Lecture 7: Program Specifications

Aims:

- To look at Hoare triples as a way of writing program specifications;
- To discuss what we mean by partial and total correctness.

7.1 Program Verification

- Our digression into propositional logic is over — at last. Remember why we made this digression. We want to consider how to *prove* that a program is correct.

  We do this when exhaustive testing is impossible (which is pretty much always) *and* when we are working in a domain where the cost of failure is high (e.g. safety-critical domains).

- If you want to provide correctness guarantees about your programs, then software development comprises the following three steps:

  1. Take the **informal** *problem specification PS* and translate it into a **formal** *program specification* $(\!|P|\!)\ (\!|Q|\!)$.
  2. Write a program C that is intended to satisfy the specification $(\!|P|\!)\ (\!|Q|\!)$.
  3. Prove that C satisfies $(\!|P|\!)\ (\!|Q|\!)$, written

  $$\vdash (\!|P|\!)\, C\, (\!|Q|\!)$$

  Obviously, this is hugely simplified. Rarely can any software project be decomposed into nice simple sequential tasks. Often program specifications evolve, there is interleaved or concurrent work on different tasks, and there is an amount of backtracking.

- In many ways, the first task is the hardest. Even coming up with the informal problem specification can be difficult. Translating it into a formal specification, written in some logic, involves all the kinds of difficulties we encountered when we looked at the connections between English and propositional logic. The informal problem specification may be inconsistent, incomplete, imprecise or ambiguous. Indeed one advantage of translating it into logic will be to reveal its inconsistency, incompleteness, imprecision or ambiguity.

- It should also be said that the steps above describe what is sometimes called *after-the-fact verification*. Only when program C has been completed is it proved. Arguably, it is preferable to practice *as-you-go verification*, in which program construction and proof go hand-in-hand. Thus, it is guaranteed that the program that gets constructed meets its specification. We don't have time to look at as-you-go verification.

- Perhaps one of the most important things to take from the highly simplified steps above is that programs can only be proved correct *with respect to a specification.* Without a specification of what the program is supposed to do, talk of correctness or verification or proof is meaningless.
- The advantages of formally specifying and verifying a program include:
  - The specification is an important piece of program documentation.
  - Experience has shown that verifying programs with respect to their specifcations eliminates more errors and does so earlier than testing-and-debugging.
  - This has the added advantage of reducing software development and maintenance time and costs.
  - Properly specified and verified software is easier to revise and reuse.

  In safety-critical domains, we may increasingly face legal or professional requirements that software comes with guarantees.

7.2 The MOCCA Programming Language

- We are going to prove programs that are written in a pretend programming language called MOCCA. It contains the core features of many modern programming languages. Indeed, anything that can be computed can be computed by MOCCA.

  In MOCCA, however, we've stripped away some of the conveniences of real languages. The only reason we do this is to simplify what you have to learn here. It is entirely possible to extend the language with all the features you know from Java, C or C++, and show how to do program verification on the extended language.

  Here is MOCCA's grammar in BNF again.

⟨program⟩	::=	⟨block⟩
⟨block⟩	::=	{ ⟨command-list⟩ }
⟨command-list⟩	::=	ϵ
⟨command-list⟩	::=	⟨command⟩ ⟨command-list⟩
⟨command⟩	::=	⟨block⟩
⟨command⟩	::=	⟨assignment⟩
⟨command⟩	::=	⟨one-armed-conditional⟩
⟨command⟩	::=	⟨two-armed-conditional⟩
⟨command⟩	::=	⟨while-loop⟩
⟨assignment⟩	::=	⟨var⟩ := ⟨expr⟩
⟨one-armed-conditional⟩	::=	**if** ⟨expr⟩ ⟨command⟩
⟨two-armed-conditional⟩	::=	**if** ⟨expr⟩ ⟨command⟩ **else** ⟨command⟩
⟨while-loop⟩	::=	**while** ⟨expr⟩ ⟨command⟩
etc.		

7.3 Program Specifications

- There are many very elaborate ways of writing formal program and system specifications. Here we use one of the simplest, that has always been associated with proving program correctness.
- A program specification comprises a pair of *assertions* about *program states*:

  Precondition P: describes the state before the program executes;

  Postcondition Q: describes the state after the program executes.

- Given a program C, we would write a *Hoare triple*:

$$(\!|\, P \,|\!)\; C \;(\!|\, Q \,|\!)$$

which means (**roughly**)

  If program C is run in a state that satisfies P, then the state after it executes will satisfy Q.

- Some books would write $\{ P \}\, C\, \{ Q \}$ instead of $(\!|\, P \,|\!)\, C\, (\!|\, Q \,|\!)$. But using curly braces is too confusing: they get confused with the curly braces that are allowed in many programming languages.

- The two assertions, P and Q (the pre- and post-condition), must describe *states*. For a simple language, such as MOCCA, we describe *states* by placing conditions on the values stored in different variables.

