it's a good idea to have type checking or type inferencing. So the difference between type checking and type inferencing is clearly this. Actually they both are very closely related.

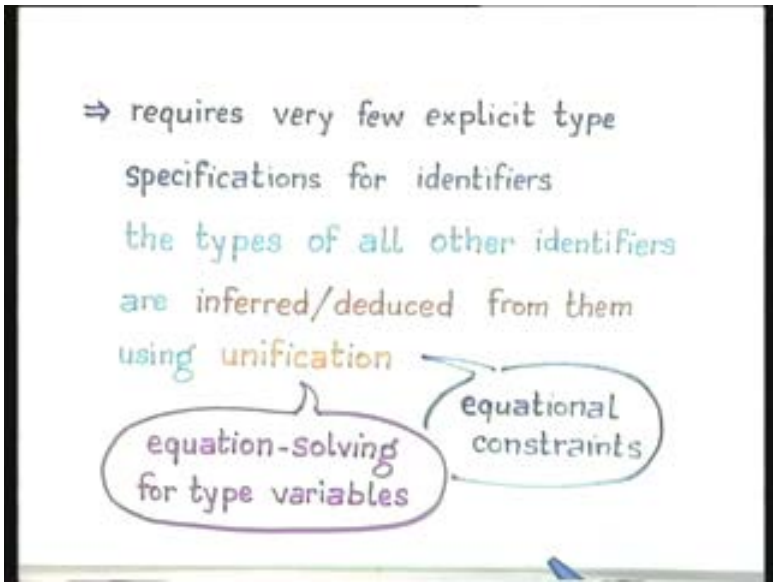
(Refer Slide Time: 35:47)



So if you look at languages like ML and CAML what happens is that in any ML session or a CAML session you might be declaring a large number of identifiers. One does not necessarily need to think of an ML program as a one complete whole, after all the fact that you can go through an interactive session means that you can keep changing things so as a result in a really long session you will be using a large number of identifiers, some of them useful and some of them useless as time goes on.

Now whatever may be the case what ML or CAML requires is only that you specify just enough type information so that the ML typing system can deduce what is called the principal type. Forget about the word principal for the moment, it can deduce the type of the expression. That is why it is not necessary for every new identifier to introduce the declaration like you do in Pascal. You can keep introducing new identifiers and as long as the ML type system accepts them and assigns them a type you are playing safe. So it actually does a form of deduction and this deduction is a form of equation solving using unification. So we will look at a small example of that but essentially it does deductions so it does type inferencing from very little type information.

(Refer Slide Time: 37:47)



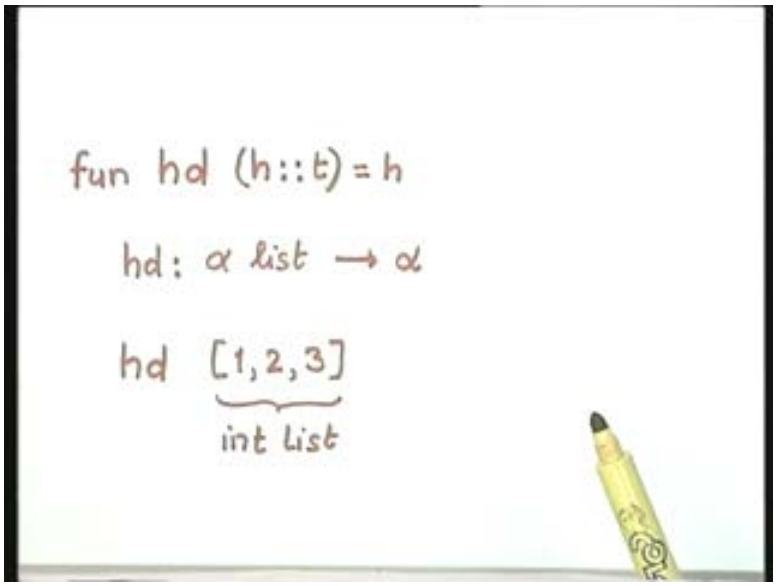
It means that you require having very few explicit type specifications for identifiers and the types of all other identifiers and expressions are inferred. Especially since ML uses pattern matching the use of pattern matching means firstly that it can use unification to do type inferencing, secondly it means that since pattern matching is also a facility in the language, it means that programmers are going to use a large number of identifiers and they don't all have to be explicitly declared. So ML actually infers the type of every identifier and then type checking the expressions is no problem. Hence type checking and type inferencing go on at the same time. And further for identifiers which do not have explicit base type definitions, for example I can define this function let's say head for a list in ML as normally I would so head of, I am incomplete pattern matching so I will just define this pattern h cons t and I will say that this is equal to h.

