Hoare Logic and Model Checking

Kasper Svendsen
University of Cambridge

CST Part II – 2016/17

Acknowledgement: slides heavily based on previous versions by Mike Gordon and Alan Mycroft
In the past lectures we have given

- a notation for specifying the intended behaviour of programs
- a proof system for proving that programs satisfy their intended specification
- a semantics capturing the precise meaning of this notation

Now we are going to look at ways of finding proofs, including:

- derived rules & backwards reasoning
- finding invariants
- ways of annotating programs prior to proving

We are also going to look at proof rules for total correctness.
Forward and backwards reasoning
Forward & backwards reasoning

The proof rules we have seen so far are best suited for **forward** directed reasoning where a proof tree is constructed starting from axioms towards the desired specification.

For instance, consider a proof of

\[ \vdash \{X = a\} \ X := X + 1 \ {X = a + 1} \]

using the assignment rule:

\[ \vdash \{P[E/V]\} \ V := E \ \{P\} \]
Forward reasoning

It is often more natural to work **backwards**, starting from the root of the proof tree and generating new subgoals until all the leaves are axioms.

We can **derive** rules better suited for backwards reasoning.

For instance, we can derive this backwards-assignment rule:

\[
\begin{align*}
\vdash P & \Rightarrow Q[E/V] \\
\vdash \{P\} & \quad V := E \quad \{Q\}
\end{align*}
\]
Backwards sequenced assignment rule

The sequence rule can already be applied bottom, but requires us to guess an assertion $R$:

$$
\vdash \{P\} \ C_1 \ \{R\} \quad \vdash \{R\} \ C_2 \ \{Q\}
$$

$$
\vdash \{P\} \ C_1; C_2 \ \{Q\}
$$

In the case of a command sequenced before an assignment, we can avoid having to guess $R$ with the sequenced assignment rule:

$$
\vdash \{P\} \ C \ \{Q[E/V]\}
$$

$$
\vdash \{P\} \ C; V := E \ \{Q\}
$$

This is easily derivable using the sequencing rule and the backwards-assignment rule (exercise).
In the same way, we can derive a backwards-reasoning rule for loops by building in consequence:

\[ \vdash P \Rightarrow I \quad \vdash \{ I \land B \} \ C \ \{ I \} \quad \vdash I \land \neg B \Rightarrow Q \]

\[ \vdash \{ P \} \ \textbf{while} \ B \ \textbf{do} \ C \ \{ Q \} \]

This rule still requires us to guess \( I \) to apply it bottom-up.
Proof rules

\[ \vdash P \Rightarrow Q \quad \vdash \{P\} \text{ skip } \{Q\} \]
\[ \vdash \{P\} C_1 \{R\} \quad \vdash \{R\} C_2 \{Q\} \]
\[ \vdash \{P\} C_1; C_2 \{Q\} \]

\[ \vdash P \Rightarrow Q[E/V] \quad \vdash \{P\} \; V := E \; \{Q\} \]
\[ \vdash \{P\} \; C \; \{Q[E/V]\} \quad \vdash \{P\} \; C; \; V := E \; \{Q\} \]

\[ \vdash P \Rightarrow I \quad \vdash \{I \land B\} \; C \; \{I\} \quad \vdash I \land \neg B \Rightarrow Q \]
\[ \vdash \{P\} \; \text{while } B \; \text{do } C \; \{Q\} \]

\[ \vdash \{P \land B\} \; C_1 \; \{Q\} \quad \vdash \{P \land \neg B\} \; C_2 \; \{Q\} \]
\[ \vdash \{P\} \; \text{if } B \; \text{then } C_1 \; \text{else } C_2 \; \{Q\} \]
Finding loop invariants
We wish to verify that the following command computes the factorial of $X$ and stores the result in $Y$.

```
while $X \neq 0$ do (Y := Y * X; X := X - 1)
```

First we need to formalise the specification:

- Factorial is only defined for non-negative numbers, so $X$ should be non-negative in the initial state.
- The terminal state of $Y$ should be equal to the factorial of the initial state of $X$.
- The implementation assumes that $Y$ is equal to 1 initially.
This corresponds to the following partial correctness Hoare triple:

$$\{ X = x \land X \geq 0 \land Y = 1 \}$$

while $$X \neq 0$$ do (\(Y := Y \ast X; X := X - 1\))

$$\{ Y = x! \}$$

Here \(!\) denotes the usual mathematical factorial function.

Note that we used an auxiliary variable \(x\) to record the initial value of \(X\) and relate the terminal value of \(Y\) with the initial value of \(X\).
How does one find an invariant?

