

Logic for CS
Prof. Dr. S. Arun Kumar
Department of Computer Science
Indian Institute of Technology, Delhi

Lecture - 37
Verification of Imperative Programs

We, will do more of the for computer science aspect of logic. So, we did logic programming we have d1 tabulo methods algorithms resolution methods. And today will do something that has always given first my first year students. Whenever they have had the misfortune of having me as teacher for programming headaches. So, will so let us look at that so we will actually look at it in a very simplified fashion in first year programming. The kinds of proofs that you have to do are very specific to those programs. And those algorithms and therefore they have to be d1 in much created detail but. We will look at it again will take a higher level view point and will just look at the theory of program verification. So, that is what will do today and its actually an extremely small capsule summary of program verification. From the point of view of hand proving of programs I will at some point either today or the next lecture. I will actually talk about the automation a little bit but, I will not go into it in great detail. So, we are looking at verification of imperative programs so what will do is like we have always d1. We will take a very small signature basically and will think about imperative programs again as parameterized on some signature.

(Refer Slide Time: 2:20)

The WHILE Programming Language

Let Σ be any signature. Then the programming language $WH(\Sigma)$ is defined by the following BNF.

$$P, Q ::= \epsilon \mid x := t \mid P; Q \mid [P] \mid \chi?P : Q \mid [\chi?P]^*$$

where $\chi \in QF(\Sigma)$.
A program is merely a *state transformer*.

NPTEL

So, that means we define a while programming language is very simple it just contains it is imperative. So, it has to have assignment it has to have sequencing of instructions it has to well it has a conditional that is not necessary it has to have a looping construct. So, it is non-recursive program so we are just considering a simple version of non-recursive imperative programs. And just because case analysis is very common I have also put in a conditional. So, this epsilon is the equivalent of having a no open hardware. And it is there mainly for two reasons which I will come to so you have the usual assignments. So, you have some simple variables notice the color coding we already have two languages right. You have the language of terms based on some signature sigma which is in violet in color. Then we have the language of predicates based on that is language of terms.

And those predicate names are all in green in color the operations are green in color right. And now we have we are defining a programming language on top of this these both except for 1 small difference. It does not you encompass the whole of first-order predicate logic. It uses only quantifier free predicates say essentially it uses only Boolean operations on the propositional operations on predicates. But the predicates are first-order predicates they have got parameters free variables and so on and so. So, you take the quantifier free fragment of first-order logic and you have the notion of variables. And you have the notion of an assignment this is. This just says that this term t is assigned to this variable x and you we have sequencing of programs. So, unlike

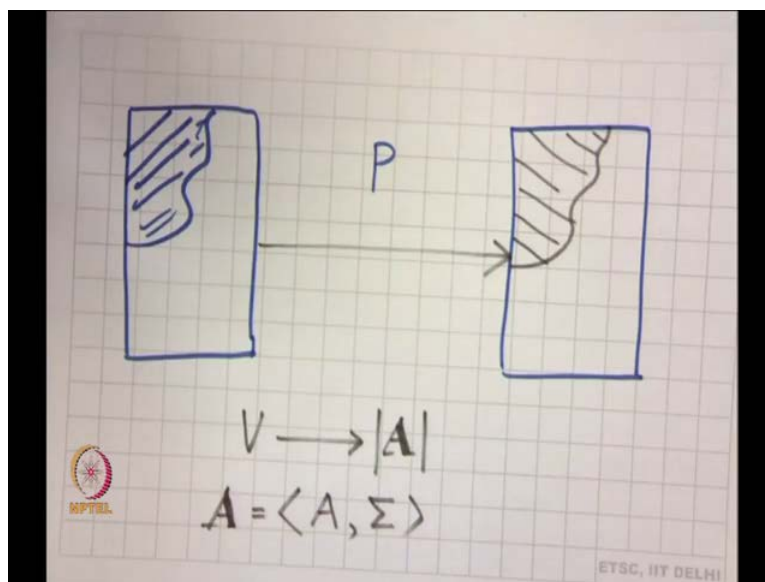
an actual programming language which has a programming program header and this and that and we have getting rid of all of those. And just having the most essential notation that is necessary to be able to compose programs. So, this semi-colon here is a composition operation i mean please do not mistake it to the further semi-colon in c where the semi-colon in c is a terminator of an instruction. So, you can think of that semi-colon as part of every instruction can be thought of as a regular expression of the form something dot star ending with a semi-colon. In the case of c whereas here the semi-colon is actually an operator it is a sequencing operator. The effects are not very different except that in the last command of a program you should not have a semi-colon this is exactly. So, this semi-colon is very much like the semi-colon in ml when you compose a sequence of let us say commands or functions. When you sequentially process some functions may be you keep let us say you are you give a sequence of print statements. Let us say in ml then they will all be separated by semi-colon but the last 1 will not have a semi-colon. Because the semi-colon is an operator which composes. It, is a composition operation equivalent to function composition. So, that is bracket square bracket is just a bracketing mechanism. Just in case i need to be able to group things in a certain order i want brackets. So, i have a pair of brackets this is a conditional very much like you see a formulas in spread sheet programs when you open ms excel or whatever. And you want to give a formula with and if then else then basically you have a Boolean condition chi which is a quantifier free formula. Which might be a quantifier free formula on the variables that you are interested in. And if it is true then this program p is executed and if it is false this q is executed.

