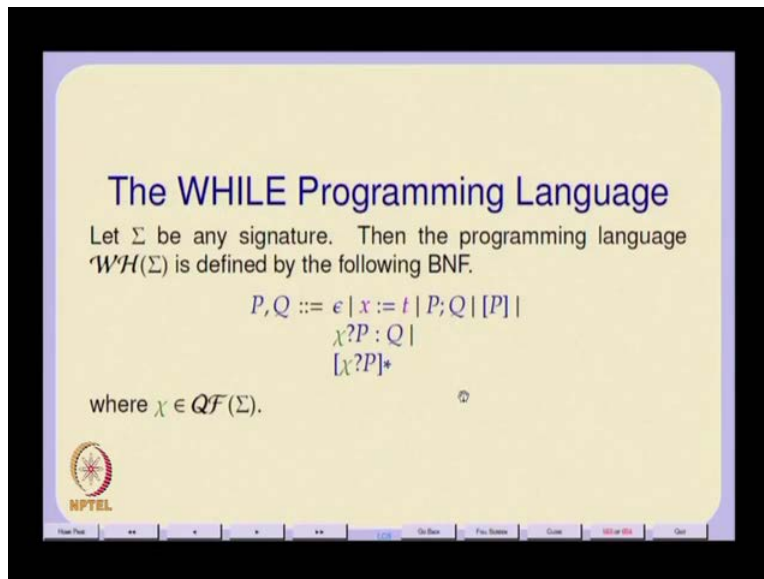


Logic for CS
Prof. Dr. S. Arun Kumar
Department of Computer Science
Indian Institute of Technology, Delhi

Lecture - 38
Verification of WHILE Programs

You are looking at Program Verification and we had the small language of WHILE Programs which, is what we wanted to verify.

(Refer Slide Time: 00:46)



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 $\chi \in QF(\Sigma)$.

The slide includes an NPTEL logo in the bottom left corner and a navigation bar at the bottom with icons for Home, First, Previous, Next, Last, and other slide controls.

So, what I did was I will look at this While Programming Language. I defined a small syntax so the moment you define syntax you have to define semantics. And, this consist a basically an identity function and assignment sequential composition a conditional and looping construct basically and unbounded looping construct. So, all bounded looping constructs can be obtained from the unbounded looping construct.

(Refer Slide Time: 01:24)

The Semantics of WHILE

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

$$M_A[[\epsilon]]_{v_A} \stackrel{df}{=} v_A$$

$$M_A[[x := t]]_{v_A} \stackrel{df}{=} v_A[x := \mathcal{V}_A[[t]]_{v_A}]$$

$$M_A[[P; Q]]_{v_A} \stackrel{df}{=} M_A[[P]]_{v_A} \circ M_A[[Q]]_{v_A}$$

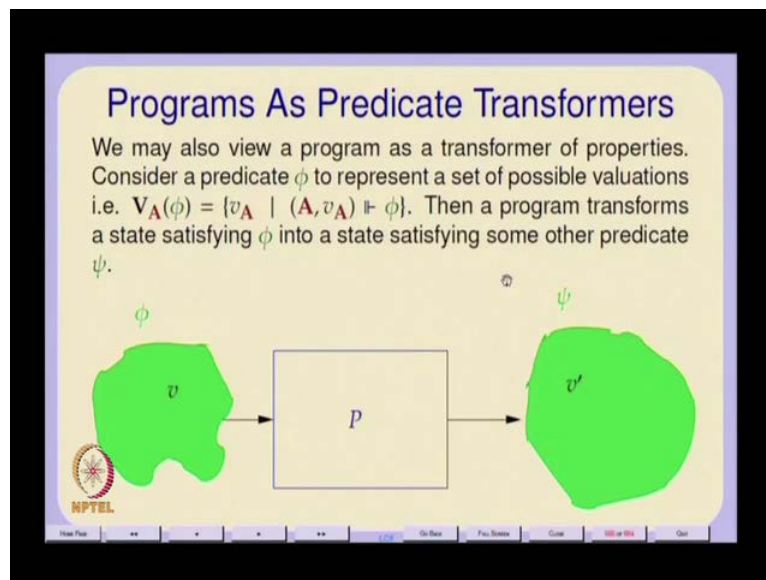
$$M_A[[\chi?P; Q]]_{v_A} \stackrel{df}{=} \begin{cases} M_A[[P]]_{v_A} & \text{if } \mathcal{T}_A[[\chi]]_{v_A} = 1 \\ M_A[[Q]]_{v_A} & \text{if } \mathcal{T}_A[[\chi]]_{v_A} = 0 \end{cases}$$

$$M_A[[\chi?P]^*]_{v_A} \stackrel{df}{=} \begin{cases} M_A[[P; \chi?P]^*]_{v_A} & \text{if } \mathcal{T}_A[[\chi]]_{v_A} = 1 \\ v_A & \text{if } \mathcal{T}_A[[\chi]]_{v_A} = 0 \end{cases}$$

NPTEL

So, basically the semantics of any programming language is usually treated as a programmers as a sort of state transformer. Which, in our case in the context of first order logic works on to valuations for some valuations under some model. Usually, and for the purpose of verification what we are essentially looking at is the first order theory of that model. So, in general otherwise the programming language is actually independent of the model the constructs so the programming language or actually independent of the model. But, you could take any kind of model but normally of course we take any integers as let us say the model. I mean since, that is up to some abstraction that is what most machines actually implement. So, a simple general purpose programming language has this functional semantics.

(Refer Slide Time: 02:13)



Which, I did last time and then the other thing that I wanted to what in the context of first order logic. We can also view programs not just a state transformers but as Predicate Transformers. So, I something that takes an input property and gives you an output property so but, that essentially means that you are talking about it still going to act as a transformer of states from some valuation V to some valuation v prime. However, what we are talking about is whether the we are taking this valuation from an input property from essentially a subset of valuation which satisfies certain property ϕ . And, we are essentially saying that v prime would be belong to a subset which is characterized by the set of valuations which satisfy like say this output property of ψ .