Here $I$ is an invariant that

- must hold initially
- must be preserved by the loop body when $B$ is true
- must imply the desired postcondition when $B$ is false

\[
\begin{align*}
\vdash P \Rightarrow I & \quad \vdash \{I \land B\} \ C \ \{I\} & \quad \vdash I \land \neg B \Rightarrow Q \\
\vdash \{P\} \textbf{ while } B \ \textbf{ do } C \ \{Q\}
\end{align*}
\]
How does one find an invariant?

\[ \vdash P \Rightarrow I \quad \vdash \{I \land B\} \; C \; \{I\} \quad \vdash I \land \neg B \Rightarrow Q \]

\[ \vdash \{P\} \textbf{ while } B \textbf{ do } C \; \{Q\} \]

The invariant \( I \) should express

- what \textbf{has been done so far} and what \textbf{remains to be done}
- that nothing has been done initially
- that nothing remains to be done when \( B \) is false
A verified factorial implementation

\{ X = x \land X \geq 0 \land Y = 1 \}

while \ X \neq 0 do (Y := Y \times X; X := X - 1)

\{ Y = x! \}

Take \ I \ to \ be \ Y \times X! = x! \land X \geq 0, \ then \ we \ must \ prove:

• \ X = x \land X \geq 0 \land Y = 1 \Rightarrow I
• \ { I \land X \neq 0 } Y := Y \times X; X := X - 1 \ { I }\}
• \ I \land X = 0 \Rightarrow Y = x!

The first and last proof obligation follow by basic arithmetic.
Proof rules

\[ \vdash P \Rightarrow Q \]
\[ \vdash \{ P \} \text{skip} \{ Q \} \]
\[ \vdash \{ P \} C_1 \{ R \} \quad \vdash \{ R \} C_2 \{ Q \} \]
\[ \vdash \{ P \} C_1; C_2 \{ Q \} \]

\[ \vdash P \Rightarrow Q[E/V] \]
\[ \vdash \{ P \} V := E \{ Q \} \]
\[ \vdash \{ P \} C \{ Q[E/V] \} \]
\[ \vdash \{ P \} C; V := E \{ Q \} \]

\[ \vdash P \Rightarrow I \quad \vdash \{ I \land B \} C \{ I \} \quad \vdash I \land \neg B \Rightarrow Q \]
\[ \vdash \{ P \} \text{while } B \text{ do } C \{ Q \} \]

\[ \vdash \{ P \land B \} C_1 \{ Q \} \quad \vdash \{ P \land \neg B \} C_2 \{ Q \} \]
\[ \vdash \{ P \} \text{if } B \text{ then } C_1 \text{ else } C_2 \{ Q \} \]
In the literature, hand-written proofs in Hoare logic are often written as informal proof outlines instead of proof trees.

Proof outlines are code listings annotated with Hoare logic assertions between statements.
Here is an example of a proof outline for the second proof obligation for the factorial function:

\[
\begin{align*}
\{ Y \ast X! &= x! \land X \geq 0 \land X \neq 0 \} \\
\{ (Y \ast X) \ast (X - 1)! &= x! \land (X - 1) \geq 0 \} \\
Y &= Y \ast X; \\
\{ Y \ast (X - 1)! &= x! \land (X - 1) \geq 0 \} \\
X &= X - 1 \\
\{ Y \ast X! &= x! \land X \geq 0 \}
\end{align*}
\]
Writing out full proof trees or proof outlines by hand is tedious and error-prone even for simple programs.

In the next lecture we will look at using mechanisation to check our proofs and help discharge trivial proof obligations.
Imagine we want to prove the following fibonacci implementation satisfies the given specification.

\[ \{ X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n \} \]

\textbf{while} (Z < N) \textbf{do}

\hspace{1em} (Y := X + Y; X := Y - X; Z := Z + 1)

\[ \{ Y = \text{fib}(n) \} \]

First we need to understand the implementation:

- the Z variable is used to count loop iterations
- and Y and X are used to compute the fibonacci number
A verified fibonacci implementation

\[
\{ X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n \} \\
\text{while } (Z < N) \text{ do} \\
\quad (Y := X + Y; X := Y - X; Z := Z + 1) \\
\{ Y = \text{fib}(n) \}
\]

Take \( I \equiv Y = \text{fib}(Z) \land X = \text{fib}(Z - 1) \), then we have to prove:

- \( X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n \Rightarrow I \)
- \( \{ I \land (Z < N) \} \ Y := X + Y; X := Y - X; Z := Z + 1 \ \{ I \} \)
- \( (I \land \neg(Z < N)) \Rightarrow Y = \text{fib}(n) \)

Do all these hold?
A verified fibonacci implementation

\[
\begin{align*}
\{X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n\} \\
\textbf{while } (Z < N) \textbf{ do} \\
\quad (Y := X + Y; X := Y - X; Z := Z + 1) \\
\{Y = \text{fib}(n)\}
\end{align*}
\]

Take \( I \equiv Y = \text{fib}(Z) \land X = \text{fib}(Z - 1) \), then we have to prove:

