Hoare logic

Lecture 6: Extending Hoare logic

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2018/19
Recap

Last time, we looked at how separation logic enables modular reasoning.

In this lecture, we will consider extending Hoare logic in other directions:

- We will look at extending partial correctness triples to enforce termination, and at adapting the Hoare logic rules for partial correctness to total correctness.
- We will look at how to handle (a crude form of) procedures.
- We will look at how to reason about simple forms of concurrency.
Total correctness
So far, we have concerned ourselves only with partial correctness, and not with what happens when the program diverges.

However, in many contexts where we care about correctness enough to use Hoare logic for verification, we also care about termination.
There is no standard notation for total correctness triples; we will use $[P] \ C \ [Q]$.

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

- assuming $C$ is executed in an initial state satisfying $P$,
- then the execution terminates,
- and the terminal state satisfies $Q$. 
A total correctness triple asserts that when the given command is executed from an initial state that satisfies the precondition, then any execution must terminate, and that any terminal state satisfies the postcondition:

\[
\models [P] C [Q] \overset{\text{def}}{=} \forall s. \ s \in [P] \Rightarrow \left( \neg (\langle C, s \rangle \xrightarrow{\omega}) \land \forall s'. \langle C, s \rangle \rightarrow \langle \text{skip}, s' \rangle \land s' \in [Q] \right)
\]
Semantics of total correctness triples

Since WHILE is safe and deterministic, this is equivalent to

$$\forall s. s \in \llbracket P \rrbracket \Rightarrow \exists s'. \langle C, s \rangle \rightarrow \langle \text{skip}, s' \rangle \land s' \in \llbracket Q \rrbracket$$

Assume $s \in \llbracket P \rrbracket$ and $\langle C, s \rangle \rightarrow^* \langle \text{skip}, s' \rangle$.

Since WHILE is safe and deterministic, $\neg(\langle C, s \rangle \rightarrow^\omega)$. Moreover, since WHILE is deterministic, for all $s''$ such that $\langle C, s \rangle \rightarrow^* \langle \text{skip}, s'' \rangle$, $s'' = s'$, so $s'' \in \llbracket Q \rrbracket$. 

Examples of total correctness triples

- The following total correctness triple is valid:

\[
\vdash [X \geq 0] \text{while } X \geq 0 \text{ do } X := X - 1 \ [X = 0]
\]

the loop terminates when executed from an initial state where \( X \) is non-negative.

- The following total correctness triple is not valid:

\[
\not\vdash [\top] \text{while } X \geq 0 \text{ do } X := X - 1 \ [X = 0]
\]

the loop only terminates when executed from an initial state where \( X \) is non-negative, but not when executed from an initial state where \( X \) is negative.
Corner cases of total correctness triples

\([P] \ C \ [\top]\)

- this says that \(C\) always terminates when executed from an initial state satisfying \(P\).

\([\top] \ C \ [Q]\)

- this says that \(C\) always terminates, and ends up in a state where \(Q\) holds.
Rules for total correctness

**while** commands are the commands that introduce non-termination.

Except for the loop rule, all the rules of Hoare logic (from the first lecture) are sound for total correctness as well as partial correctness.

\[
\begin{align*}
\vdash [P] \text{skip} [P] \\
\vdash [P] \ C_1 \ [Q] & \quad \vdash [Q] \ C_2 \ [R] \\
\hline
\vdash [P] \ C_1; \ C_2 \ [R] \\
\vdash [P \land B] \ C_1 \ [Q] & \quad \vdash [P \land \lnot B] \ C_2 \ [Q] \\
\hline
\vdash [P] \text{if } B \text{ then } C_1 \text{ else } C_2 \ [Q] \\
\vdash P_1 \Rightarrow P_2 & \quad \vdash [P_2] \ C \ [Q_2] \quad \vdash Q_2 \Rightarrow Q_1 \\
\hline
\vdash [P_1] \ C \ [Q_1]
\end{align*}
\]
The loop rule that we have for partial correctness is not sound for total correctness:

\[
\begin{align*}
\vdash (T \land T) \Rightarrow T & \\
\vdash \{T\} \text{ skip } \{T\} & \\
\vdash T \Rightarrow T & \\
\vdash \{T \land T\} \text{ skip } \{T\} & \\
\vdash \{T\} \text{ while } T \text{ do skip } \{T \land \neg T\} & \\
\vdash T \land \neg T \Rightarrow \bot & \\
\vdash \{T\} \text{ while } T \text{ do skip } \{\bot\}
\end{align*}
\]

If the loop rule were sound for total correctness, then this would show that \textbf{while } T \text{ do skip} always terminates in a state satisfying \( \bot \).
We need an alternative total correctness loop rule that ensures that the loop always terminates.

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

If this is the case, then the loop terminates, as there would otherwise be an infinite decreasing sequence of natural numbers.