(Refer Slide Time: 03:19)

Examples: Factorial 1

Let $\Sigma \supseteq \mathbb{Z} \cup \{! : s, +, -, * : s^2 \rightarrow s; =, > : s^2\}$.

Example 37.5 Let $P_1 \stackrel{df}{=} p := 1; [\neg(x = 0)?\{p := p * x; x := x - 1\}]^*$.
Let $Z = \langle \mathbb{Z}, \Sigma \rangle$. Then

1. $Z \models \{x = x_0\} P_1 \{p = x_0!\}$
2. However $Z \not\models \{x = x_0\} P_1 \{p = x_0!\}$ since P_1 will not terminate for negative values of x .


NPTEL

Navigation bar: Home, Back, Forward, Search, etc.

So, then we define the notion of partial correctness assertions and total correctness assertions. And, let me correct some mistakes I made last time. Let, us look at this example of Factorial. Let, us assume that our signature is some superset of these things I am including the factorial symbol also there because, I am going to use it in my verification. But, you can and the other thing is that I am taking the entire set of integers as a countably infinite set of constants. Constant symbols which, is usually the case except that we in an actual machine we are bounded by a finite only a finite subset of Z . But, for the purpose of verification let us assume this an infinite set of constants. However, the set of terms that you get is still only countably infinite. Even, if you take this entire set Z as a set of constants as part of the signature. So, the model so let us as we write more and more complicated programs are signature will actually increase. So, for the purpose of this factorial program it is enough to have this much so that is why I have written the signature sigma is a superset of this. So, it contains at least these operations let us take this first program P_1 . P_1 actually has this condition x is not equal to 0.

And, under this condition x equals x_0 P_1 p equals x_0 actually this should be x equals x_0 greater than or equal to 0. Then, p would be the factorial of x_0 prime. But, this partial correctness assertion is correct in the sense that if x_0 would negative then, this program would not hold. Because, this program checks exactly for x being equal to 0. So, if x is negative it is still enter the loop and it will keep decrementing it will keep multiplying something to p and so on so forth.

But, basically it is never going to terminate. So, in that sense this partial correctness assertion actually specifies the partial correctness of this program P1.


So, this total correctness assertion however is not valid because of the fact that there is no guaranty of termination. And, this program might actually not terminate so even if you so if x naught is negative then of course the program is not going to terminate. So, with this total correctness assertion would not be valid so, that is the first thing.

(Refer Slide Time: 06:18)

Examples: Factorial 2

Example 37.6 Let $P_2 \stackrel{df}{=} p := 1; [x > 0? [p := p * x; x := x - 1]]^*$.
 Let $Z = \langle \mathbb{Z}, \Sigma \rangle$. Then

1. $Z \models [x = x_0 \geq 0] P [p = x_0!]$
2. $Z \models [x = x_0 \geq 0] P [p = x_0!]$
3. A more "technically complete" specification is
 $Z \models [x = x_0] P [(x_0 \geq 0 \rightarrow p = x_0!) \wedge (x_0 < 0 \rightarrow p = 1)]$

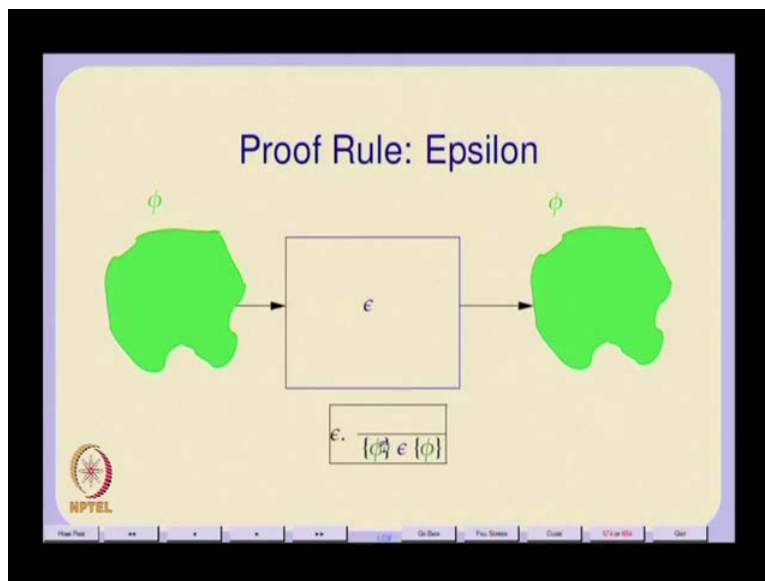
 NPTEL

Home Next Previous Go Back Find Screen Close Slide Show Quit

The second thing was actually what I gave yesterday as the program last time. So, here I actually have the assertion x greater than 0 as a condition. And, then both the same thing specification holds as both partial correctness and total correctness because, this program is guaranteed to terminate under the integers always I mean. So, that certain difference between partial and total correctness is something that has to be maintained. However, this specification is not technically complete in a certain sense. Which is that it only gives you half the story the other half of the story when x naught is negative is something that it does not spare say. Thus, notion of technical completeness is the I have put it within the double codes. But, you should actually look at my the slides of my programming notes where I have defined it more formally in the context of pipe systems and so on and so forth.