Now the point about ML is that there are at least three new identifiers; one is head, another is h and another is t. the types of none of them have been explicitly specified except that I use this reserved word fun and therefore it is clear that this head is not actually a base type, that is all it specifies but this application says that, this head is actually a unary function let us assume that there is some precedence so for example if I don't assume precedence then I put this brackets so all that you can really look at and infer from here is that head must be a unary function of, from this cons you actually infer that which applies on lists of some kind, I don't know lists of what kind but if that list satisfies this pattern h cons t then I return h. So what ML does is it actually assigns a type variable.

Since the types of h and t are not explicitly specified so ML says that this function is really a function from some alpha list where alpha is to the type alpha where alpha is a type variable. And since ML is polymorphic what it means is you can for example apply head to the list consisting of 1, 2, 3 and from the very syntax of this ML gathers that this is of type int list and it actually tries to apply head to this and sees whether it can do it.

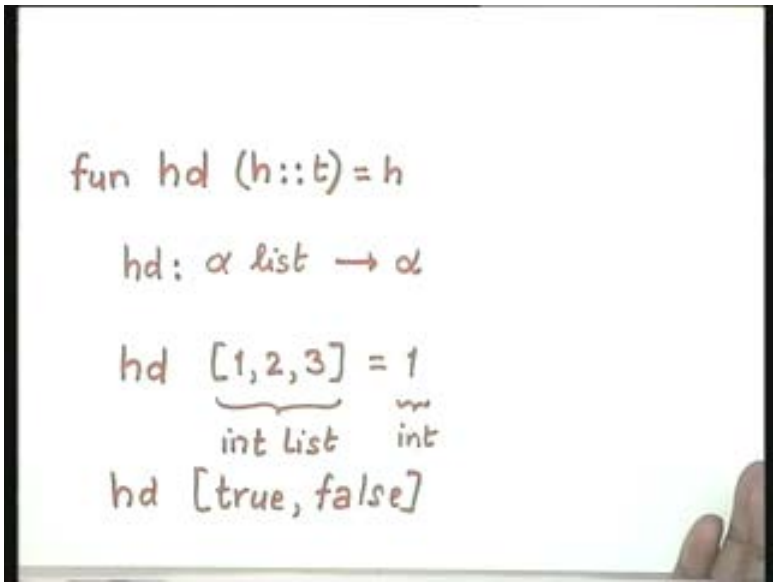
And it can apply head to this only if it can equalize this alpha list with int list. Again it uses its own pattern matching facility and infers that alpha therefore must be int here and for this particular context it assigns head. It assigns alpha the value of int. This is like a trivial equation solving.

(Refer Slide Time: 42:11)



What is the value of alpha such that alpha list equals int list, the value of alpha is int. It is the most elementary form of equation solving. It gives you this and then it actually types check this, so this is int. If alpha is int then this type checks. So you could for example apply this head again to something else like true, false and then it actually does this alpha list equals to this so it knows it is a bool list and then it does ad equation solving and for this application it does the type checking.

(Refer Slide Time: 43:03)

