# LECTURE NOTES 20210224 HOARE LOGIC 1

In this lecture, we start to learn the first approach of describing program specification and/or program semantics.

# Assertions

In order to talk about what properties a program does/should satisfy, we have to be able to talk about properties of program states (程序状态) first.

To talk about specifications of programs, the

first thing we need is a way of making asser-

tions about properties that hold at particular

points during a program's execution — i.e.,

claims about the current state of the memory

when execution reaches that point.

--- << Software Foundation, Volume 2 >>

Informally, an assertion (断言) is a proposition (命题) which describes a particular property of program states. Using the following C function as an example,

int fib(int n) {

int a0 = 0, a1 = 1, a2;

int i;

for (i = 0; i < n; ++ i) {

a2 = a0 + a1;

a0 = a1;

a1 = a2;

}

return a0;

}

In this C function, there are only 5 program variables, a0, a1, a2, i and n. A program state is determined by these program variables' values and the followings are typical program assertions.[[a0]] = 0 AND [[a1]] = 1[[a0]] < [[a1]]EXISTS k, [[a0]] = fib(k) AND [[a1]] = fib(k+1) AND [[a2]] = fib(k + 2)In more general cases, a C program state contains program variables' value, program variables' address, memory contents etc. In the last lecture, we have seen a wrong program which computes the sum of elements in a linked list. Here is a correct program.

struct list {unsigned int head; struct list \*tail;};

unsigned int sumlist (struct list \* t) {

unsigned int s = 0;

while (t) {

s = s + (t->head);

t = t->tail;

}

return s;

}

And here is an assertion.( [[t]] ⟼ 0 ) AND ( [[t]] + 4 ⟼ NULL ) AND ( [[s]] = 0 )We introduce a new predicate in this assertion X ⟼ Y. It means that the value stored on address X is Y.

## What is a "proposition"?

As mentioned above, an assertion is a proposition which describes a property of program states. And we have seen many assertions already. You may still ask: what is a proposition, formally?Mainly, it is a philosophical question. We have two answers to it. Answer 1: a proposition is the sentence itself which describes the property. Answer 2: a proposition is the meaning of the sentence. The math definitions of "proposition" beyond these two answers are different. For example, assertions may be defined as syntax trees (sentences) or sets of program states (meaning of sentences). Both approaches are accepted by mathematicians and computer scientists. In this course, we will just say "propositions" when we do not need to distinguish these two representations.

## Assertions v.s. boolean functions

On one hand, assertions and boolean functions are different.1. Not all assertions can be represented as boolean function. Here is an example:FORALL k, k < [[n]] OR (k is\_prime) OR (fib(k) is\_not\_prime)2. Not all boolean functions can be represented as assertions. There can be side effects.3. Assertions and boolean functions are categorically different. Assertions describes properties but boolean functions are mainly about computation.On the other hand, there are some connections between them. Many dynamic program analysis tools do use boolean functions to represent assertions.

# Assertion equivalence and comparison

Given two assertions P and Q, if every program state m which satisfies P also satisfies Q, we say that P is stronger than Q, or written as P  ⊢ Q. If P is stronger than Q and Q is stronger than P at the same time, we say that P and Q are equivalent with each other. We write P ⊣ ⊢ Q.

# A formally defined toy language

Before we go on and introduce more advanced concepts, it is important that we can make things really formal. Specifically, we will have a formal programming language (but a simple one) and a formal assertion language. Since it is the first time that we use Coq formal definitions in this course, we hide those Coq code but only show some examples.

Require Import PL.Imp.

Import Assertion\_S.  
Import Concrete\_Pretty\_Printing.