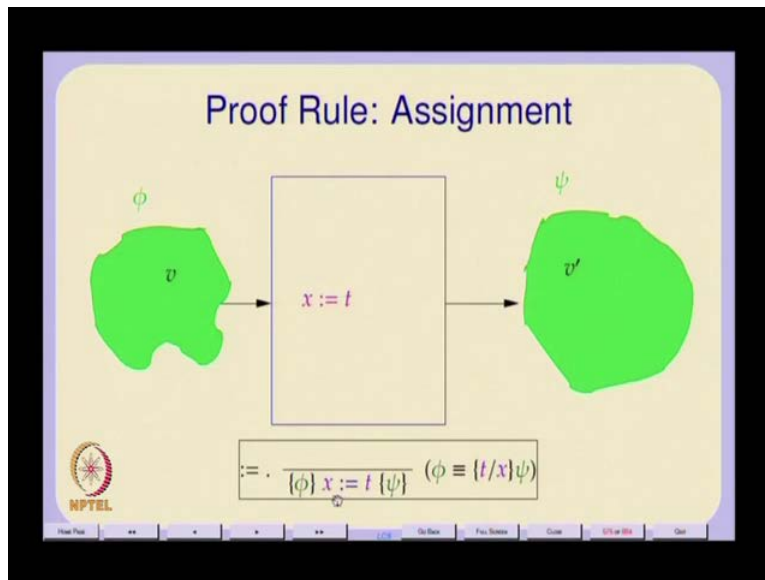
But, so normally what we expect is the technically complete specification. So, in that sense this specification here are slightly weaker than what might be called a technically complete specification. So, the basic idea here is that every program is correct it only depends upon what the specification is so this program is technique. I mean so, that technically complete specification under the integers essentially would specify what happens when x naught is both negative and non negative. And, you actually get this interesting case that when x naught is negative then you get your product to be 1. You get this so it is not really factorial I mean and we are not programming the gamma function. So, this program is not really the factorial program in a strict sense because of the fact that for negative integers it gives a value 1. So, but then but however I said every program is correct it only depends what your specification is. So, if you give this as your specification as a then that is it take sort of technically more complete specification under the integers.

(Refer Slide Time: 09:12)



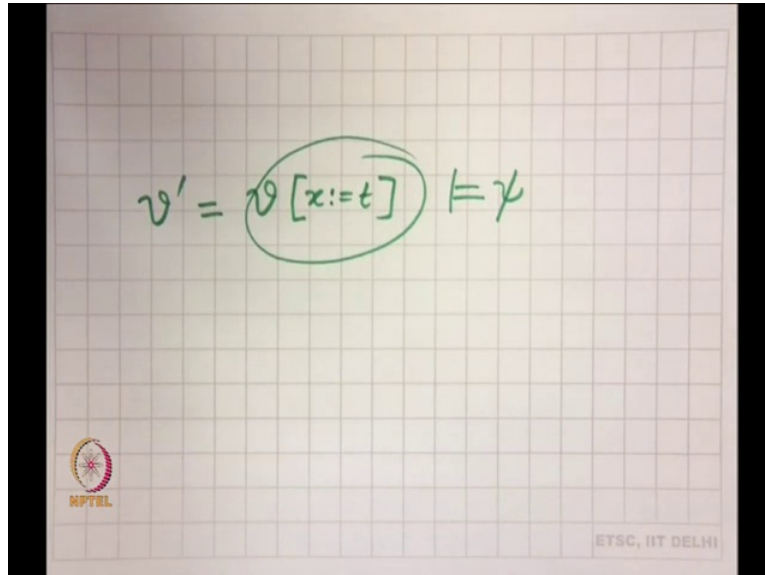
So, we were looking at Proof Rules. So, here is a first Proof Rule Epsilon is basically as it is a identity function. So, as a predicate transformer it leaves a input predicate unchanged as output so that is why you have phi epsilon phi.

(Refer Slide Time: 09:30)



And, I am only giving partial correctness proof rules here initially.

(Refer Slide Time: 09:43)

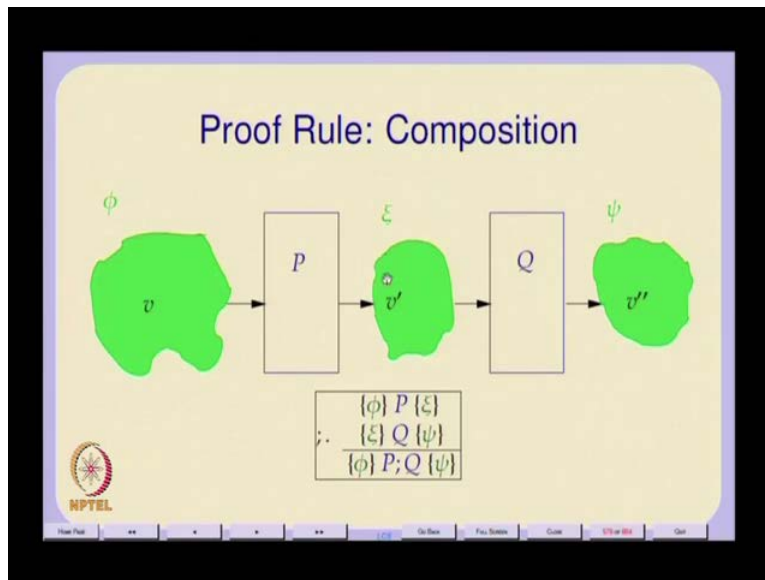


So, the assignment as I said this v prime is actually a variant an x variant of v . In which, x gets the value t . So, now this would satisfy let us say supposing this belongs to a set of valuations which satisfies the predicates ψ . Then, the only alternative the only input possible predicates are satisfiable that can act as specification for this assignment are those in which if you had

syntactically replaces x by the term t . Then, you get a predicate that is true I mean so that subset in which x is actually replaced by t by substitution. So, that is here is where the main difference comes between this while programming language as a theoretical programming language as opposed to while programming language which is going to be implemented on a machine. Where, x is going to be the name of a memory location rather than the name of a term. I mean so the differences between l value and r value then become prominent. So, what we are actually with the abstraction we are doing is that we are ignoring the fact that these variables that we are talking about in actual implementations are actually names of memory locations. And, therefore they are containers they are not so they are not associations in a functional sense. They, are not just abbreviations for terms they are not just names for terms they are actually containers therefore, the notion of an l value and r value is something that they are ignoring completely. So, that sense our model is a purely theoretical model and does not actually confirm to the notion of memory in a standard for no-mean architecture.