So, this is essentially a basic if then else. And this the next 1 is just the while statement the indefinite looping that you have in imperative programs non-recursive indefinite looping. So this says that while chi is true essentially do p and this star essentially says that you keep doing it indefinitely simple programming language. This programming language is sufficiently powerful to encode any universal Turing machine it is I mean. So, for example we do not have function and so on and so forth we do not have things like for loops. Because, for loops can all be rendered as while loops we do not have pointers of course that is 1 big problem in terms of things. But otherwise its computing abilities are equivalent to any standard computing things may be less efficient or more cumbersome to do it this way then through other ways. But it does not reduce the power of computation in any way that is the important thing. So, this is a very simple programming language called the while programming language it is way. It is been very

popular in most retizes on in the beginning verification of programs. Since all most all programs can be expressed in while in fact for the purpose of provability it is better to translate all programs into while.

And then prove them rather than keep them in their native form for certain reasons. So, this epsilon no op actually is present in every programming language in some form. So, for example in c if you just put an open brace close braces that is your epsilon in any assembly programming language there is a no op. And in particular if I want just the if then construct then I use the if then else with q being epsilon right. And, so I can and then of course I can compose things but the more interesting thing about epsilon algebraically speaking is that epsilon is a identity element for composition. So, epsilon semi-colon p is the same as p that is a intension p semicolon epsilon is the same as p. Even if p is not dominating i mean now with the concept of a programming language. There is an issue of termination which we have to address at some point. But so p semicolon epsilon is the same as p and so on and so. So, essentially this set of programs defined by this programming language ah forms a monoid under semicolon with epsilon as the identity element. So, it has a nice neat structure. So, when we look at a program programming language and when we look at programs we think of them as essentially transformers of state a state transformer.

(Refer Slide Time: 11:03)



So, what happens is you essentially have some memory. You have a memory bank of which you are using let us say some part. And what you do through a program is there is an initial state of this memory by program of course we do not mean an interactive program. We are talking about programs which have just we start from some initial state. And if they terminate they terminate in some finite state. And state is essentially the memory map that you are using for all practical purposes we can assume that this memory is infinite. And what this program does is it essentially transforms the memory map. In the case of this while programming language it does not use any extra memory there are no declarations no ways of getting new variables program variables. And so on and so essentially what happens is that the picture of memory remains more or less the same except. That the values told in those locations are now probably different. So, a program is essentially a state transformer so it has a fix set of variables. And we will identify those variables with the variables of logic. Actually what happens is that those variables in a memory are really abbreviations or names of memory locations. However we will not take the notion of a memory location very seriously we will just take it as. So, therefore the notion of a state of a program as the contents of the memory. That is being used by the program is now identified with just a valuation. So, actually the state of a program in an actual machine refers to transforming memory maps even for a small while programming language like this. If you implement it will actually transform memory maps but we are not bringing in the concept of memory. We are just having we are just looking at it as valuation the memory map itself as a valuation of variable. And essentially as valuation which takes valuation in the same sense in which we define the semantics of first order logic. The semantics of terms in first order logic we had a valuation which was essentially the set of variables arrow some values in the domain.

So, it is essentially variable to value mapping so you have a domain bold \mathcal{A} and essentially take the carrier of this bold \mathcal{A} . So, your bold \mathcal{A} is essentially something of this form there is a carrier \mathcal{A} and based on σ on your signature σ . So, at this moment σ is just some parameter so it is given an arbitrary signature of terms. I can define a while programming language parameterized on that signature. I mean in fact that is what actually what happens all your higher level software you take mat lab for example. The basic construct of mat lab or all like imperative constructs assignments this that. You take object oriented programming your basic assignment looping sequencing. And conditional they are all there the signature is expanded it is not just whatever signature is there in the bare machine has a signature basically whatever operations.

The machine allows let us say arithmetic operations. And so on and so forth integer arithmetic operations real operations floating point operations whatever that is the signature of the bare machine itself. But when you have something like mat lab. You actually have you can deal with entire you can deal with also structures mat lab maple automath and so on so. You can deal with entire structures and given any complicated structures we can define a while programming language on top of that. That sense it is absolutely general purpose and very highly polymorphic on the signature. So, program is merely a state transformer and so of course you defined a language. Then we have to define its semantics.

