Lecture 19:
Invariants of Loops

Aims:

- To look at the inference rule for **while** loops;
- To discuss the role of invariants.

19.1 The Inference Rule for While-Loops

- While

$$\frac{(\!| \mathit{Inv} \wedge B |\!)\, C\, (\!| \mathit{Inv} |\!)}{(\!| \mathit{Inv} |\!)\ \mathbf{while}\ B\, C\, (\!| \mathit{Inv} \wedge \neg B |\!)}$$

  Wff *Inv* is chosen to be an invariant of the loop:

  > provided B is **true** if *Inv* is true before we start C, and C terminates, then *Inv* is also true at the end of C.

- The key to proving the partial correctness of a **while** loop is the discovery of the *invariant*, *Inv*.

  In general, the body of the loop, C, changes the state (changes the values of variables). But we must try to find a relationship between the values of the variables that is preserved by execution of C.

  Indeed, the invariant must be true just before we enter the body of the loop (whether this be for the first time or on some subsequent iteration), and it must be true after we have exited the loop.

  This is reflected in the inference rule. The condition of the rule, $(\!| \mathit{Inv} \wedge B |\!)\, C\, (\!| \mathit{Inv} |\!)$, requires that, whenever we execute the body of the loop, then the invariant is satisfied before and after execution. (In this triple, the precondition also requires that B is true, because this is what will make us execute the body of the loop.)

  The rule allows us to conclude that, prior to the loop, the invariant is true (it is the precondition). And, after the loop, the invariant is still true, but the loop-test is false (this is the postcondition).

  Note that *Inv* does not have to be continuously true during execution of C. All that is required is that it is true before C is executed and true again after C has executed.

- How do we prove the correctness of a **while** loop?

  1. Guess a wff *Inv* that you hope is an invariant. Unfortunately, finding useful invariants requires some ingenuity and a deep understanding of the program. You may be dismayed by this. But really it was only to be expected. Programming itself requires understanding and ingenuity, so we shouldn't expect a free lunch in proving correctness. I'll give some hints in the next lecture at how to discover invariants. For the simple examples and exercises you encounter in this module, finding invariants will not be too hard.

  2. Prove that $(\mathit{Inv} \wedge \neg B) \Rightarrow Q$. As with the Rule of Consequence, we use 'normal' logic and what we know about arithmetic, etc. to do this.
  3. Push *Inv* upwards through C. Let's call the wff you get from this Inv'.
  4. Prove that $(\mathit{Inv} \wedge B) \Rightarrow \mathit{Inv}'$. Again we step outside of Floyd-Hoare logic to do this.
  5. Now write *Inv* above the **while** loop. (Continue to push this up through the rest of the program, if any.)

  If any of the proofs don't go through then either you guessed the wrong invariant or your program is wrong.

- As an observation, what we're doing is a somewhat disguised form is a *proof by induction*.

  The *base case* is that the invariant is true after 0 iterations. We establish this because we write the invariant ahead of the loop and push it upwards, so we're making sure that the earlier parts of the program do establish it.

  The *step case* is where we assume the invariant holds for k iterations and we prove it holds for $k+1$ iterations. The place where we are doing this is when we write the invariant at the end of the body, push it upwards through the body and then write the invariant again and use the Rule of Consequence to show that one implies the other.

  That's our proof by induction. We also have the additional requirement to show that, if we exit the loop, the invariant will imply the postcondition of the program. We achieve this when we push the postcondition up to the point just after the loop and establish a connection using the Rule of Consequence.

- Let's see an example in which we go through those steps. I want to show you that, apart from guessing the invariant, it's all simple stuff —as mechanical as the stuff we've been doing so far. So, to make life easy, I will give you the invariant in this case.

- Prove that $\vdash_{\text{par}} (\!| x \geq 0 |\!)\ \mathit{ProgA}\ (\!| y = x! |\!)$ where *ProgA* is:

```
y := 1;

z := 0;

while z ≠ x
{

    z := z + 1;

    y := y × z;

}
```

As usual, start by writing the program's precondition at the top and the postcondition at the bottom.

$(\!| x \geq 0 |\!)$

$y := 1;$

$z := 0;$

while $z \neq x$
{

$\quad z := z + 1;$

$\quad y := y \times z;$

}

$(\!| y = x! |\!)$

Working upwards, the command prior to the postcondition is the **while** loop. So we go through the five steps given earlier.

The first involves inventing the invariant. I shall just tell you that it is $y = z!$

The second step involves writing $Inv \wedge \neg B$ just after the loop and then show that the two statements 'connect' using the Rule of Consequence:

$(\!| x \geq 0 |\!)$

$y := 1;$

$z := 0;$

while $z \neq x$
{

$\quad z := z + 1;$

$\quad y := y \times z;$

}
$(\!| y = z! \wedge z = x |\!)$
$(\!| y = x! |\!)$Consequence (proof ①)

Proof ①: To show $(y = z! \wedge z = x) \Rightarrow y = x!$.