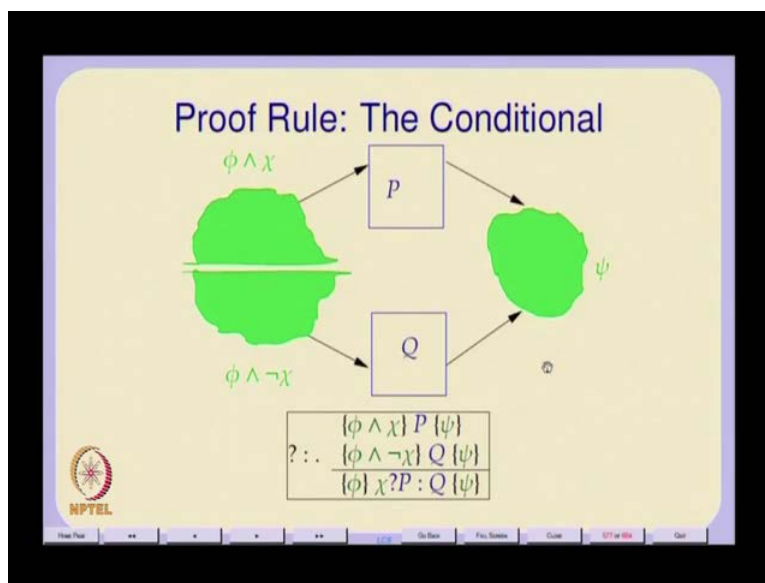
And, so you cannot extend this sort of you cannot extend this while programming language to one with pointers and expect to get proof rules all such things in just in exactly the same way. And, in fact that issue of dealing with heap structures and pointer evaluations in memory is still a subject of certain kind of research it is not easy to get elegant programming logics by which you can prove completely the properties that you normally expect from let us say object oriented programs or programs with pointers and so on and so forth. Because, the notions of l value and r value are very crucial that. So, this notion this particular assignment essentially treats this variable as just another name for atom. And, that is way the substitution is possible otherwise under l value r value conditions pure substitution would not be possible.

(Refer Slide Time: 13:08)



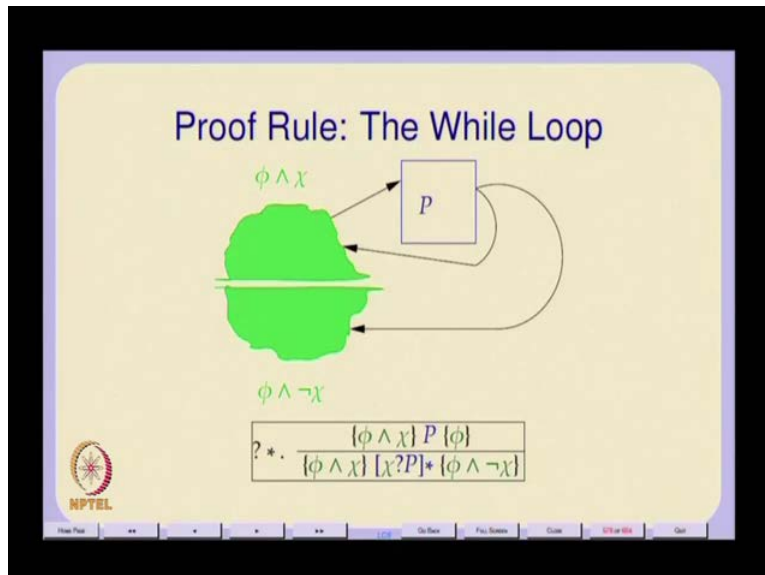
Composition so essentially what we are saying is you start from a state satisfying a predicate ϕ . And, the effect of P would be to give you a state satisfying some ψ and the effect of Q would be to give you on that state satisfying ψ would be to give you a state satisfying ψ . And, that composition is essentially functional composition written as this proof rule.

(Refer Slide Time: 13:40)



The, conditional of course is just that you take this state satisfying phi and that can be divided in 1/2 between those that subset we satisfies the condition chi and that is. And, the other 1/2 which does not satisfy chi. So, I have put an a huge gap just to emphasize the difference. So, essentially what we are saying is you execute this program you execute this conditional if chi then P else Q. As, a if it starts form a state satisfying chi and phi then, it will take Q to some post condition to a state satisfying psi through P. And, if it starts in a state not satisfying chi but satisfies only phi then it will take you through Q again to the set satisfying psi. So, the conditional allows a branching of the division of the input predicate in two subsets to partitions the inputs into two subsets.

(Refer Slide Time: 14:48)



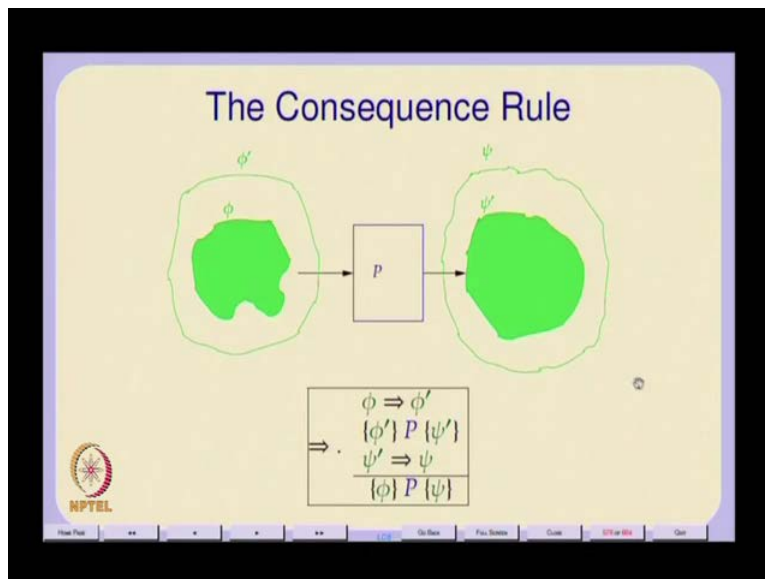
Finally, the loop the proof rule for the loop and one of the things that is always been the bug bare of most programmers is understanding the notion of a invariance. So, essentially if you look at this proof rule what we are saying is there is this state in located in phi. There, are two possibilities either chi is satisfied or chi is not satisfied. If, chi is not satisfied of course then, what happens is it actually nothing happens it remains in this state I mean the effect of the program is to do nothing to it. But, if it is in a state if it starts from state satisfying phi and chi. Then, it comes back to some other state may be satisfying both phi and chi or it goes to another state satisfying chi or phi but not chi. Now, this looping can happen as often as possible and this phi therefore actually is sort of I mean it you can think of it as, that while loop as and if then else

within infinite unfolding and infinitely nested if then else which has infinite exit branches depending upon the number of iterations of the while loop.