(Refer Slide Time: 16:21)

The Semantics of WHILE

Let \mathbf{A} be a Σ -algebra. Let $\mathbf{V}_{\mathbf{A}} = \{v_{\mathbf{A}} \mid v_{\mathbf{A}} : V \rightarrow |\mathbf{A}|\}$ be the set of all valuations (also called *states*). The meaning of a program P is given by $\mathcal{M}_{\mathbf{A}}[[P]] : \mathbf{V}_{\mathbf{A}} \rightarrow \mathbf{V}_{\mathbf{A}}$

$$\begin{aligned} \mathcal{M}_{\mathbf{A}}[[\epsilon]]_{v_{\mathbf{A}}} &\stackrel{df}{=} v_{\mathbf{A}} \\ \mathcal{M}_{\mathbf{A}}[[x := t]]_{v_{\mathbf{A}}} &\stackrel{df}{=} v_{\mathbf{A}}[x := \mathcal{V}_{\mathbf{A}}[[t]]_{v_{\mathbf{A}}}] \\ \mathcal{M}_{\mathbf{A}}[[P]]_{v_{\mathbf{A}}} &\stackrel{df}{=} \mathcal{M}_{\mathbf{A}}[[P]]_{v_{\mathbf{A}}} \\ \mathcal{M}_{\mathbf{A}}[[P; Q]]_{v_{\mathbf{A}}} &\stackrel{df}{=} (\mathcal{M}_{\mathbf{A}}[[Q]] \circ \mathcal{M}_{\mathbf{A}}[[P]])_{v_{\mathbf{A}}} \\ \mathcal{M}_{\mathbf{A}}[[\chi ? P : Q]]_{v_{\mathbf{A}}} &\stackrel{df}{=} \begin{cases} \mathcal{M}_{\mathbf{A}}[[P]]_{v_{\mathbf{A}}} & \text{if } \mathcal{T}_{\mathbf{A}}[[\chi]]_{v_{\mathbf{A}}} = 1 \\ \mathcal{M}_{\mathbf{A}}[[Q]]_{v_{\mathbf{A}}} & \text{if } \mathcal{T}_{\mathbf{A}}[[\chi]]_{v_{\mathbf{A}}} = 0 \end{cases} \\ \mathcal{M}_{\mathbf{A}}[[\chi ? P]^*]_{v_{\mathbf{A}}} &\stackrel{df}{=} \begin{cases} \mathcal{M}_{\mathbf{A}}[[P; \chi ? P]^*]_{v_{\mathbf{A}}} & \text{if } \mathcal{T}_{\mathbf{A}}[[\chi]]_{v_{\mathbf{A}}} = 1 \\ v_{\mathbf{A}} & \text{if } \mathcal{T}_{\mathbf{A}}[[\chi]]_{v_{\mathbf{A}}} = 0 \end{cases} \end{aligned}$$

So, for all practical purposes this is the semantics

Student: why does the which language why does it happen

Well I chose a star because you might have 0 iterations

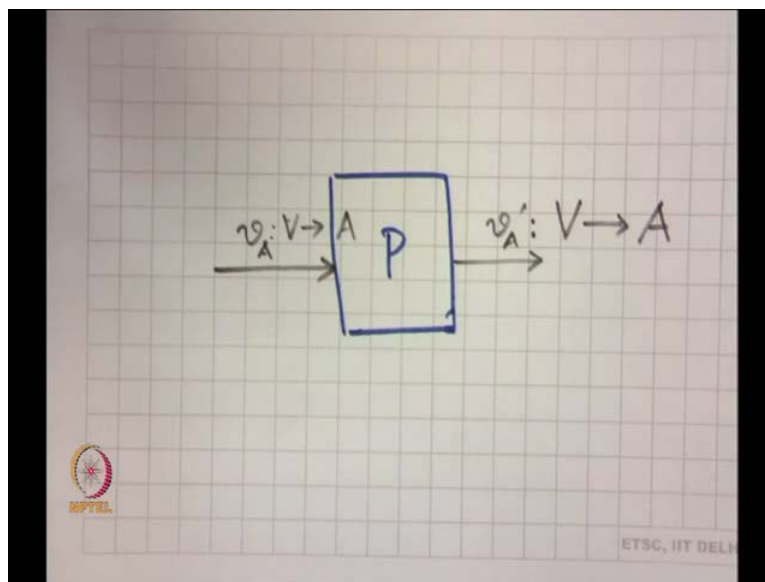
Student: In terms if it is a 0 iterations is in

But that depends on the state right whether it's going to be 0 iterations or not it has depends on the evaluation of chi

Student: So if chi is valuated

So, actually it is you takes any while loop if the body executes k times. Then the condition is evaluated k plus 1 time. That is very clear so there is a k plus one'th checking of the while loop before you exit the while loop. So, that is why i used a star but that is only notation everything depends on the semantics. So, here is the semantics we have to define a semantics in the case of this simple language. We define essentially what is known as a din a additional semantics functional. It is a completely defined functions rather than what we normally do in a course on programming languages which is you define the low level operations. And then a big steps a small steps semantics and then a big steps semantics and so on and so forth. Here I am taking a more simplified view I am looking at all these as a pure state transformer. Think of it as a, black box state transformer.