- \( X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n \Rightarrow I \)
- \( \{I \land (Z < N)\} \ Y := X + Y; X := Y - X; Z := Z + 1 \ \{I\} \)
- \( (I \land \neg(Z < N)) \Rightarrow Y = \text{fib}(n) \)

Do all these hold? The first two do (Exercise!)
A verified fibonacci implementation

\[ \{ X = 0 \land Y = 1 \land Z = 1 \land 1 \leq N \land N = n \} \]

**while** \((Z < N)\) **do**

\((Y := X + Y; X := Y - X; Z := Z + 1)\)

\[ \{ Y = \text{fib}(n) \} \]

While \(Y = \text{fib}(Z) \land X = \text{fib}(Z - 1)\) is an invariant, it is not strong enough to establish the desired post-condition.

We need to know that when the loop terminates then \(Z = n\). We need to strengthen the invariant to:

\[ Y = \text{fib}(Z) \land X = \text{fib}(Z - 1) \land Z \leq N \land N = n \]
Total correctness
Total correctness

So far, we have many concerned ourselves with partial correctness. What about total correctness?

Recall, total correctness = partial correctness + termination.

The total correctness triple, $[P] C [Q]$ holds if and only if

- whenever $C$ is executed in a state satisfying $P$, then $C$ terminates and the terminal state satisfies $Q$
Total correctness

WHILE-commands are the only commands that might not terminate.

Except for the WHILE-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness.
The WHILE-rule is not sound for total correctness

\[\vdash \{\top\} \ X := X \ \{\top\} \quad \vdash \{\top \land \top\} \ X := X \ \{\top\}\]

\[\vdash \{\top\} \ \textbf{while true do} \ X := X \ \{\top \land \neg \top\} \quad \vdash \top \land \neg \top \Rightarrow \bot\]

\[\vdash \{\top\} \ \textbf{while true do} \ X := X \ \{\bot\}\]

If the WHILE-rule was sound for total correctness, then this would show that \textbf{while true do} \ X := X always terminates in a state satisfying \(\bot\).
Total correctness

We need an alternative total correctness WHILE-rule that ensures the loop always terminates.

The idea is to show that some non-negative quantity decreases on each iteration of the loop.

This decreasing quantity is called a variant.
Total correctness

In the rule below, the variant is $E$, and the fact that it decreases is specified with an auxiliary variable $n$

\[
\vdash [P \land B \land (E = n)] \quad C \quad [P \land (E < n)]
\]  
\[\vdash P \land B \Rightarrow E \geq 0 \]

\[\vdash [P] \quad \textbf{while} \quad B \quad \textbf{do} \quad C \quad [P \land \neg B] \]

The second hypothesis ensures the variant is non-negative.
Using the rule-of-consequence we can derive the following backwards-reasoning total correctness WHILE rule:

<table>
<thead>
<tr>
<th>⊢ P ⇒ I</th>
<th>⊢ I ∧ ¬B ⇒ Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ I ∧ B ⇒ E ≥ 0</td>
<td>⊢ [I ∧ B ∧ (E = n)] C [I ∧ (E &lt; n)]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>⊢ [P] while B do C [Q]</td>
</tr>
</tbody>
</table>
Consider the factorial computation we looked at before

\[ X = x \land X \geq 0 \land Y = 1 \]

while \( X \neq 0 \) do (\( Y := Y \times X; X := X - 1 \))

\[ Y = x! \]

By assumption \( X \) is non-negative and decreases in each iteration of the loop.

To verify that this factorial implementation terminates we can thus take the variant \( E \) to be \( X \).
Total correctness: Factorial example

\[ X = x \land X \geq 0 \land Y = 1 \]

while \( X \neq 0 \) do \( (Y := Y \times X; X := X - 1) \)

\[ Y = x! \]

Take \( I \) to be \( Y \times X! = x! \land X \geq 0 \) and \( E \) to be \( X \).

Then we have to show that

- \( X = x \land X \geq 0 \land Y = 1 \Rightarrow I \)
- \( [I \land X \neq 0 \land (X = n)] \quad Y := Y \times X; X := X - 1 \ [I \land (X < n)] \)
- \( I \land X = 0 \Rightarrow Y = x! \)
- \( I \land X \neq 0 \Rightarrow X \geq 0 \)
The relation between partial and total correctness is informally given by the equation

Total correctness = partial correctness + termination

This is captured formally by the following inference rules

\[ \vdash \{P\} C \{Q\} \quad \vdash [P] C [\top] \quad \vdash [P] C [Q] \quad \vdash \{P\} C \{Q\} \]
We have given rules for total correctness.

They are similar to those for partial correctness.

The main difference is in the WHILE-rule

- WHILE commands are the only ones that can fail to terminate
- for WHILE commands we must prove that a non-negative expression is decreased by the loop body