We pack those definitions in another Coq file and we "import" it in Coq by this line of code above.The following instructions tell you how do that on your own laptop. You can also find this instruction from << Software Foundation >> volume 1, Chapter 2, Induction (slightly different).BEGINNING of instruction from << Software Foundation >>.For the Require Import to work, Coq needs to be able to find a compiled version of Imp.v, called Imp.vo, in a directory associated with the prefix PL. This file is analogous to the .class files compiled from .java source files and the .o files compiled from .c files.First create a file named \_CoqProject containing the following line (if you obtained the whole volume "Logical Foundations" as a single archive, a \_CoqProject should already exist and you can skip this step):-Q . PLThis maps the current directory (".", which contains Imp.v, RTClosure.v, etc.) to the prefix (or "logical directory") "PL". PG and CoqIDE read \_CoqProject automatically, so they know to where to look for the file Imp.vo corresponding to the library PL.Imp.Once \_CoqProject is thus created, there are various ways to build Imp.vo:

* In Proof General: The compilation can be made to happen automatically when you submit the Require line above to PG, by setting the emacs variable coq-compile-before-require to t.
* In CoqIDE: Open RTClosure.v; then in the "Compile" menu, click on "Compile Buffer"; Open Imp.v; then, in the "Compile" menu, click on "Compile Buffer".
* From the command line: Generate a Makefile using the coq\_makefile utility, that comes installed with Coq (if you obtained the whole volume as a single archive, a Makefile should already exist and you can skip this step):coq\_makefile -f \_CoqProject \*.v -o MakefileNote: You should rerun that command whenever you add or remove Coq files to the directory.Then you can compile Imp.v by running make with the corresponding .vo file as a target:make Imp.voAll files in the directory can be compiled by giving no arguments:makeUnder the hood, make uses the Coq compiler, coqc. You can also run coqc directly:coqc -Q . PL RTClosure.v coqc -Q . PL Imp.vBut make also calculates dependencies between source files to compile them in the right order, so make should generally be prefered over explicit coqc.

If you have trouble (e.g., if you get complaints about missing identifiers later in the file), it may be because the "load path" for Coq is not set up correctly. The Print LoadPath. command may be helpful in sorting out such issues.In particular, if you see a message likeCompiled library Foo makes inconsistent assumptions over library Barcheck whether you have multiple installations of Coq on your machine. It may be that commands (like coqc) that you execute in a terminal window are getting a different version of Coq than commands executed by Proof General or CoqIDE.

* Another common reason is that the library Bar was modified and recompiled without also recompiling Foo which depends on it. Recompile Foo, or everything if too many files are affected. (Using the third solution above: make clean; make.)

One more tip for CoqIDE users: If you see messages like Error: Unable to locate library Imp, a likely reason is inconsistencies between compiling things *within CoqIDE* vs *using coqc from the command line*. This typically happens when there are two incompatible versions of coqc installed on your system (one associated with CoqIDE, and one associated with coqc from the terminal). The workaround for this situation is compiling using CoqIDE only (i.e. choosing "make" from the menu), and avoiding using coqc directly at all.END of instruction from << Software Foundation >>.

Module Playground\_for\_Program\_Variables\_and\_Assertions.

This toy language only have one kind of program variables—-variables with integer type. And we can introduce some new program variables as below.

Local Instance a0: var := new\_var().  
Local Instance a1: var := new\_var().  
Local Instance a2: var := new\_var().

And now, we can use assertions to talk about some properties.

Definition assert1: Assertion := [[a0]] = 0 AND [[a1]] = 1.  
Definition assert2: Assertion := [[a0]] < [[a1]].

Fibonacci numbers can be easily defined in Coq. But we do not bother to define it here; we assume that such function exists.

Hypothesis fib: Z -> Z.

Z means integer in math. And this hypothesis says fib is a function from integers to integers. We can use this function in Coq-defined Assertions as well.

Definition assert3: Assertion :=  
  EXISTS k, [[a0]] = fib(k) AND [[a1]] = fib(k+1) AND [[a2]] = fib(k + 2).

End Playground\_for\_Program\_Variables\_and\_Assertions.