(Refer Slide Time: 18:30)

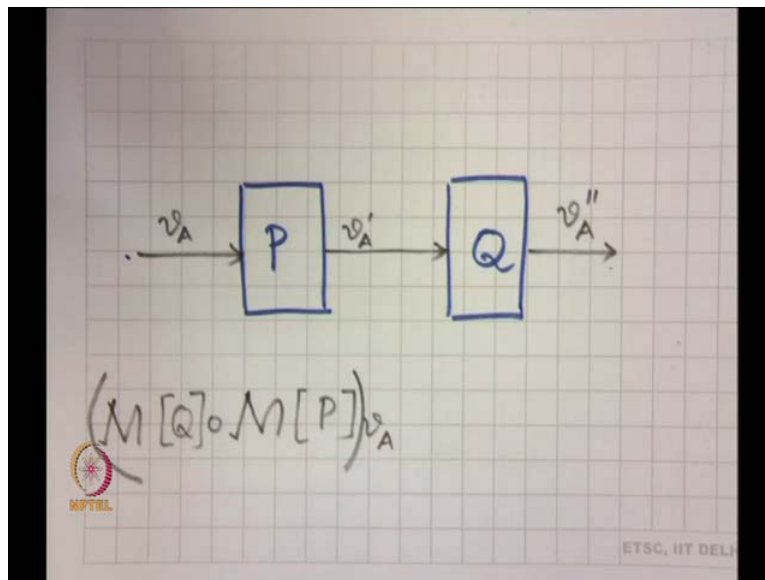


So, you actually are taking this program think of this program now as there is this program it is a black box. And we have we are not dealing with the memory any more the semantics of this language is defined as essentially li start with a valuation. And that valuation V_A is the input and what it produces is a new valuation V_A' . And each of these is the function from the set of variables to the carrier of the domain in which you are looking at. So, this is what a completely black box picture. So, it is the a biggest step semantics based on just structural induction. So, there is this notion of meaning so a program is a state transformer a program of course is just a piece syntax. So, the meaning under some domain A of this program P is something that

transforms VA to some VA prime where VA and VA prime. Both belong to this bold VA which is the set of all possible valuations of variables. So, these valuation are also called states in the case of an imperative programming language and that is you have states. And of course everything that satisfies every element of this is a program right every full element of this structure is a program. So, you can compose you can add programs can catinate programs and so on and so forth. You can do all that and get free its closed under these composition operation for example.

Which is not true of most actual programming languages because they have a program header. And this and some terminator and so on and so forth so you cannot just take 1 program. And can catinate another program blindly to it you have to do some other work also for this. So, the empty program just leaves the state unchanged the assignment it creates a new state in which. The news valuation is exactly the same as a old valuation except possibly for the values given to x. And the value that is being given to x is that you evaluate the term t under the old valuation VA. Whatever value you get from that you give that to x remember that this is VA. All these other terms which I am not explaining are exactly as we have d1 before in the semantics of first-order logic. So, this VA is something that comes from the time then we defined the semantics of terms. In the case of a bracketed program if you add an extra pair of brackets you are not doing anything. This meaning of this program is exactly the meaning of the program without brackets. So, that finishes the some of the simplest constructs and now you have this semicolon as I said. Semicolon essentially means that.

(Refer Slide Time: 22:30)

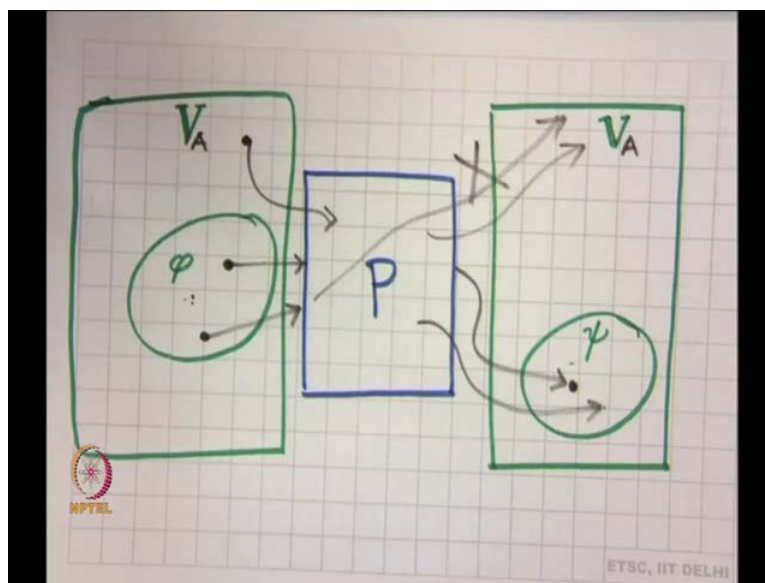


So, what we are saying now is that you take two programs p and q since it is closed under composition. What we are saying is you take this you give a valuation VA here you think of this valuation itself as an input. To the program after programs work on memory right so they work on valuations. And this p will give some will result in a transformed valuation VA prime which is the input to q and which gives me a new valuation VA double. So, this is essential so this is completely a functional style of defining. So, essentially if i take the meaning of p and evaluate it in A_i get VA prime. And essentially I am going to take the meaning of q and compose it with this and evaluate them both under VA . So, this semicolon is exactly function composition which is which is fair enough this is what I say. So, it is now as far as the conditional is concerned it is very simple this conditional is a quantifier free formula. And of course your valuation deals with all the variables. So, it is possible to evaluate this condition χ for truth under the valuation VA . And if it is true then the new valuation that you get is whatever you get by executing p on VA . And if it is false under VA then whatever the new valuation that you get is whatever you get. By executing q that is the semantics of if then else.