And, this phi this loop invariant is essentially a disjunction of all those thing that guarantee 0 iterations, 1 iteration, 2 iteration, 3 iteration and so on in infinitely. So, that is why that is actually an infinite disjunction but, what you want to do is that you want to capture that infinite disjunction by what you call an invariant property. So, this invariant properties are but of course what these invariant properties are what make your understanding of the loop actually quite complete. In this way there are some trivial invariants not like the predicate rule itself is an invariant anyway of every loop.

But, unfortunately that is not helping you prove the program. So, what you require is a fairly strong invariant properties so if the choice of a fairly strong invariant property which allows you to prove the program is what you are usually looking at will come to that. But, more importantly most often your specification is some phi and some psi and the relationship between the phi and psi is not that of is not necessarily that of implication.

(Refer Slide Time: 17:51)



So, there is a weakening rule called the Consequence Rule. So, this is the only logical rule which is not part I mean which is not structurally inductive in the sense that. Here, you are actually using the theory of first order logic completely so take a look at this. So, here the picture

essentially says that supposing I want I have got this specification $\phi \rightarrow P \rightarrow \psi$. So, I want to prove that this program P satisfies the specification pre conditional ϕ and post condition ψ . But, I am able to prove was stronger property. And, what do I mean by a stronger property? I mean that supposing there is a ϕ' which is a weaker than ϕ and there is a ψ' that is stronger than ψ . The notion of strong and weak is just related by implication so if ϕ implies ϕ' then ϕ is stronger than ϕ' . So, if I can prove that from a larger subset of states I am always guarantee to go to a smaller subset in the output.

Then, I can claim that from this smaller subset I am always going to some element in this larger set. So, this is a weakening rule and in fact in the context of so this is what allows you to go from loop invariants to specifications. So, if your loop invariant is some ϕ_i and you are suppose to prove that this while loop actually satisfies some ϕ and some ψ pre condition and post condition. Then, all you require is that this ϕ_i imply this ϕ . And, this ϕ_i and ψ imply that ψ that you are looking at. So, you still have to find the strong enough invariant but having found this string enough invariant. You relate it to the ϕ and ψ are of you original specification through these logical implications. Now, these logical implications are essentially things that require a proof an preferably an automated proof. But, in our case will just do a hand proof of let us say a program.

So, what these $\phi \rightarrow \phi'$ and $\psi' \rightarrow \psi$ are essential components of given starting form a given specification going to an implementation and proving its correctness these are essential components. So, normally what happens in modern program verifiers is that you have a separate first order logic theorem prover. And, you form out all these implications to that theorem prover. So, that proves it in the first order theory of integers or whatever structure is your underlying model and the results and the true or and the results of that come. So, what your actual program verifier does is? It just gets these triples and all the connection between various kinds of triples have to be done through the theorem prover let us say a first order logic theorem prover in this case.

(Refer Slide Time: 21:45)

Proof Rules for Partial Correctness

$$\epsilon. \frac{}{\{\phi\} \epsilon \{\phi\}}$$

$$:=. \frac{}{\{\{t/x\}\psi\} x := t \{\psi\}}$$


$$[]. \frac{\{\phi\} P \{\phi\}}{\{\phi\} [P] \{\phi\}}$$

$$?*. \frac{\{\phi \wedge \chi\} P \{\phi\}}{\{\phi \wedge \chi\} [\chi?P]* \{\phi \wedge \neg\chi\}}$$

$$; \cdot \frac{\{\phi\} P \{\xi\} \quad \{\xi\} Q \{\psi\}}{\{\phi\} P; Q \{\psi\}}$$

$$?: \cdot \frac{\{\phi \wedge \chi\} P \{\psi\} \quad \{\phi \wedge \neg\chi\} Q \{\psi\}}{\{\phi\} \chi?P : Q \{\psi\}}$$

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




So, will see how this rules are applied again by so here is a summary of all these rule of Partial Correctness. There, is no difference is just to put them all in one place it is a same the only difference is there is the bracketing rule. This, is of course purely syntactic and what we are saying is that brackets do not have any particular significance. So, we just this should have been a psi here written phi here should have been psi both the numerator and the denominator.

(Refer Slide Time: 22:19)

Towards Total Correctness

1. The only construct which may not terminate is the while loop.
2. Termination of the while loop may be proven separately as follows:
 - (a) Define a "measure" called the **bound function**, β which satisfies the following properties.
 - The $Dom(\beta)$ is the tuple of possible values of the program variables (\vec{v}) .
 - The $Ran(\beta)$ is a well-ordered set (usually a subset of the naturals) ordered by an irreflexive and transitive relation $<$, and bounded below by a least element 0 .
 - $\phi \wedge \chi \Rightarrow \beta(\vec{v}) > 0$ and $\beta(\vec{v}) = 0 \Rightarrow \phi \wedge \neg\chi$
 - Each execution of the body of the loop decreases the value of the bound function.



Now, however now there is a question of Total Correctness. So, if you look at all the basic constructs except for the while loop. So, termination is guaranteed in all constructs except when there is a while loop so that is obvious. So, now what so therefore the proof of total correctness and therefore of termination can actually be separated out from the proof of partial correctness by proving termination separately. And, so what we do is we have to define some measure called the bound function. And, let me call that Beta. So, beta is actually a function of all the program variables. So, which means you basically take the free variables of your program and beta is some function about that such that so essentially the domain of this function beta is the set of all triple values of the program variables taken in some order. Take all the programs variables in some order triple look at all so essentially it is a Cartesian product that is the domain. The, range of beta should be some well ordered set. Which, means it could be an infinite set. But, it has a bound below.