- Examples of states:

$$(\!|\, x = 3 \wedge y = 4 \,|\!)$$
$$(\!|\, x > 0 \,|\!)$$
$$(\!|\, x = y^2 \wedge z = 2x \,|\!)$$

- Here's a really simple informal problem specification:

Problem 7.1.

Parameters:	*A non-negative integer, x.*
Returns:	*The factorial of x.*

- We take the informal problem specification and translate it into a formal program specification, written as a Hoare triple:

$$(\!|\, x \geq 0 \,|\!)\; C \;(\!|\, y = x! \,|\!)$$

In other words, this specification says: we want a program C which (still **roughly**), if executed starting in a state in which program variable x contains a number greater than or equal to 0, will, after execution of C, end in a state in which program variable y contains $x!$, the factorial of x.

- How the precondition becomes true is not relevant. (Maybe parameters are passed in and used to initialise x; maybe the user is prompted to enter an integer.) Similarly, what we do with any values computed to make the precondition true is not relevant. (Maybe the value of y is returned to some client code. Maybe the value is displayed on the screen or written to a file.) We are not concerned with these details. We are concerned only with the correctness of the computation itself.

- In fact, there is something wrong with this program specification, which we must remedy. We'll return to this later and fix it.

- Suppose we then try to write a program C to satisfy this specification. There are many programs we could write. Some will satisfy the specification, and some won't. Here are two examples.

- This one is true. Still **roughly**, if this program is executed in a state where $x \geq 0$ then, after execution, the program will be in a state where $y = x!$.

```
(| x ≥ 0 |)
y := 1;
z := 0;
while z ≠ x
{   z := z + 1;
    y := y × z;
}
(| y = x! |)
```

- But this one is false.

```
$(\!|\, x \geq 0 \,|\!)$
$y := 1;$
$z := 0;$
**while** $z \neq x$
{    $y := y \times z;$
     $z := z + 1;$
}
$(\!|\, y = x! \,|\!)$
```

- Unfortunately, this one is also true.

```
$(\!|\, x \geq 0 \,|\!)$
$x := 3;$
$y := 6;$
$(\!|\, y = x! \,|\!)$
```

  If this trivial program is executed in a state where $x \geq 0$ then, after execution, the program will be in a state where $y = x!$.

  Clearly, the program isn't what we wanted when we wrote the problem specification. But the fault here doesn't lie with the program; it lies with the program specification. The program specification doesn't prevent us from changing the value of x.

- And unfortunately, and much more subtly, this one is false.

```
$(\!|\, x \geq 0 \,|\!)$
$y := 1;$
**while** $x \neq 0$
{    $y := y \times x;$
     $x := x - 1;$
}
$(\!|\, y = x! \,|\!)$
```

  This time the program looks good. It certainly computes the factorial of the original value of x. But again it alters x.

  Suppose x originally contained 4. By the time we come to test the postcondition, y will contain the factorial of the original value of x (4), i.e. y will contain 24. But x will now contain zero, whose factorial is 1. So the postcondition isn't true: $24 \neq 1$.

- The solution to our woes is to distinguish program variables and specification variables.

  Program variables: Appear in the program or in preconditions and postconditions

  Specification variables: Only appear in preconditions and postconditions

    – cannot change value (because, for example, they cannot be used in assignment commands within the program)

  - enable us to 'remember' initial values
  - generally written using a subscript, e.g. x_0

- So here is a revised Hoare triple, using specification variables.

$$(\!| x = x_0 \wedge x \geq 0 |\!)\, C\, (\!| y = x_0! |\!)$$

- Now this is false, as we would hope:

```
(| x = x_0 ∧ x ≥ 0 |)
x := 3;
y := 6;
(| y = x_0! |)
```

- And this one is now true, as we would hope:

```
(| x = x_0 ∧ ≥ 0 |)
y := 1;
while x ≠ 0
{   y := y × x;
    x := x − 1;
}
(| y = x_0! |)
```

Class Exercise

- Write program specifications in the form of Hoare triples for the following problem specifications.

Problem 7.2.

Parameters:	*An integer, x*
Returns:	*Twice the value of x.*

Problem 7.3.

Parameters:	*A non-negative integer, x*
Returns:	*The largest integer that does not exceed $\sqrt{x}$.*

7.4 Partial and Total Correctness

- Our explanation of the meaning of a Hoare triple, $(\!| P |\!)\, C\, (\!| Q |\!)$, has been informal so far. Note how the word "roughly" peppered the text. We did not say what to conclude about a triple if C does not terminate. It has been conventional to separate consideration of termination from other aspects of program correctness.