As far as the while loop is concerned it is very simple you first evaluate this χ if it is false under the valuation VA . Then of course it is exactly as you said it just returns back the same valuation. So, the result is the same valuation VA if it is true. Then this while essentially acts like the replication operation in the lambda calculus right like the y combinatory. So, what does it do

it spawns a new copy of the body of it of itself. And the body is executed in composition with that. So on demand you spawn a it is like spawning a new recursion while and recursion are essentially the same. So, it spawns a new version of itself and executes that after having executed body. So, that is so this while is not entirely structurally inductive but i will not get into it. There are structurally inductive ways of it is specifying the fixed point. And so on and so forth but we will not get into it but we have essentially given a black box view. So, essentially what happens is that it spawns the copy of itself and attaches to the body at the end. So, the body is executed and again it goes into execute the while again. So, this is a recursive definition so and this is an intuitive semantics of the while programming language. However the problem now is you can take point now is where does logic come in and where does verification come in. Here we said that a program is essentially a state transforming given a certain VA it gives you a certain VA prime. There is also another way of looking at it you can think of a program also as a property transform.

(Refer Slide Time: 27:14)



So, what i can say is i have a program p and basically i am not interested in any particular valuation. And how it fix it suppose deterministic this language is deterministic and it is there therefore it is possible to give a functional semantics for it. But it is not necessary to think of it as a state transformer because most often we are not interested in the individual states. In the individual valuations we are not interested in all possible valuations each comp1nt of the

valuation. You are not interested in the value of each variable for example what you are usually interested is that I have a whole collection of memory maps. Think of this green box as essentially the set of all possible valuations. So, essentially think of it as this bold VA so it is the set of all possible valuations of which there is some subset of this valuations. Which satisfies some property ϕ then what we are saying is there is in the what the program does is that. It actually gives me a new valuation satisfying probably some slightly different property or it is not slightly satisfying a different property. So, there is some subset of this VA which satisfies some predicate ψ . And essentially what I am saying is that this program as a state transformer what it does is it takes. Let us say an element starting from ϕ as input and as output it gives me an elements satisfying ψ . So, rather than worry about individual valuations and individual variables and valuations we are looking at properties. That is the fundamental basis on which every all program verification based on first-order logic is defined. So, we are so only worried about properties we are not worried about the values of individual variables. We are only we might be basically what most of the time what are we doing in a program. If you think of you remove io from the program and make every program clearly functional single inputs single output. Then you have some let us say it reads a input from some area of memory let us say all. The variables that required to have a certain values then the variables have bear certain interrelationships of which. You are interested in only a small set of those interrelationships and that set of interrelationships that you are interested in is defined by this ϕ . And what you are interested in achieving is some other interrelationships between those variables. And that is what you specify by ψ you are not actually interested in tracking individual variables. Except when something goes wrong with your program.

So, the values of other variables which ϕ and ψ may not mention at all or have of no interest. As long as the program does what it is intended to do. And what you want to show in program verification is given a specification of properties ϕ and ψ . And given a program p which claims to do that you want to be able to verify or you want to be able to prove. That the program p starting from some state which satisfies ϕ terminates in a state that satisfies ψ . And is always guaranteed to do that. So, for example it should not happen that you start from some state inside ϕ and go often to some other place. That should not happen I do not have so that is a no the other thing is what you are saying is that. If it starts from some state outside ϕ then really it does not matter where it goes. So, if you give this as input and it goes off somewhere here or it

goes off into this also it does not prove anything. But something like this going off like that is a no. What you want to show is that given that there are no bets outside phi I do not care what happens. If the input is from outside phi but what I am saying is if the input is from phi I want a guarantee that it ends up if at all in psi. So, in that sense programmers are not just state transformers we can elevate them to property transformers. So, the properties of a valuation are changed by changing the valuation appropriate.

(Referred Time: 34:27)

Programs As Predicate Transformers

We may also view a program as a transformer of properties.

Definition 37.1 A partial correctness assertion (also called a Hoare-triple) is a triple of the form $\{\phi\} P \{\psi\}$ where ϕ is a formula called the precondition, P is a program and ψ is the postcondition.

Definition 37.2

- $\{\phi\} P \{\psi\}$ holds in a state v_A (denoted $(A, v_A) \models \{\phi\} P \{\psi\}$) if
 1. $(A, v_A) \models \phi$ and
 2. $M_A[P]v_A = v'_A$ implies $(A, v'_A) \models \psi$
- $\{\phi\} P \{\psi\}$ is valid in A , (denoted $A \models \{\phi\} P \{\psi\}$) if $(A, v_A) \models \{\phi\} P \{\psi\}$ for every state v_A .

