THE UNIVERSITY OF MELBOURNE

SWEN90010 High Integrity System

Workshop 8 Hoare Logic

Chuang Wang

THE UNIVERSITY OF MELBOURNE

1. Recap

Rules of Hoare Logic

Rule of Assignment

$$\frac{}{\{P[E/x]x := E\ \{P\}\}}$$

Skip Rule

$$\frac{}{\{P\}\ SKIP\ \{P\}}$$

Rule of Consequence

$$\frac{P' \Rightarrow P, Q \Rightarrow Q', \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$$

Conditional Rule

$$\frac{\{P \wedge B\}\ S1\ \{Q\},\ \ \{P \wedge \neg B\}\ S2\ \{Q\}}{\{P\}\ \boldsymbol{if}\ B\ \boldsymbol{then}\ S1\ \boldsymbol{else}\ S2\ \boldsymbol{endif}\ \{Q\}}$$

Generalised Assignment Rule

$$\frac{Q \Rightarrow P[E/x]}{\{Q\}\ x := E\ \{P\}}$$

Rule of Sequencing

$$\frac{\{P\}\ S1\ \{R\}, \{R\}\ S2\ \{Q\}}{\{P\}\ S1, S2\ \{Q\}}$$

Iteration Rule

$$\frac{\{P \wedge B\}\ S\ \{P\}, P\ \wedge \neg B \Rightarrow \{Q\}}{\{P\}\ \boldsymbol{while}\ B\ \boldsymbol{do}\ S\ \boldsymbol{done}\ \{Q\}}$$

THE UNIVERSITY OF MELBOURNE

2. Exercise

Question 1

Iteration Rule: $\dfrac{\{P \wedge B\}\ S\ \{P\}, P \wedge \neg B \Rightarrow \{Q\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{Q\}}$

The rule of consequence, CONSEQ, is very useful for deriving new rules from existing ones. For example, the course notes for the subject present a slightly less general rule for **while**-loops:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{P \wedge \neg B\}}$$

Show how this rule can be derived from the WHILE rule above.

To prove the more restricted rule, with assumption that $\{\boldsymbol{P} \wedge \boldsymbol{B}\}\ \boldsymbol{S}\{\boldsymbol{P}\}$, it suffices to prove that

$$\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{P \wedge \neg B\}$$

By assumption ✓ — $\{\boldsymbol{P} \wedge \boldsymbol{B}\}\ \boldsymbol{S}\{\boldsymbol{P}\}$

By predict logic ✓ — $\boldsymbol{P} \wedge \neg \boldsymbol{B} \Rightarrow \{\boldsymbol{P} \wedge \neg \boldsymbol{B}\}$

By iteration rule

$$\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{P \wedge \neg B\}$$

Question 2

The following program is designed to sum up the elements of an array a, of length N, where elements of a are indexed from 1.

$$i := 0;\ sum := 0;\ \textbf{while}\ i \neq N\ \textbf{do}\ i := i + 1;\ sum := sum + a[i]\ \textbf{done}$$

An *invariant* I for a **while**-loop **while** B **do** S **done** is a predicate that:

(a) is True at the start of the **while**-loop (i.e. at the point in the program just before the loop's guard B is tested for the first time),

(b) is maintained by each iteration of the loop body S, i.e. $\{I \wedge B\}\ S\ \{I\}$ holds,

(c) is strong enough to conclude that the postcondition of the loop holds, i.e. if the loop's postcondition is Q, then $I \wedge \neg B \Rightarrow Q$ must hold.

Suppose we want to prove that after the **while**-loop above,

$$sum = \sum_{j=1}^{N} a[j]. \tag{1}$$

Find a suitable invariant I for this loop to establish this postcondition, arguing informally that your choice of I meets the three criteria above.

Question 2 (cont.)

$i := 0;$
$sum := 0;$
while $i \neq N$ ***do***
    $i := i + 1;$
    $sum := sum + a[i];$
done

$$sum = \sum_{j=1}^{N} a[j]$$

Iteration	i	sum (Var)	$sum = \sum$
0	0	0	$\sum_{j=1}^{0} a[j]$
1	1	$0 + a[1]$	$\sum_{j=1}^{1} a[j]$
2	2	$0 + a[1] + a[2]$	$\sum_{j=1}^{2} a[j]$

Goal: Find a suitable loop invariant I

The pattern emerging left is that:

$$sum = \sum_{j=1}^{i} a[j]$$

So let I be this statement.
We show each of the three criteria:

1) At the start of the while-loop

$i = 0$ and $sum = 0$.

So $\sum_{j=1}^{0} a[j]$. and so $sum = \sum_{j=1}^{0} a[j]$ is true as required.

