Lecture 17

Introduction to Formal Verification

Program correctness: Does a given program satisfy its specification—does it do what should?

Some techniques for showing program correctness:

- inspection, code walk-throughs
- testing (white box, black box): look at boundary cases (specific inputs).
- *formal verification*: formally state a specification and prove that the program satisfies the specification for all inputs.

Note: Testing is NOT a proof. While it is great for finding bugs, it is not exhaustive.

Specifying software is a widespread practice while formally verifying software is not—sometimes it is unnecessary, sometimes it is too expensive and/or time consuming.

Formal verification of hardware and safety-critical software systems is common.

The steps of formal verification:

1. Convert the informal description $R$ of requirements for an application domain into an “equivalent” formula $\Phi_R$ of some symbolic logic.
2. Write a program $C$ which is meant to realize $\Phi_R$ in some given programming environment.
3. Prove that the program $C$ satisfies the formula $\Phi_R$.

We shall consider only the third part in this course.

For this class we are considering imperative or procedural programs (C/C++ like code).

These programs manipulate the values of “variables”.

The state of a program $C$ is the values of the variables at a particular time in the execution of $C$.

- Expressions evaluate relative to the current state of $C$.
- Executing a statement changes the state of $C$.

Example: States

Consider the states at the “while” test:

```plaintext
y = 1;                \bullet Initial state $s_0$: x=0, z=0, y=1
z = 0;                \bullet Next state $s_1$: z=1, y=1
while (z != x) {
    z = z + 1;    \bullet State $s_2$: z=2, y=2
    y = y * z;    \bullet State $s_3$: z=3, y=6
}                     \bullet State $s_4$: z=4, y=24
\bullet and so forth...
```

1
**Example:** Specifications in English

Compute a number $y$ whose square is less than the input number $x$.

But this will not work for any input number $x$. What if $x = -4$?

We must revise our specification! We need take this issue into consideration.

**Revised example.**

If the input number $x$ is greater than 0, compute a number $y$ whose square is less than $x$.

As seen above, we need information not just about the state after the program executes, but also about the state before it executes.

Furthermore, it is important to note that specification is NOT behaviour.

**Hoare Triples and Correctness**

Our assertions about programs will have the form:

$\langle P \rangle \quad C \quad \langle Q \rangle$

| $\langle P \rangle$ — precondition | $C$ — program or code | $\langle Q \rangle$ — postcondition |

The meaning of the triple $\langle P \rangle \ C \ \langle Q \rangle$:

If program $C$ is run starting in a state that satisfies $P$, then the resulting state after the execution of $C$ will satisfy $Q$.

An assertion $\langle P \rangle \ C \ \langle Q \rangle$ is called a **Hoare triple**.

A specification of program $C$ is a Hoare Triple with $C$ as the second component.

For a Hoare triple, its set of **logical variables** are those variables that occur free in $P$ or $Q$ and do not occur in $C$.

**Example:** Specification.

The revised specification in the previous example might be expressed as:

$\langle x > 0 \rangle \ C \ \langle y \times y < x \rangle$.

We want to develop a notion of proof that will allow us to prove that a program $C$ satisfies the specification given by the precondition $P$ and the postcondition $Q$.

The proof calculus is different from Natural Deduction for Predicate Logic, since Hoare triples have two features not present in logical formulas:

- program instructions (actions), rather than propositions, and
- a sense of time: before execution versus after execution.
Partial vs Total Correctness

A triple $\langle P \rangle \ C \ \langle Q \rangle$ is satisfied under partial correctness, denoted $\vdash_{\text{par}} \langle P \rangle \ C \ \langle Q \rangle$, if and only if for every state $s$ that satisfies condition $P$, if execution of $C$ starting from state $s$ terminates in a state $s'$, then state $s'$ satisfies condition $Q$.

A triple $\langle P \rangle \ C \ \langle Q \rangle$ is satisfied under total correctness, denoted $\vdash_{\text{tot}} \langle P \rangle \ C \ \langle Q \rangle$, if and only if for every state $s$ that satisfies $P$, execution of $C$ starting from state $s$ terminates and the resulting state $s'$ satisfies $Q$.