NPTTEL

So, these are we are they can also be called predicate transformers. So, this phi so a what is known as a partial correctness assertion. Essentially consist of this phi which is called a precondition and this psi which is called a post condition. And what you are trying to say is that this program p satisfies this precondition in this post condition. And you write this as a triple in this form and this triple is called a partial correctness assertion. So, what we are saying is that in any state VA this triple holds if the initial valuation VA actually satisfies this phi. So, that you are starting with that assumption that i am restricting myself attention to this blob phi. And now the program of course has a semantics which transform states into some new valuation VA prime. And what you are saying is that if it does give me a new valuation VA prime. Then that a VA prime should satisfy psi notice that this is implies in the sense. That it does not say that this thing could necessarily have to go to a final state. The program may not terminate so here, essentially what you are saying is if the program execution does terminate. And it gives me a

new state VA prime then that VA prime must satisfy ψ . That is what this is about then we would say that this $\phi \rightarrow \psi$ is a valid formula in a . And this is denoted by \models we use a usual validity notion. Where extended it to programming language with partial correctness if this holds for every initial state VA . So, the first thing actually this first bullet actually tells you about satisfiability of this triple form of this hoare-triple. What is known as a hoare-triple after Tony Hoare who actually gave us this. So, and the second one talks about validity and actually we are most often interested in validity. Notice that we are not interested in validity in a domain independent fashion. We are interested in validity for particular domains a which are particular sigma algebra. So, unlike logical validity I have not gone to that logical validity which would say that for all sigma algebra's a . This should be valid no we are not saying that and that is the we are restricting ourselves to validity in a given domain. Because normally when you talk about a programming language the given domain is actually, whatever the underlying machine can represent as a model of let us numbers for example. I mean your machines are not going to represent non-standard models of the naturals. I mean they only represent one particular model of the naturals one particular model of the integers. And so you are interested in validity only within that model. So, that is what so now programs therefore our predicate transformers. And this is the concept that was probably hardest for people to understand in first year in my programming courses. But the important thing here is that this only defines partial correctness as I said this one says that. If the meaning of p under VA is a valuation VA prime then VA prime should satisfy ψ . It does not say necessarily the meaning of p under VA should be some valuation. And one of the important things that they are interested in programs is that they terminate.

(Refer Slide Time: 38:48)

The slide is titled "Total Correctness of Programs" in blue text. It contains two definitions. Definition 37.3 states that a total correctness assertion is a triple $[\phi] P [\psi]$ where ϕ is a precondition, P is a program, and ψ is a postcondition. Definition 37.4 defines when this triple holds in a state v_A and when it is valid in an algebra A . The slide also features an NPTEL logo in the bottom left corner and a navigation bar at the bottom.

Total Correctness of Programs

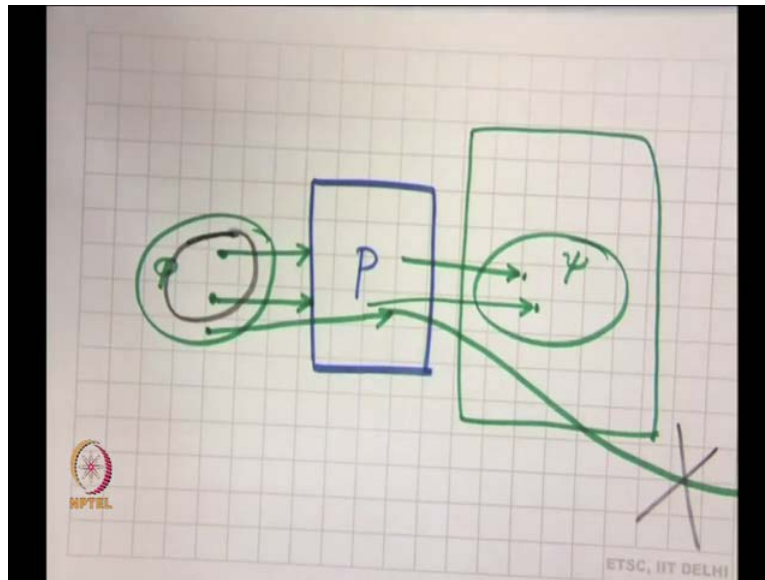
Definition 37.3 A total correctness assertion is a triple of the form $[\phi] P [\psi]$ where ϕ is a formula called the precondition, P is a program and ψ is the postcondition.

Definition 37.4

- $[\phi] P [\psi]$ holds in a state v_A (denoted $(A, v_A) \models [\phi] P [\psi]$) if $(A, v_A) \models \phi$ implies for some v'_A $M_A \llbracket P \rrbracket_{v_A} = v'_A$ and $(A, v'_A) \models \psi$.
- $[\phi] P [\psi]$ is valid in A , (denoted $A \models [\phi] P [\psi]$) if $(A, v_A) \models [\phi] P [\psi]$ for every state v_A .

And that is the notion of a total correctness. So, will use this notion of total correctness and moreover total correctness also depends intrinsically on the underlying algebra. You have to come up with well-orderings and so on and so forth. In an algebra, you have to define a measure of descent on a well-ordering on a well-ordered path with a finite. Which means it has to there should not be an infinite chain it should be a bounded chain. So, this algebra validity is always with respect to a given algebra algebra. With a given carrier set so a total correctness assertion is a triple with by the way these are black square brackets. They different from the blue square brackets which are part of the programming language. So, ϕ P ψ here again ϕ is a precondition ψ is the post condition but here what you are saying is that unlike in the previous definition. Here you are saying that firstly this ϕ should hold in the initial valuation v_A . And the very fact that ϕ holds in the initial valuation v_A must imply that the program does terminate from that valuation. So, the difference between partial correctness and total correctness is really that.