To make things simple, we only allow two different kinds of expressions in this toy language. Also, only limited arithmetic operators, logical operators and programs commands are supported. Here is a brief illustration of its syntax.a ::= Z | var | a + a | a - a | a \* ab ::= true | false | a == a | a <= a | ! b | b && bc ::= Skip | var ::= a | c ;; c | If b Then c Else c Endif | While b Do c EndWhileNo function call, pointer, no memory space, no break or continue commands are in this language. Also, we assume that there is no bound on arithmetic results.Although this language is simple, it is enough for us to write some interesting programs.

Module Playground\_for\_Programs.

Local Instance A: var := new\_var().  
Local Instance B: var := new\_var().  
Local Instance TEMP: var := new\_var().

Definition swap\_two\_int: com :=  
  TEMP ::= A;;  
  A ::= B;;  
  B ::= TEMP.

Definition decrease\_to\_zero: com :=  
  While ! (A ≤ 0) Do  
    A ::= A - 1  
  EndWhile.

Definition ABSOLUTE\_VALUE: com :=  
  If A ≤ 0  
  Then B ::= 0 - A  
  Else B ::= A  
  EndIf.

End Playground\_for\_Programs.

One important property of this simple programming language is that it is type-safe, i.e. there is no run-time-error problem. We intensionally delete "/" and pointer operations to achieve this. This enables us to introduce new concepts and theories in a concise way. But these theories can all be generalized to complicated real programming languages, like C.

# Pre/postconditions

Remark. Some material in this section and the next section is from << Software Foundation >> volume 2.Next, we need a way of making formal claims about the behavior of commands.In general, the behavior of a command is to transform one state to another, so it is natural to express claims about commands in terms of assertions that are true before and after the command executes:

* "If command c is started in a state satisfying assertion P, and if c eventually terminates in some final state, then this final state will satisfy the assertion Q."

Such a claim is called a *Hoare Triple* (霍尔三元组). The assertion P is called the *precondition* (前条件) of c, while Q is the *postcondition* (后条件).This kind of claims about programs are widely used as specifications. Computer scientists use the following notation to represent it.

      {{ P }}  c  {{ Q }}

# Hoare triples as program semantics

Till now, we have learnt to use pre/postconditions to make formal claims about programs. In other words, given a pair of precondition and postcondition, we get a program specification.Now, we turn to the other side. We will use Hoare triples to describe program behavior. Formally speaking, we will use Hoare triples to define the program semantics of our simple imperative programming language (指令式编程语言).Remark 1. We have not yet describe how a program of com will execute! We only have some intuition on it by the similarity between this simple language and some other practical languages. Now we will do it formally for the first time.Remark 2. When we talk about "program specification", we say whether a specific program satisfies a program specification or not. When we talk about "program semantics", we say the program semantics of some programming language, which defines the behavior of specific programs.

## Sequence

The following axiom defines the behavior of sequential compositions.

Axiom hoare\_seq : ∀(P Q R: Assertion) (c1 c2: com),  
   {{P}}  c1  {{Q}}  ->  
   {{Q}}  c2  {{R}}  ->  
   {{P}}  c1;;c2  {{R}} .

This axiom says, if the command c1 takes any state where P holds to a state where Q holds, and if c2 takes any state where Q holds to one where R holds, then doing c1 followed by c2 will take any state where P holds to one where R holds.Remark. If we instantiate P, Q, R and c1, c2 with concrete commands and assertions, this rule is only about the logical relation among three concrete Hoare triples, or in other words, only describe how the behavior of two concrete program c1 and c2 relates to their sequential combination. But this rule is not about concrete programs and concrete assertions! It talks about sequential combination in general. That's why we say that we are using the relation among Hoare triples to define the semantics of this simple programming language.

## Example: Swapping

We want to prove that the following program always swaps the values of variables X and Y. Or, formally, for any x and y,

        {{ [[X]] = x AND [[Y]] = y }}   
       TEMP ::= X;;  
       X ::= Y;;  
       Y ::= TEMP  
        {{ [[X]] = y AND [[Y]] = x }} .

First, the following three triples are obviously true.