Total Correctness $=$ Partial Correctness and Program Termination

Note: we usually prove total correctness by proving partial correctness and termination separately.

Example: Infinite Loop.

Consider the following program: while true { x = 0; }  
This program satisfies all specifications for partial correctness.  
It does not satisfy total correctness as it does not terminate.

Exercise: Translate the above definitions into predicate logic.

For further examples, see the Common Overheads.

Proofs

A full proof will have one or more conditions before and after each code statement. Each statement makes a Hoare triple with the preceding and following conditions. Each triple (postcondition) has a justification that explains its correctness.

$\langle \ \langle \text{program precondition} \rangle \ \rangle$
$y = 1;$
$\langle \ldots \rangle$ \hspace{1cm} (justification)

while (x != 0) {
    $\langle \ldots \rangle$ \hspace{1cm} (justification)
    $y = y * x;$
    $\langle \ldots \rangle$ \hspace{1cm} (justification)
    $x = x - 1;$
    $\langle \ldots \rangle$ \hspace{1cm} (justification)
$}$
$\langle \ \langle \text{program postcondition} \rangle \ \rangle$ \hspace{1cm} (justification)

Each construct in our programming language will have a rule. We will consider assignments, conditionals, while loops and arrays.
Assignment Rule

\[
\frac{\langle Q[E/x] \rangle}{\langle Q \rangle} \quad (\text{assignment})
\]

Intuition: \(Q(x)\) will hold after assigning (the value of) \(E\) to \(x\) if \(Q\) was true of that value beforehand.

In program correctness proofs, we usually work backwards from the postcondition:

\[
??? \quad \langle Q[E/x] \rangle \\
x = y ; \
\langle x \quad 0 \rangle \\
\langle Q \rangle
\]

Example 1: \(\vdash_{\text{par}} \langle y + 1 = 7 \rangle \quad x = y + 1 \quad \langle x = 7 \rangle\); by one application of the assignment rule.

Example 2:

\[
\langle y = 2 \rangle \quad \langle Q[E/x] \rangle \\
x = y ; \
\langle x = E \rangle \\
\langle x = 2 \rangle \\
\langle Q \rangle
\]

Example 3:

\[
\langle 0 < 2 \rangle \quad \langle Q[E/x] \rangle \\
x = 2 ; \
\langle x = E \rangle \\
\langle 0 < x \rangle \\
\langle Q \rangle
\]

Sometimes we will need to make use of implied statements.

“Precondition strengthening”

\[
\frac{P \rightarrow P'}{\langle P \rangle \quad C \quad \langle Q \rangle} \quad (\text{implied})
\]

“Postcondition weakening”

\[
\frac{\langle P \rangle \quad C \quad \langle Q' \rangle}{\langle P \rangle \quad C \quad Q} \quad Q' \rightarrow Q \quad (\text{implied})
\]
Example 4:

\[ \begin{align*}
&\langle y = 6 \rangle \\
&x = y + 1 \\
&\langle x = 7 \rangle
\end{align*} \]

Going bottom up, we should consider the segment:

\[ \begin{align*}
&? \quad ? \\
&x = y + 1 \\
&\langle x = 7 \rangle
\end{align*} \]

We see that this can be an application of the assignment rule and annotate it as such:

\[ \begin{align*}
&\langle y = 6 \rangle \\
&\langle y + 1 = 7 \rangle \\
&x = y + 1 \\
&\langle x = 7 \rangle \quad \text{assignment}
\end{align*} \]

Now we must think, how do we go from \( \langle y = 6 \rangle \) to \( \langle y + 1 = 7 \rangle \)?

This will be done via precondition strengthening and annotate it as such:

\[ \begin{align*}
&\langle y = 6 \rangle \\
&\langle y + 1 = 7 \rangle \quad \text{implied} \\
&x = y + 1 \\
&\langle x = 7 \rangle \quad \text{assignment}
\end{align*} \]

Lastly, we need to prove this implication: \( \langle y = 6 \rangle \) implies \( \langle y + 1 = 7 \rangle \).

This can be done using simple arithmetic underneath the annotated program. (Exercise)

Note: if there are many implied statements, it will be best to label them for easy identification in the implied proof section of your formal proof.

All implied statements must be proved.