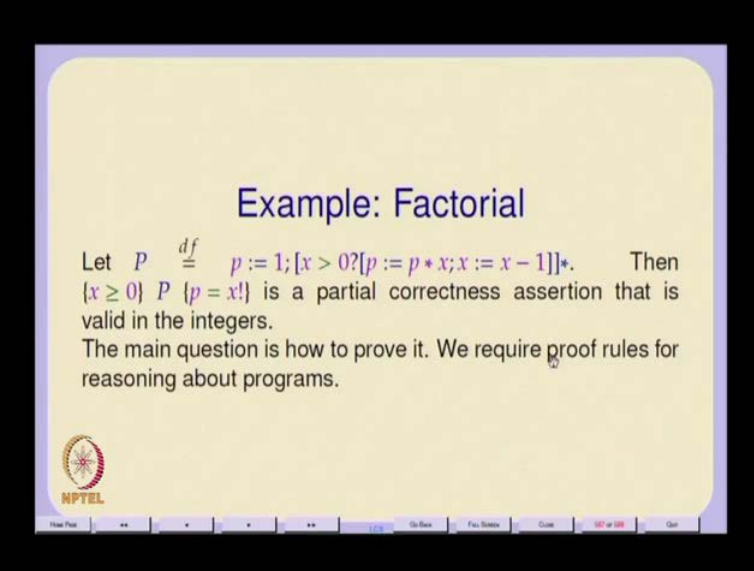
(Refer Slide Time:41:13)



Take these two scenarios so I have the program p and I have essentially 1 blob ϕ here and another blob ψ here. And in the case of partial correctness for which I will use blue for which I will green. What we are saying is that given a valuation here it can go into a valuation here or given a valuation here it can go should not terminate. So, if this is my set of all valuations it really goes outside the set of the valuations. Because it is like an undefined state which is outside the valuation. So, this is perfectly valid for partial correctness for total correctness. What you are saying is that this is not acceptable. What you are saying is that for every such thing it should actually go into this ψ somewhere. So, in general what you are saying therefore is that when you look at total correctness. There will be only subset of these ϕ 's which guaranteed this, property. And essentially what you are saying is that there outside the subset but still satisfying ϕ what you can have is this. So, your notion of ϕ therefore is circumscribed by whether you are interested in partial correctness or total correctness. Notion of validity again is exactly the same it just that this definition has two parts. Where there is the and is here and they implies is here and this definition has implies first and then an and. The meanings are of course different. So, this is what we have is the main notions of specifications so in a specification ideally from now on. When you are asked to write a program we should just be able to give a ϕ and ψ in first-order logic. And you should write the program and you should be able to prove that your

program without executing it. You should be able to prove that phi that the programs satisfies a specification phi psi were phi is a precondition and psi is a post condition.


(Refer Slide Time: 44:33)



Example: Factorial

Let $P \stackrel{df}{=} p := 1; [x > 0? [p := p * x; x := x - 1]]^*$. Then $\{x \geq 0\} P \{p = x!\}$ is a partial correctness assertion that is valid in the integers.

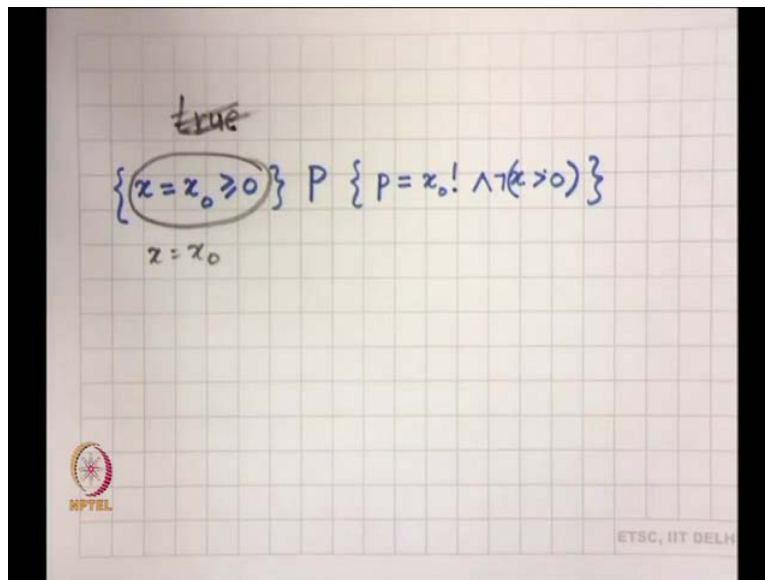
The main question is how to prove it. We require proof rules for reasoning about programs.