By arithmetic, we simplify $(\!| y = z! \wedge z = x |\!)$ to $(\!| y = x! |\!)$. So we have $y = x! \Rightarrow y = x! \equiv$ **True.**

Next, we write the invariant at the bottom of the loop body and push it upwards to get Inv'.

$(\!| x \geq 0 |\!)$

$y := 1;$

$z := 0;$

while $z \neq x$
{
$\quad (\!| y \times (z+1) = (z+1)! |\!)$
$\quad z := z + 1;$
$\quad (\!| y \times z = z! |\!)$Assignment
$\quad y := y \times z;$
$\quad (\!| y = z! |\!)$Assignment
}
$(\!| y = z! \wedge z = x |\!)$
$(\!| y = x! |\!)$Consequence (proof ①)

Next we write $Inv \wedge B$ just above Inv' and we show they 'connect' using the Rule of Consequence:

$(\!| x \geq 0 |\!)$

$y := 1;$

$z := 0;$

while $z \neq x$
{ $\quad (\!| y = z! \wedge z \neq x |\!)$Invariant and loop test
$\quad (\!| y \times (z+1) = (z+1)! |\!)$Consequence (proof ②)
$\quad z := z + 1;$
$\quad (\!| y \times z = z! |\!)$Assignment
$\quad y := y \times z;$
$\quad (\!| y = z! |\!)$Assignment
}
$(\!| y = z! \wedge z = x |\!)$
$(\!| y = x! |\!)$Consequence (proof ①)

Proof of ②: To show $(y = z! \wedge z \neq x) \Rightarrow y \times (z+1) = (z+1)!$.

We know that $(z+1)! = (z+1) \times z!$. So if we take $y \times (z+1) = (z+1)!$ and we substitute in, we get $y \times (z+1) = (z+1) \times z!$, and then can divide both sides by $(z+1)$ to get $y = z!$ This leaves us with $(y = z! \wedge z \neq x) \Rightarrow y = z! \equiv$ **True.**

So now we can write the invariant above the loop and a comment at the end of the loop.

$(\!| x \geq 0 |\!)$

$y := 1;$

$z := 0;$
$(\!| y = z! |\!)$
while $z \neq x$
{ $(\!| y = z! \wedge z \neq x |\!)$Invariant and loop test
$(\!| y \times (z+1) = (z+1)! |\!)$Consequence (proof ②)
$z := z + 1;$
$(\!| y \times z = z! |\!)$Assignment
$y := y \times z;$
$(\!| y = z! |\!)$Assignment
}
$(\!| y = z! \wedge z = x |\!)$While
$(\!| y = x! |\!)$Consequence (proof ①)

Now we just push the precondition of the loop (i.e. the invariant that we wrote just before the loop) up through the rest of the program:

$(\!| x \geq 0 |\!)$
$(\!| 1 = 0! |\!)$Consequence (proof ③)
$y := 1;$
$(\!| y = 0! |\!)$Assignment
$z := 0;$
$(\!| y = z! |\!)$Assignment
while $z \neq x$
{ $(\!| y = z! \wedge z \neq x |\!)$Invariant and loop test
$(\!| y \times (z+1) = (z+1)! |\!)$Consequence (proof ②)
$z := z + 1;$
$(\!| y \times z = z! |\!)$Assignment
$y := y \times z;$
$(\!| y = z! |\!)$Assignment
}
$(\!| y = z! \wedge z = x |\!)$While
$(\!| y = x! |\!)$Consequence (proof ①)

Proof ③: To show $x \geq 0 \Rightarrow 1 = 0!$.

$0!$ is 1, so we have $1 = 1$, which is **True**, so we have $x \geq 0 \Rightarrow$ **True** $\equiv$ **True.**

- Here's another one that we will do as a class exercise. This program is supposed to copy the value of x into y, without changing x. Obviously, we could do this with just $y := x$. but then we wouldn't have much of an exercise. So this program does it by repeatedly adding 1 to y enough times.

The invariant we will try is $y + a = x$. Prove that $\vdash_{\text{par}} (\!| x \geq 0 |\!)\ \mathit{ProgB}\ (\!| x = y |\!)$ where *ProgB* is:

$a := x;$

$y := 0;$

while $a \neq 0$
{

$y := y + 1;$

$a := a - 1;$

}

Acknowledgements

I continue to base material on that in Chapter 4 of [HR00].

References

[HR00] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.