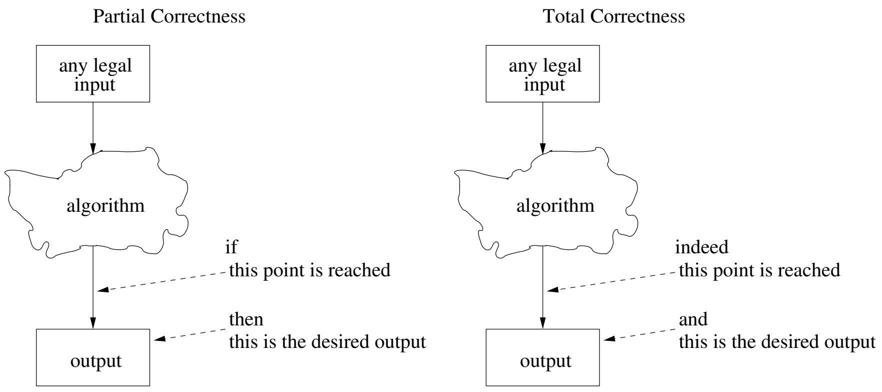
Partial correctness: We say that triple $(\!| P |\!)\, C\, (\!| Q |\!)$ is *partially correct* if, when C is executed in any state that satisfies precondition P, the following holds: if C terminates then, after execution, the postcondition Q holds. In this case, we write

$$\models_{\text{par}} (\!| P |\!)\, C\, (\!| Q |\!)$$

Total correctness: We say that triple $(\!| P |\!)\, C\, (\!| Q |\!)$ is *totally correct* if, when C is executed in any state that satifies precondition P, the following holds: C terminates and, after execution, the postcondition Q holds. In this case, we write

$$\models_{\text{tot}} (\!| P |\!)\, C\, (\!| Q |\!)$$

Obviously, we're really interested in total correctness. The motivation for the distinction is to allow us to split our proofs into more manageable parts. We can prove partial correctness first and then prove termination. And these two proofs establish total correctness.



Class Exercise

- Let's check our intuitions.
- Say which of these hold.
  1. $\models_{\text{par}} (\!| x = 1 |\!)\, x := x + 1\, (\!| x = 2 |\!)$
  2. $\models_{\text{tot}} (\!| x = 1 |\!)\, x := x + 1\, (\!| x = 2 |\!)$
  3. $\models_{\text{par}} (\!| x = 1 |\!)\, y := x\, (\!| y = 1 |\!)$
  4. $\models_{\text{tot}} (\!| x = 1 |\!)\, y := x\, (\!| y = 1 |\!)$
  5. $\models_{\text{par}} (\!| x = 1 |\!)\, y := x\, (\!| y = 2 |\!)$
  6. $\models_{\text{tot}} (\!| x = 1 |\!)\, y := x\, (\!| y = 2 |\!)$
- Say which of these hold.
  1. $\models_{\text{par}} (\!| x = x_0 \wedge y = y_0 |\!)\, \mathit{temp} := x; x := y; y := \mathit{temp}\, (\!| x = y_0 \wedge y = x_0 |\!)$
  2. $\models_{\text{tot}} (\!| x = x_0 \wedge y = y_0 |\!)\, \mathit{temp} := x; x := y; y := \mathit{temp}\, (\!| x = y_0 \wedge y = x_0 |\!)$
  3. $\models_{\text{par}} (\!| x = x_0 \wedge y = y_0 |\!)\, x := y; y := x\, (\!| x = y_0 \wedge y = x_0 |\!)$
  4. $\models_{\text{tot}} (\!| x = x_0 \wedge y = y_0 |\!)\, x := y; y := x\, (\!| x = y_0 \wedge y = x_0 |\!)$
  5. $\models_{\text{par}} (\!| x = x_0 |\!)\, \textbf{while True}\ x := x + 2\, (\!| y = x_0! |\!)$
  6. $\models_{\text{tot}} (\!| x = x_0 |\!)\, \textbf{while True}\ x := x + 2\, (\!| y = x_0! |\!)$

7.5 Proof

- We're not going to show
$$\models_{\text{par}} (\!|\, P \,|\!)\; C \;(\!|\, Q \,|\!)$$
We're going to show
$$\vdash_{\text{par}} (\!|\, P \,|\!)\; C \;(\!|\, Q \,|\!)$$
i.e. we're going to use a deduction system. But the deduction system we use is sound and complete, so $\models_{\text{par}}$ and $\vdash_{\text{par}}$ coincide.
- We're not going to show
$$\models_{\text{tot}} (\!|\, P \,|\!)\; C \;(\!|\, Q \,|\!)$$
We're going to show
$$\vdash_{\text{tot}} (\!|\, P \,|\!)\; C \;(\!|\, Q \,|\!)$$
i.e. we're going to use a deduction system. But the deduction system we use is sound and complete, so $\models_{\text{tot}}$ and $\vdash_{\text{tot}}$ coincide.
- So, it's time to look at the inference rules in the first of the two deduction systems: the rules for proving partial correctness.

Acknowledgements

My approach to this subject is based heavily on that given in Chapter 4 of [HR00]. I acknowledge also the strong influence of Mike Gordon's *Specification and Verification I* course notes [Gor].

The diagram comes from [Har92], p.103.

References

[Gor] M. Gordon. Specification and Verification I (Course Notes).

[Har92] D. Harel. *Algorithmics: The Spirit of Computing.* Addison-Wesley, 2nd edition, 1992.

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