So, it is well ordered so it has to be an ordering relation. So, let us call that less than and of course the converse of less than is greater than it is bounded below by some least element for the moment let me call it 0. Basically, you take any countable well ordered set. That, countable well ordered set can be placed in one to one correspondence with a subset of the naturals. So, I am not saying so that is why I am using this 0 here that is a sort of typical element it need not be 0 it could be some so, you take any well ordering any well ordered set it can be mapped on to a subset of the naturals. Such, that the well ordering the less than relation on the well ordered set corresponds to the less than relation on naturals. And, this well ordered subset of naturals of course has a least element and that is the bound that you are looking at.

So, I am for the moment I am just calling it I am just using 0 as I am using a red 0 to say that this red color indicates that you know what we are talking about here is not exactly related everything it may not be related to the domain of structure the model on which the program is based. So, it may so in the sense that if you were to take some program. Let, us say on some data structure with no arithmetic in it. Let, us say you are just doing less processing or some such thing with no arithmetic. But, you can still get your bound functions and so on and so forth and arithmetic in terms of length of that list length of the list that is left to be processed and so on and so forth.

So, this red color indicates that these that domain the range of B need not necessarily lie within the same model of the program variables. It could be different and in particular you it could be so

you may not be dealing with a natural numbers at all in your programs. But, however your bound functions would still involve counting. Let, us say at least it can be mapped on to a counting problem which so therefore it goes down to it can always be mapped on to just the termination aspect just bound function can be mapped on to some subset of the naturals under less than. What is bound function needs to guarantee? What is the relationship it has with the program variables? One is that when this invariant property and the condition are both true. Then, this bound function should evaluate to something other than the bottom element of your well ordered set. So, it should be somewhere higher order so that is this greater than 0 that I have written here. And, whenever this bound function is 0 or whenever it reaches the bottom of the well ordered set. The bound of the well ordered set it should definitely mean the negation of the condition of the while loop. Now, notice it these are the actually one way implications.

So, one thing is that it is quite possible that this condition becomes false even though the bound function does not reach the bottom of the well ordered set that is possible. And, if the bound function is positive that does not necessarily mean that the condition should be true. If, you want to understand these things just take something like GCD. Let, us take the GCD program I do not have to write it out we all know how to do GCD. So, that the GCD traverses down essentially given a initial set of naturals x and y both them positive it travels down a well ordering. The ordered pair x and y keeps reducing according to a certain let us say Lexico graphic ordering of on pairs. And, it never touches 0 but it will terminate as soon as the ordered pair is two elements of the ordered pair are equal for example. And, so therefore the fact that the bound function never reaches in that case the bottom element of the but your still your range is still in a well ordered set. So, this is by the way range B ordered actually I should have said B is the this beta should be an injective function into a well ordered set. But, so that is so these two implications are carefully chosen to satisfy only the forward properties and not necessarily the reverse properties. So, you can terminate the loop before you reach the bottom of the ordering.

And, if you have reach the bottom of the ordering then it does not mean that your if you have not reach the bottom of the ordering then it does not mean that you have any more iterations to do. But, if you have any iterations left to do then you should not have reached the bottom of the ordering. And, if you have reach the bottom of the ordering then there should be anymore iterations that is way the boundary function has to be designed as a measure. And this can be

very trivially in certain cases but it can also be quite complicated in certain other cases. I will show you where it is there is an still open problem regarding these bound functions which have available I will let you know. It actually came as a problem of number theory but we can always translated into a problem of programming.

(Refer Slide Time: 30:56)

Termination and Total Correctness

$\epsilon!$ $\frac{}{[\phi] \epsilon [\phi]}$	$:=!$ $\frac{}{[[t/x]\psi] x := t [\psi]}$	$\square!$ $\frac{[\phi] P [\phi]}{[\phi] [P] [\phi]}$
$;! $ $\frac{[\phi] P [\xi] \quad [\xi] Q [\psi]}{[\phi] P; Q [\psi]}$	$? :! $ $\frac{[\phi \wedge \chi] P [\psi] \quad [\phi \wedge \neg \chi] Q [\psi]}{[\phi] \chi? P : Q [\psi]}$	$\Rightarrow!$ $\frac{\phi \Rightarrow \phi' \quad [\phi'] P [\psi']}{[\phi] P [\psi]}$
$?*!$ $\frac{[\phi \wedge \chi \wedge \beta(\vec{v}) = b_0 > 0] P [\phi \wedge \beta(\vec{v}) < b_0]}{[\phi \wedge \chi] [\chi? P]* [\phi \wedge \neg \chi]}$		

where β is a bound function satisfying the **properties**

So, now our termination rules are exactly like our partial correctness rules except for that this explanation not indicates termination except for the while rule. So, what we are saying is again I am using red because my measures of termination may be different from the domains and using for programming. So, I have a looping variant I have a condition and I have a bound function of the programming variables which I am writing as v vector. Which, has some value b naught which is greater than the bottom element of the well ordered set. And, each execution of the body of the loop maintains a program a maintains a invariant property phi. It may not maintain the condition chi but what it does due to the measure is that the new measure the new value of the measure on the new value of the program variables is should be guaranteed to lesson to traverse down the at least one step in the well ordering. So, if you started with an initial value of b naught now beta of v evaluator should give you something less than b naught. And, if you guarantee this then what we are saying is that this loop will always terminate. Because, this measure cannot go below the lower bound of the well ordering and when it does terminate the condition would be false. So, beta is of course the bound function satisfying these properties.