 NPTEL

Slide navigation controls: Home, First, Previous, Next, Last, Refresh, Close, 107 of 108, Quit

So, here let us take a we have to I mean all programming begins with factorial. So, let us start with factorial so what we are saying is this is an simple imperative program. And of course there is no io there are no program address nothing. So, there are two variables in this program x and p and basically think of this program. Where something that reads in the value of x some from somewhere or it has the value of xp specified. So, and then what happens at the actual program initializes p to 1 and if x is greater than 0 then it executes. Then this is a while loop this whole thing is a while loop what does it do it p is assigned p star x and x is decremented by 1. So, will assume that all these operations are available and so on and so forth all these relations are available in the underlying domain. So, your signature includes all these things then what we are saying remember here. Then this is a partial correctness assertion says that if x is originally greater than or equal to 0 then p is going to be equal to factorial of x actually. What I should change this unfortunately in the case of programming. Since valuations change variable values get updated so we should actually state this partial correctness assertion in this fashion.

(Refer Slide Time: 46:21)



There is an initial value of x so x equals x not greater than or equal to 0 p . And of course we have I am assuming integers there is an implicit assumption in there p equals x not factorial and x is. And not x is greater than 0 this should be the correct partial correctness assertion. so ignore this for the moment. So, this is a partial correctness assertion which this program satisfies notice it is a partial correctness assertion because if x is negative. The program does not terminate

Student: (Refer Time: 47:40) precondition does not

Then the precondition does not hold

Student: (Refer Time: 47:45) integer total correctness sir it is total

It is total it is total because if the whenever the precondition holds it actually terminates. Let us write a partial correctness assertion where supposing we change this to true what happens

Student: (Refer Time: 48:16) we change it to but x naught but not greater than 0

Will just change it to x equals x not but not necessarily greater than or equal to 0 then you get a partial correctness assertion. So, of course the main question now is how to prove this

Student: (Refer Time:48:42) Sir but, it is not a partial correctness it is then it will be p equals to 1 because if x is less than 0 then p will be assigned to 1

Student: further condition x is greater than 0 is possible term.

So, it will it actually terminate it will always terminate so this program always terminates that is a problem so i should not a i should have actually thought of something else. Anyway at the moment let us just look at this as a partial correctness assertion. But the question is of how to prove it and you have this language. And therefore now what we need to give a proof rules for this programming language.

(Refer Slide Time: 49:30)

Proof Rules for Partial Correctness

$\epsilon. \frac{}{\{\phi\} \epsilon \{\phi\}}$	$:=. \frac{}{\{\phi\} x := t \{\psi\}} \quad (\phi \equiv \{t/x\}\psi)$	$\square. \frac{\{\phi\} P \{\phi\}}{\{\phi\} [P] \{\phi\}}$
$\therefore. \frac{\{\phi\} P \{\xi\} \quad \{\xi\} Q \{\psi\}}{\{\phi\} P; Q \{\psi\}}$	$? \therefore. \frac{\{\phi \wedge x\} P \{\psi\} \quad \{\phi \wedge \neg x\} Q \{\psi\}}{\{\phi\} x?P: Q \{\psi\}}$	$?*. \frac{\{\phi \wedge x\} P \{\phi\}}{\{\phi \wedge x\} P \{\phi \wedge \neg x\}}$

$$\Rightarrow \frac{\phi \Rightarrow \phi' \quad \{\phi'\} P \{\psi'\} \quad \psi' \Rightarrow \psi}{\{\phi\} P \{\psi\}}$$

So, here are the rules for partial correctness. So, these set of rules for partial correctness is approximately structurally inductive. Basically there is 1 rule for each construct of the programming language. So, the empty program basically does not change any state so any property that it holds initially as a precondition also holds finally. So, that is what you said this actually does not transform the predicate ϕ at all. Next we come to this assignment and here is so what we are specifying here is that supposing there is some post condition ψ which needs to be satisfied. Then an appropriate precondition is 1 that is obtained syntactically from ψ by replacing all free occurrences of x by the term t . So, this ϕ and ψ are full first-order predicates they could have quantifiers in them they need all and they could have free variables also. In particular the program variables will be free variables of valier assertions. There might be other bound variables but they will not necessarily be part of the program in particular. What happens

is you might want to make an assertion supposing. You take the prime number generation program at a certain stage let us say after I iterations you have found the first i primes. Then here i is a counting variable of the program. Which let say keeps track of the number of primes that have been generated. But you will have a bound variable for your assertion which says for all j less than i p_i as been p_j has been generated. So, there will be bound variable so there will be quantifiers also which essentially says that I have computed. All the first i primes and that statement requires a universal quantifier.

And it requires a bound variable for the universal quantifier but that bound variable will not be a program variable. So, the free variables are what will be program variables. So, this I say so therefore if ϕ actually has x as a free variable of course it is quite possible that ϕ does not talk about x at all. In which case let us say ψ does not talk about x at all. Then of course the result of the substitution then x will not be a free variable of ψ and the result of the substitution will just leave it as ψ . But if ψ does have x has a free variable then what we are saying now is before the assignment was executed. If I replace all those free occurrences of x by their term t . Then in that current valuation in that of the precondition in that state ϕ should that ψ with t . For x should be should have been true only if that is true can I claim after the assignment that ψ would be true.

Student: (Refer Time: 53:58)