Question 2 (cont.)

Iteration Rule: $\dfrac{\{P \wedge B\}\ S\ \{P\},\ P \wedge \neg B \Rightarrow \{Q\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{Q\}}$

Goal: Find a suitable loop invariant I

$$sum = \sum_{j=1}^{i} a[j]$$

Since we know that I holds in the pre-state, we know that

$$sum_0 = \sum_{j=1}^{i_0} a[j]$$

2) In each iteration (informal proof)

We need to show
$\{I \wedge i \neq N\}$
$i := i + 1;$
$sum := sum + a[i];$
$\{I\}$

- Let i_0 and sum_0 be the values of i and sum respectively in the pre-state before the two assignment statements are executed
- Let i_f and sum_f be the final values of i and sum respectively after the two assignment statements are executed

$i_f = i_0 + 1$

$sum_f = sum_0 + a[i_f] = sum_0 + a[i_0 + 1]$

This statement follows trivially

$\sum_{j=1}^{i_0} a[j] + a[i_0 + 1] = \sum_{j=1}^{i_0+1} a[j]$

$sum_0 + a[i_0 + 1] = \sum_{j=1}^{i_0+1} a[j]$

$$sum_f = \sum_{j=1}^{i_f} a[j]$$

Question 2 (cont.)

Goal: Find a suitable loop invariant I

$$sum = \sum_{j=1}^{i} a[j]$$

3) I guarantees the postcondition (when the loop guard is false)

Iteration Rule: $$\frac{\{P \wedge B\}\, S\, \{P\},\ P \wedge \neg B \Rightarrow \{Q\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{Q\}}$$

$i := 0;$
$sum := 0;$
$\{I\}$
while $i \neq N$ **do**
    $i := i + 1;$
    $sum := sum + a[i];$
done
$$sum = \sum_{j=1}^{N} a[j]$$

We need to show

$$\{I \wedge i = N\} \Rightarrow sum = \sum_{j=1}^{N} a[j]$$

This statement holds trivially ✓

$$\left(\left(sum = \sum_{j=1}^{i} a[j]\right) \wedge i = N\right) \Rightarrow sum = \sum_{j=1}^{N} a[j]$$

$$\{I \wedge i = N\} \Rightarrow sum = \sum_{j=1}^{N} a[j]$$

Question 3

Prove the above program correct. First work out what is the weakest precondition P (i.e. the statement that makes the fewest assumptions about the initial state) that makes the following Hoare logic statement apparently true:

$$\{P\}\ i := 0;\ sum := 0;\ \textbf{while}\ i \neq N\ \textbf{do}\ i := i + 1;\ sum := sum + a[i]\ \textbf{done}\ \{sum = \sum_{j=1}^{N} a[j]\}.$$

Then use the rules of Hoare logic to prove that the statement is true for your choice of P.

$\{\boldsymbol{P}\}$
$i := 0;$
$sum := 0;$
$\boldsymbol{while}\ i \neq N\ \boldsymbol{do}$
    $i := i + 1;$
    $sum := sum + a[i];$
$\boldsymbol{done}$
$\boldsymbol{sum = \sum_{j=1}^{N} a[j]}$

The weakest P is $\boldsymbol{true}$.
We therefore prove

$\{\boldsymbol{true}\}$
$i := 0;$
$sum := 0;$
$\boldsymbol{while}\ i \neq N\ \boldsymbol{do}$
    $i := i + 1;$
    $sum := sum + a[i];$
$\boldsymbol{done}$
$\boldsymbol{sum = \sum_{j=1}^{N} a[j]}$

Question 3(cont.)

$\{true\}$
$i := 0;$
$sum := 0;$
while $i \neq N$ **do**
    $i := i + 1;$
    $sum := sum + a[i];$
done

$$sum = \sum_{j=1}^{N} a[j]$$

Sequence Rule

$\{true\}$
$i := 0;$
$\{?\ P\}$
$sum := 0;$
$\{?\ I\}$
while $i \neq N$ **do**
    $i := i + 1;$
    $sum := sum + a[i];$
done

$$sum = \sum_{j=1}^{N} a[j]$$

Three Goals

- Goal 1:

$\{true\}$
$i := 0;$
$\{?\ P\}$

- Goal 2:

$\{?\ P\}$
$sum := 0;$
$\{?\ I\}$

- Goal 3:

$\{?\ I\}$
while $i \neq N$ **do**
    $i := i + 1;$
    $sum := sum + a[i];$
done

$$sum = \sum_{j=1}^{N} a[j]$$

Question 3(cont.)