(Refer Slide Time: 32:58)

Example: Factorial

$$\begin{array}{l}
 [x = x_0] \\
 p := 1; \quad [x = x_0 \wedge p = 1] \\
 \Downarrow \\
 [(x_0 = x) \wedge (x \geq 0 \rightarrow p * x! = x_0!) \wedge (x_0 < 0 \rightarrow p = 1)] \\
 \Downarrow \\
 [\phi_0 \wedge (x \geq 0 \rightarrow p * x! = x_0!)] \\
 [x > 0? \quad [\phi_0 \wedge (x > 0) \wedge (p * x! = x_0!) \wedge \beta(x, p) = x = \beta_0 > 0]] \\
 [p := p * x; \quad [\phi_0 \wedge (x > 0) \wedge (p * (x - 1)! = x_0!) \wedge \beta(x, p) = x = \beta_0 > 0]] \\
 [x := x - 1 \quad [\phi_0 \wedge (x \geq 0) \wedge (p * x! = x_0!) \wedge \beta(x, p) = x < \beta_0 > 0]] \\
] \\
]^* \quad [\phi_0 \wedge (x = 0) \wedge (p * x! = x_0!) \wedge \beta(x, p) = x = 0] \\
 \Downarrow \\
 [(x_0 \geq 0 \rightarrow p = x_0!) \wedge (x_0 < 0 \rightarrow p = 1)] \\
 \text{where } \phi_0 \equiv (x_0 \geq 0) \wedge (x_0 < 0 \rightarrow p = 1)
 \end{array}$$

Now, let us just look at this Factorial and let us just do a quick proof. So, what I have done now is this typically I should have written them as triples. But, under sequential composition the right hand side of the k'th the post condition of the k'th instruction is often the precondition of the k plus one'th instruction. So, instead of writing it as tuples I have written it as pairs. So, except when the precondition of the next instruction is different or I need to prove some logical consequences so on and so forth. So, you can think of these downward implication signs essentially the use of logical consequence and whatever precedes this statement is the precondition. So, this is a precondition and notice that the assignment rule if I start with a precondition which does not involve anything to do with the variable that is being assigned.

Student: Sir if so in nothing to natural numbers from of beta is it compulsory in it.

It is not compulsory.

Student: Can we if it mapped to the real numbers then it.

The choice of beta has to be done by the programmer. It cannot be mapped to the real numbers because the real numbers are not well ordered by a well ordering I mean I am actually you for all practical purposes you can think of it is a discrete set not a set which is got is it cannot so even if

you take. Let, us say the non negative real numbers the non negative real numbers have a lower bound but any open inter open sub interval within them does not have a lower bound.

Student: So if it is the non negative real number.

So, what it means is that those kinds of bound functions do not guarantee anything because you are not getting a well ordering. So, the whole point is that your programming language uses discrete sets and so you have to ensure discreteness and a well ordering which so what are we saying now by when we say a well order set. We are actually saying that you take any element in that well ordered set there does not exist in infinite descending chain to the bottom. Whereas, you take the non negative real even though they are bounded from below there exist a lot of infinite from a any positive real number there are more than a countably infinite number of length descending chains.

So, that is not valid whenever I say well ordering I mean that all descending chains have to be finite. They, cannot be infinite the set itself can be infinite but from any particular point on the set you cannot do an infinite decent. Now, that automatically throws out the real's it and once it throws out the real's I mean you could use rational's if you like. But, you know that also throws out in fact you cannot even use den sets. So, you have to use discrete sets but then any discrete set would just be at most countably infinite and therefore there is a mapping into the naturals so that is what happens.

So, what usually happens is that I mean if I am my this programming language of course is actually independent of the domain that we are considering. So, as I said might you might just be doing manipulating a lists of let us say characters or integers or some sets I will let us lists of characters and they might be no notion of integers. But, your bound function is still somehow related to the structural inductive co property of the list and there has to be a finite bound always you cannot descend in infinitely that is the point. So, you could so your well ordering could be some structurally inductive structure which is not necessarily in the naturals. But, if it is well ordered then I can provide a I can there is a through zone's lemma and various things you can naturally provide a mapping into the naturals. So, that is not a problem so in the case of this particular case. Of course, what we are going to do is. I measure is going to be just one of the variables itself the variables that is decreasing and that is so it is as simple as that. But, so here

what happens is let us look at these assignment so essentially if I take if this to be ψ . Then, what we are saying is the precondition should be something that such that.

If, I replace all occurrence all free occurrences of p by 1 by this 1 . Then, I get something that is true so actually this one actually is x equals x , x naught implies x equals x naught and 1 equals 1 . So, there is an implication there so I am actually using the assignment so that is a this backward assignment is one of the most engineers things that came up with the back what is known as the backward rule for assignment. And, that is essentially this. So, this essentially works out to 1 equals 1 . If, you look at the assignment rule which says that I replace all free occurrences of this predicate of p in this predicate by this term 1 .

And, when I replace all free occurrences of p in this predicate by this term 1 I get 1 equals 1 . And, that is a trivial tautology of first order logic with equality on terms. And, of course then we have this which is so if x is greater than or equal to 0 . Then, my invariant property essentially says that I take p any time and take the factorial of x and I should get the factorial of x naught prime. And, initially that is true in fact your initialization for p is guided essentially by that. So, that a by the identity element for multiplication so this is true and of course if x naught is less than 0 then of course we are setting p equals 1 . So, I am taking the technically complete total correctness specification that I mentioned earlier and what I am going to do is. I am going to essentially prove that prove this post condition if, x naught is greater than or equal to 0 then p is equal to factorial of x naught. And, if x naught is less than 0 then p is equal to 1 . I am going to prove this as a post condition and in order to prove this. I am finding all these kinds of logical consequences which, would somehow help me come up with an invariant property. Because, I got a loop here so I require a loop I require an invariant property for the loop.

So, I have to find various kinds of logical consequence I have to massage things in such a way that I can get my invariant. One thing is if you look at this predicate x naught less than 0 arrow p equals 1 is and x naught equals x here. This x naught equals x should go this ϕ naught so, I let me take this ϕ naught. This, ϕ naught does not mention any x anywhere and if x naught is greater than 0 greater than or equal to 0 . Then, anyway this predicate x naught less than 0 arrow p equals one is really true. So, this ϕ naught is actually anyway going to be invariant of this loop. It is not going to change in this loop its truth is not going to p alter in this loop. So, I factor out this ϕ naught here and here since x naught is equal to x I can actually write this.

And, I have therefore ϕ and x is greater than or equal to 0. Then, p multiplied by x factorial is x factorial. When I have this condition x greater than 0 then I can make that a conjunct of this. And, when I make that conjunct of this I get x is greater than 0 and p multiplied by x factorial equals x factorial. And, what I am going to do is and what I am saying is a part of a log and this actually this whole predicate ϕ and this x greater than or equal to 0 arrow $p \star x$ factorial equals x factorial and x greater than 0 together actually implies this whole thing.