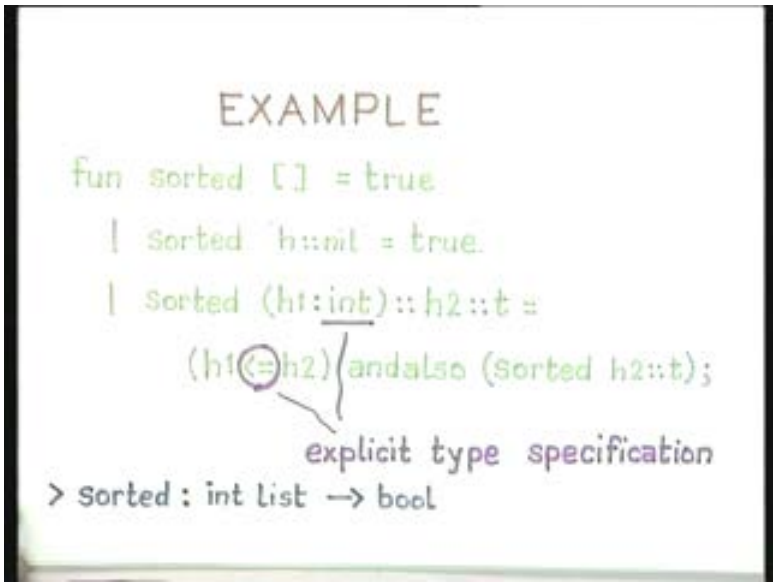


But ML actually is capable of very complicated type checking, type inferencing, equation solving with the unification so it uses all these words like list as function constructors in the unification algorithm. Arrow is also a function constructor in the unification algorithm. The base types may be the int bool real or whatever there is a type language but all other things are function constructors over which unification has to be performed.

For example, you could have alpha arrow alpha list. You could have a list of functions so arrow, list and even the record constructor, the tuple constructor, the star for the tuple type constructor are all function symbols to be applied to be used in a unification algorithm. So, for example list is a postfix, this constructor (Refer Slide Time: 44:20) is a postfix function constructor over the language of types. This arrow is an infix constructor over the language of types, the star is an infix constructor over the language of types and it uses all these and runs the unification algorithm to do equation solving and having done the equation solving and having inferred the values of all these variables not all variables will, if I stop my ML session here alpha will not have an explicit value it is still a type variable. But the point is that it assigns a principal type. That means it finds the most general unifier of that set of equations.

The most general unifier is usually not expressed in terms of a monotype always. I use head over lists of functions of type alpha arrow, alpha and so on and so forth then that alpha is never actually given an explicit base type definition in the language of types. It remains always a type variable. So it generates new type variables and does unification and finally has a list of type variables in terms of which all other type variables are expressed.

(Refer Slide Time: 47:16)



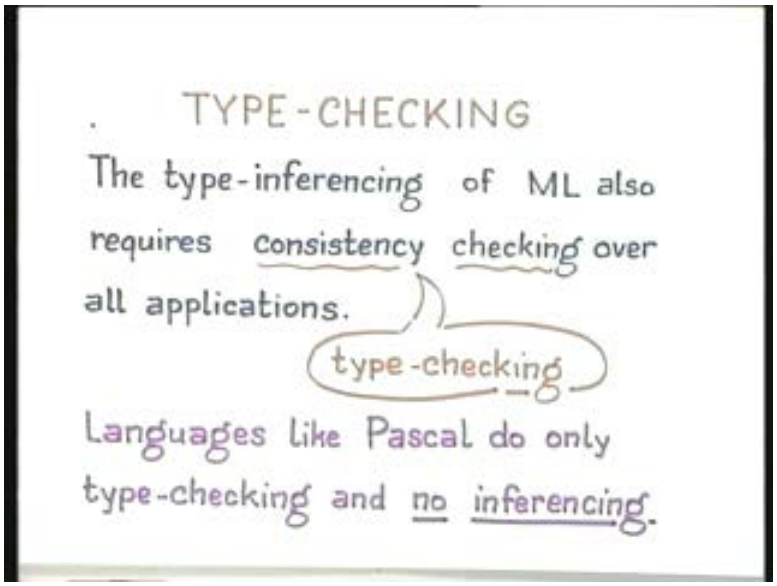
If it is a simply typed language then what it means is that at the end of something this alpha should have an expression, alpha should solve to an equation of the form int arrow int or something in terms of the base types, it should be an expression, it should be a type expression in terms of the base types but if it is polymorphic I just leave it as a variable and I do the type checking assuming that all the type variables that I have got have somehow been expressed in terms of some minimal set of type variables and base types in the type language.

So let us quickly look at this example. If you do not give these declarations, there are lots of identifiers here h, h2::t "sorted" there is no declaration for any of them. But there is a declaration for this h1 and this declaration is required because you are going to use a less than or equal to relation and this less than or equal to relation is not defined over all types it is not a polymorphic relation it is a relation that is defined in the ML environment only for integers, reals, may be characters for example it is not defined on strings.