Iteration Rule: $\frac{\{P \wedge B\}\ S\ \{P\},\ P \wedge \neg B \Rightarrow \{Q\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{Q\}}$

Seq Rule: $\frac{\{P\}\ S1\ \{R\},\{R\}\ S2\ \{Q\}}{\{P\}\ S1,S2\ \{Q\}}$

Generalized Assign Rule: $\frac{Q \Rightarrow P[E/x]}{\{Q\}\ x := E\ \{P\}}$

Assign Rule: $\frac{}{\{P[E/x]x := E\ \{P\}\}}$

Loop Invariant I is $sum = \sum_{j=1}^{i} a[j]$

- **Goal 1:**

$\{true\}$
$i := 0;$
$\{?\ P\}$

Goal 1: Gen Assign rule

$$\frac{\{true\} \Rightarrow \left\{0 = \sum_{j=1}^{0} a[j]\right\}}{\{true\}\ i := 0;\ \{0 = \sum_{j=1}^{i} a[j]\}}$$

- **Goal 2:**

$\{?\ P\}$
$sum := 0;$
$\{I\}$

Goal 2: Assign rule

$P\ is \quad 0 = \sum_{j=1}^{i} a[j]$

- **Goal 3:**

$\{I\}$
while $i \neq N$ **do**
  $i := i + 1;$
  $sum := sum + a[i];$
done
$sum = \sum_{j=1}^{N} a[j]$

Goal 3: Iteration Rule

- **Goal 3.1(formal) :**

$\{I \wedge i \neq N\}$
$i := i + 1;$
$sum := sum + a[i];$
$\{I\}$

Goal 3.1: Seq rule

- **Goal 3.1.1 :**

$\{I \wedge i \neq N\}$
$i := i + 1;$
$\{?\ R\}$

Goal 3.1.1: Gen Assign rule

$$\frac{\left\{\left(sum = \sum_{j=1}^{i} a[j]\right) \wedge i \neq N\right\} \Rightarrow \left\{sum + a[i+1] = \sum_{j=1}^{i+1} a[j]\right\}}{\{I \wedge i \neq N\}\ i := i + 1\ \{R\}}$$

- **Goal 3.1.2 :**

$\{?\ R\}$
$sum := sum + a[i];$
$\{I\}$

Goal 3.1.2: Assign rule

$R\ is \quad sum + a[i] = \sum_{j=1}^{i} a[j]$

- **Goal 3.2 :**

$$\{I \wedge i = N\} \Rightarrow sum = \sum_{j=1}^{N} a[j]$$

Goal 3.2: Q2.3

$$\left(\left(sum = \sum_{j=1}^{i} a[j]\right) \wedge i = N\right) \Rightarrow sum = \sum_{j=1}^{N} a[j]$$

Question 4

Compare the reasoning you just did to the informal reasoning you did earlier about the loop invariant. Ignoring the process of having to come up with the appropriate loop invariant, which kind of reasoning felt more mechanical? Which was more straightforward? Which felt more prone to errors?

The idea is that the **second kind of reasoning with the rules** is supposed to be entirely mechanical, once you have found the appropriate loop invariant. Because it is done by **a series of rule applications and logical reasoning**, the second (formal) approach should also be **less error prone**. Indeed, when reasoning informally about code you have written, it is always easy to convince yourself that your code is correct. The point of Hoare logic is to force you to check your reasoning **objectively**, by relying only on logical deduction and dispensing with intuition.

Where intuition does come in, in Hoare logic, is of course the process of coming up with appropriate loop invariants. However, **once you have a good enough loop invariant, you no longer need to trust your intuition** — you can use the rules of Hoare logic to prove that your intuition was correct (as well as proving your program correct too).

Question 5

Suppose we implemented the above program in Ada, exactly as written, with sum and i being declared as type `Integer`. Does the proof we did above guarantee that the Ada program is correct? Why/why not?

$i := 0;$
$sum := 0;$
$\textbf{while}\ i \neq N\ \textbf{do}$
    $i := i + 1;$
    $sum := sum + a[i];$
$\textbf{done}$

$$sum = \sum_{j=1}^{N} a[j]$$

- i will overflow

  The array length is 2^{31} => the last iteration: $i := (2^{31} - 1) + 1$(overflow!)

- sum will overflow

  The array is $[1,\ 2^{31} - 1]$ => the 2nd iteration: $sum := 1 + (2^{31} - 1)$(overflow!)

No. If the array is too large, the Integer type will overflow, causing a potential runtime error. The proof above deals with mathematical integers, not fixed width ones. It probably guarantees that the program would be correct up to the limits of fixed width integers, although that assumes no bugs in the compiler or operating system etc. on which the program executes.

THE UNIVERSITY OF MELBOURNE

3. Open Discussion

THE UNIVERSITY OF MELBOURNE

Thank you !