And, I have brought a it brings in the bound function also automatically because my bound function is just the variable x the value of the variable x . So, the x greater than 0 ensures that my bound function I have got only two variables in the program x and p . And, I am defining my beta of x and p to be just equal to x . And, that is anyway greater than 0 here of course red 0 and the violet 0 are the same 0. Where, the both same brown 0 that is because it is an integer programs but so if this condition is true. Then, this is what I am going to keep as my sort of invariant property the everything other than red is the invariant property.

So, now when I look at this assignment so I am going to do these two assignments. So, look at this assignment p is assigned $p \star x$. So, what we are saying here is so take this condition and replace p by $p \star x$. That, is what your assignment goes backward from the post condition to the precondition. If, you replace p by $p \star x$ and you have got x minus 1 factorial then by the associativity of multiplication you get $p \star x$ factorial equals x factorial. So, this is hold but, the invariant does not necessarily hold so what happens here is that I am replacing.

So, if you take here $p \star x$ factorial x factorial then, if I replace this assignment is x is assigned x minus 1. So, if I replace x by x minus 1 syntactically then, I get this 1. So, notice that in this case I have maintained this invariant portion this is ϕ and x greater than or equal to 0. And, $p \star x$ factorial equals x factorial is the invariant ϕ . And, that is maintained from here to here. In, between because of these updations the invariant gets partially destroy and then it gets restore. So, in fact here it is here this invariant property is got destroyed because I have x minus 1 instead of x . But, then doing the assignment x is assigned x minus 1 restores the invariant.

So, with inside the loop the invariant can get destroyed and restored. But, at the end of the loop before you go back to checking this condition again that invariant has to hold and it does hold as a termination is concerned. What, we have ensured is that the bound function reduces by at least 1. And, the entire theory of proving programs using invariants essentially requires this at the start of the loop you have to define an invariant a strong enough invariant. Before, the end of the loop that invariant has to be restore even if it gets destroyed in between. So, this ensures that this loop with this precondition and this post condition is perfectly correct. And, of course this post condition implies this the post condition of the actual program that we wants. So, we have used using the consequence axiom consequence rule in order to show that this is this program is totally correct. So, this is the formal basis by which you have to do program verification. In, a typical programming codes where there is an first order logic.

What we usually do is we usually should look for an invariant property a strong enough invariant property. And, you should look for a bound property and you have to at least intuitively argue. That, you are bound function decreases with every iteration. And, usually in a typical programming course we take the bound function to be in their naturals or some naturals or some such thing or some finite subset of integers or at least subset of integers bounded below. And, that invariant property normally I explain it as saying it essentially gives you a conjunction of two things. How, much work has your loop already done? How, much more work left needs is left to be done? So, this bound function. How much work you already done is what this invariant tells you. And, how much work possibly needs to be done is what this bound function tells you. This, bound function essentially gives you an upper bound on the amount of work you may have to do. Then, that is the number of iteration's left is at most bounded by that.

So, this is the rest I think you should go back to your original programs and write some trivial programs. And, see that you can come up with invariants even if you could not come up with them in a CSL 102 at least. Now, in the light of fresh situation may be you should try to do it. And, by the way all these rules these rules have been applied exactly syntactically where the syntax varies we have to prove logical consequence a completely syntactically. Remember that that is important that something that we do not normally emphasize in programming because, that takes too much time. But, for small programs it so which it is completely syntactic what it

means is that you can actually do a backward movement from the post condition to some precondition. And, then form out the corresponding these are known as verification conditions.

These logical implications which use the consequence rule are called verification conditions. So, basically you what you want to do is you want to go through these rules syntactically driven. And, come up with all kinds of predicates then, you have a specification which is some ϕ and some ψ . And, you have to prove what are known as verification condition you have to prove that precondition implies whatever precondition you are calculated syntactically. And, whatever post condition you have calculated syntactically implies the post condition that is given. So, those two verification conditions have to be formed out to let us say first-order theorem prover you know independent of the program. So, the two modules remain independently there is a syntactic engine with just does a broot force propagation of rules through the rules of post conditions and preconditions and fines. And essentially annotates every line of the program and then the connections using logical implications are all formed out separately to a theorem prover. So, that is what happens talking about an open problem a lot of you after forth year will have lots of time in your hands. In, fact you will have the rest of your life in your hands so you are so maybe you spend some time thinking about this it is called the COLLATZ PROBLEM. And it was actually it was actually defined by a Dutch mathematic student and it is a very famous open problem in number theory. But, actually what this problem is that it just it is just it just worded as an iterative program.

You, can think of it is a while program. So, what you are saying is you take an positive integer greater than 1 I mean if it is 1 there is nothing to be done. If, it is even divided by 2 if it is odd multiplied by 3 and add 1 do these two do these operations repeatedly. And, the conjecture is that it will always terminate with 1 this process cannot be infinite. So, dividing by 2 reduces takes you down your well-ordering. But, multiplying by three and adding 1 makes it even you can think of it as multiplying by 3 and adding 1 and dividing by 2 also as 1 operation if you like. But, that takes you up the ordering of the naturals so it is a sort of fluctuating thing and the conjecture is that it cannot do that forever it has to come down to 1 at some point.

So, this is called the COLLATZ PROBLEM. And, basically what is this is a simple single while loop that is it. While so, if x is even x is assigned x due 2 if x is odd x is assigned $3x + 1$. And, now what you are saying is found a bound function which you will ensure that it

terminates. It touches that is really what you are looking at and, the whole thing is within first-order number theory. And, there is at the moment at least from the point of the view of that problem there is not anything more than first-order number theory.

You, require because the signature you are using is very simple you just using either, successor and you are using division by 2 and multiplication by 3 that is it. So, you are doing something very trivially it is a very trivial program. But, because of the fluctuations it is not clear what patterns was fluctuations takes place it is been an open problem for over 100 years and it is not clear what kinds of techniques should be used to prove that this converges to 1 always. So, those of you are on the threshold of graduation I mean you can spend the next 50, 60 years trying this out typing you will have plenty of time to try them out.