In your own environment you might have defined something like a less than or equal to for some other data types but then you are still being absolutely specific about it. So this less than or equal to is not a relation that is freely available over all types and therefore the only way ML can type check this is if it knows which less than or equal to are you using then this declaration actually explicitly specifies and determines the type of the entire thing.

For example, from this declaration it follows that this has to be an int list and therefore h two has to be an int, therefore t has to be an int list. Therefore this empty list is the empty list of int lists and this h therefore must be an int list and this nil is also the empty int list and then of course then this less than or equal to type checks for integers, it doesn't type check for many other types and this whole thing is a Boolean so "sorted" has a type which is int list to bool.

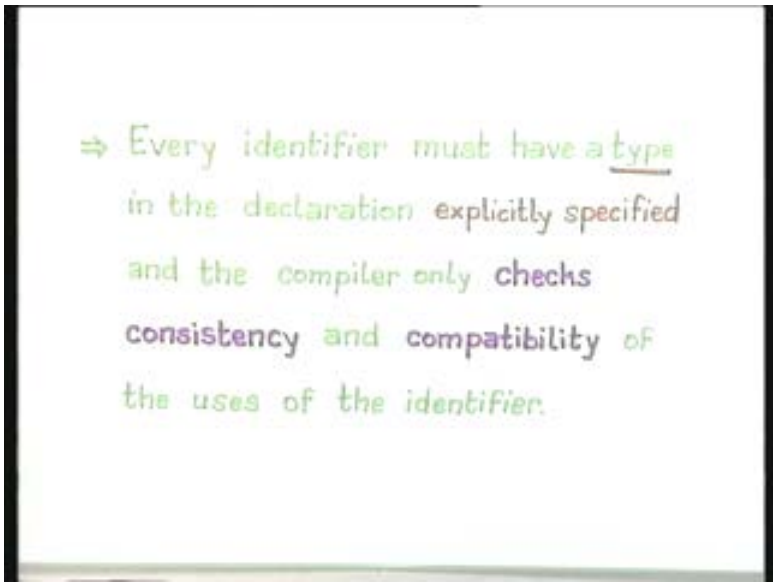
(Refer Slide Time: 50:03)



So the programmers are lazy and would like the convenience of not explicitly specifying types. But then when they do not explicitly specify type they also make type mistakes. So the best way to combine convenience with typing is to do type inferencing which is what ML does. The other extreme of course is something like Pascal which does only type checking. They expect that every identifier has an explicit type specified and after that they only do type checking they do not do any type inferencing. The type inferencing is restricted to expressions but then that is for checking consistency and that kind of type checking can be done trivially by just matching parities and so on, it doesn't really require a sophisticated collection of type constructors to do it.

Therefore, languages like Pascal do only type checking and actually even though I have written no inferencing they actually do some inferencing, after all expressions come without predefined types. Every expression is not type defined by the program. They do that amount of type inferencing which is already given by the rules and they do the type checking essentially to maintain consistency of types. Essentially that's how type checking goes on. So type checking and type inferencing go together. If every identifier has been explicitly declared then you require to do only type checking but you use the same set of inference rules for both.

(Refer Slide Time: 50:58)



If not every identifier is declared then you have to do type inferencing. In that example of sorted if you do not explicitly specify somewhere that it is an integer list, either in an implied form or in an explicit form if you do not specify in some place by which the inferencing system can determine what is the type then it automatically gives you a message unresolved type or some such type or ambiguous type unable to resolve type. So both ML and Pascal are actually statically typed languages in the sense that the types are determined at compile time, that's what static means.

Actually Pascal is quite strong but I have written weak here for a specific reason and that has to do with variant records where you can do type mismatches. But otherwise both ML and Pascal are strongly typed. Pascal has other weaknesses also like for example in procedures which have functions as parameters, the types are not clearly determined and they are not determinable really at compile time except at the call but more or less Pascal is strongly typed most types are determinable.

APL and Snobol and so on do not actually have a notion of compilation, they are very highly interactive so they have a very weak dynamic type checking facility. Just before the operation is applied in the execution they actually check the consistency of the types and see if the operation can be applied. LISP and scheme are mostly untyped except for the underlined base types for which type tags are already available in the hardware or in the for through firmware or through assembly or some such thing. That is how typing goes.

(Refer Slide Time: 52:04)