    1.  {{ [[X]] = x AND [[Y]] = y }}   
       TEMP ::= X  
        {{ [[Y]] = y AND [[TEMP]] = x }}   
  
    2.  {{ [[Y]] = y AND [[TEMP]] = x }}   
       X ::= Y  
        {{ [[X]] = y AND [[TEMP]] = x }}   
  
    3.  {{ [[X]] = y AND [[TEMP]] = x }}   
       Y ::= TEMP  
        {{ [[X]] = y AND [[Y]] = x }} .

Then, from 2 and 3, we know:

    4.  {{ [[Y]] = y AND [[TEMP]] = x }}   
       X ::= Y;;  
       Y ::= TEMP  
        {{ [[X]] = y AND [[Y]] = x }} .

In the end, from 1 and 4:

    5.  {{ [[X]] = x AND [[Y]] = y }}   
       TEMP ::= X;;  
       X ::= Y;;  
       Y ::= TEMP  
        {{ [[X]] = y AND [[Y]] = x }} .

## Example: Swapping Using Addition and Subtraction

Here is a program that swaps the values of two variables using addition and subtraction instead of by assigning to a temporary variable.

       X ::= X + Y;;  
       Y ::= X - Y;;  
       X ::= X - Y

Again, we can prove it correct by three triples for assignments and hoare\_seq.

    1.  {{ [[X]] = x AND [[Y]] == y }}   
       X ::= X + Y  
        {{ [[X]] = x + y AND [[Y]] = y }}   
  
    2.  {{ [[X]] = x + y AND [[Y]] = y }}   
       Y ::= X - Y  
        {{ [[X]] = x + y AND [[Y]] = x }}   
  
    3.  {{ [[X]] = x + y AND [[Y]] = x }}   
       X ::= X - Y  
        {{ [[X]] = y AND [[Y]] = x }} .

## Skip

Since Skip doesn't change the state, it preserves any assertion P.

Axiom hoare\_skip : ∀P,  
   {{P}}  Skip  {{P}} .

## Condition

What sort of rule do we want for describing the behavior of if-commands?Certainly, if the same assertion Q holds after executing either of the branches, then it holds after the whole conditional. So we might be tempted to write:

Axiom hoare\_if\_first\_try : ∀P Q b c1 c2,  
   {{P}}  c1  {{Q}}  ->  
   {{P}}  c2  {{Q}}  ->  
   {{P}}  If b Then c1 Else c2 EndIf  {{Q}} .

However, this is rather weak. For example, using this rule, we will fail to show that the following program satisfies the following Hoare triple since the rule above tells us nothing about the state in which the assignments take place in the "then" and "else" branches.In other words, this axiom above does not define the program semantics in a complete sense.

Module Playground\_for\_Counterexample.

Local Instance X: var := new\_var().  
Local Instance Y: var := new\_var().

Definition a\_counterexample :=  
      {{ True }}   
     If X == 0  
     Then Y ::= 2  
     Else Y ::= X + 1  
     EndIf  
      {{ [[X]] ≤ [[Y]] }} .

End Playground\_for\_Counterexample.

If we try to use hoare\_if\_first\_try here, we have to show that

      {{ True }}   
     Y ::= 2  
      {{ [[X]] ≤ [[Y]] }}

and

      {{ True }}   
     Y ::= X + 1  
      {{ [[X]] ≤ [[Y]] }} .

They correspond to two assumptions of hoare\_if\_first\_try. But it is obvious that the first triple of them is not true.That means, we need a better proof rule which can reason about if-then-else in a more precise mannar. For example, in the "then" branch, we know that the boolean expression b evaluates to true, and in the "else" branch, we know it evaluates to false. Making this information available in the premises of the rule forms a more complete definition of program semantics. Here is the Coq formalization:

Axiom hoare\_if : ∀P Q b c1 c2,  
   {{ P AND [[b]] }}  c1  {{ Q }}  ->  
   {{ P AND NOT [[b]] }}  c2  {{ Q }}  ->  
   {{ P }}  If b Then c1 Else c2 EndIf  {{ Q }} .

(\* 2021-02-23 23:55 \*)