This decreasing quantity is called a variant.
In the rule below, the variant is $t$, and the fact that it decreases is specified with an auxiliary variable $n$:

\[
\frac{
\vdash [P \land B \land (t = n)] \quad C [P \land (t < n)] 
\quad \vdash P \land B \Rightarrow t \geq 0
}{
\vdash [P] \textbf{while } B \textbf{ do } C [P \land \neg B]
}\]

The second hypothesis ensures that the variant is non-negative.

The variant $t$ does not have to occur in $C$. 
Using the rule of consequence, we can derive the following backwards reasoning total correctness loop rule:

\[
\begin{align*}
\vdash P \Rightarrow I & \quad \vdash I \land \neg B \Rightarrow Q \\
\vdash I \land B \Rightarrow t \geq 0 & \quad \vdash [I \land B \land (t = n)] C [I \land (t < n)] \\
\hline
\vdash [P] \textbf{while } B \textbf{ do } C [Q]
\end{align*}
\]
Consider the factorial computation we looked at before:

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

\[\text{while } X \neq 0 \text{ do } (Y := Y \times X; X := X - 1)\]

\[Y = x!\]

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 \(t\) to be \(X\).
Total correctness: factorial example

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

\[
\text{while } X \neq 0 \text{ do (} Y := Y \times X; X := X - 1 \text{)}
\]

\[
[Y = x!]
\]

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

Then we have to show that

- \( X = x \land X \geq 0 \land Y = 1 \Rightarrow I \)
- \([I \land X \neq 0 \land (X = n)] \ Y := Y \times X; X := X - 1 [I \land (X < n)] \)
- \( I \land \neg(X \neq 0) \Rightarrow Y = x! \)
- \( I \land X \neq 0 \Rightarrow X \geq 0 \)
Total correctness, partial correctness, and termination

Informally: total correctness = partial correctness + termination. This is captured formally by:

- If $\vdash \{P\} C \{Q\}$ and $\vdash [P] C [\top]$, then $\vdash [P] C [Q]$.
- If $\vdash [P] C [Q]$, then $\vdash \{P\} C \{Q\}$.

It is often easier to show partial correctness and termination separately.

Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of $X$:

```plaintext
while $X > 1$ do
    if ODD($X$) then $X := 3 \times X + 1$ else $X := X \div 2$
```

Microsoft’s T2 tool is used to prove termination of systems code.
We have given rules for total correctness, similar to those for partial correctness.

Only the loop rule differs: the premises of the loop rule require that the loop body decreases a non-negative expression.

It is even possible to do amortised, asymptotic complexity analysis in Hoare logic:

- A Fistful of Dollars, Armaël Guéneau et al., ESOP 2018
Functions (not examinable)
Consider an extension of our language with the following form of functions:

\[ C ::= \ldots | \textbf{let } F(\mathit{X}_1, \ldots, \mathit{X}_n) = \mathit{C}_1 \textbf{ in } \mathit{C}_2 | \mathit{F}(\mathit{X}_1, \ldots, \mathit{X}_n) \]

For this to work, we need to be careful to not have aliasing between program variables. We will elide this.

\[
\{ \mathit{X} = \mathit{x} \land \mathit{x} > 0 \} \\
\textbf{let } \mathit{F}(\mathit{X}, \mathit{N}) = \\
\hspace{1em} (\textbf{if } \mathit{X} > 1 \textbf{ then } (\mathit{X} := \mathit{X} - 1; \mathit{N} := \mathit{N} \times \mathit{X}; \mathit{F}(\mathit{X}, \mathit{N})) \textbf{ else skip}) \textbf{ in } \\
\hspace{1em} \mathit{N} := \mathit{X}; \\
\hspace{1em} \mathit{F}(\mathit{X}, \mathit{N}) \\
\{ \mathit{N} = \text{fact}(\mathit{x}) \}\]
We also need to extend our judgment $\vdash$ with a component $\mathcal{F}$ to keep track of the pre- and postconditions of functions:

\[
\mathcal{F}(F) = \langle P, Q \rangle \quad \ldots
\]

\[
\vdash_{\mathcal{F}} \{ P[Z_1/X_1, \ldots, Z_n/X_n] \} \ F(Z_1, \ldots, Z_n) \ \{ Q[Z_1/X_1, \ldots, Z_n/X_n] \}
\]

\[
\vdash_{\mathcal{F}[F \mapsto \langle P', Q' \rangle]} \{ P' \} \ C_1 \ \{ Q' \}
\]

\[
\vdash_{\mathcal{F}[F \mapsto \langle P', Q' \rangle]} \{ P \} \ C_2 \ \{ Q \} \quad \ldots
\]

\[
\vdash_{\mathcal{F}} \{ P \} \ \textbf{let} \ F(X_1, \ldots, X_n) = C_1 \ \textbf{in} \ C_2 \ \{ Q \}
\]
Recursive function pre- and postconditions are like loop invariants, but with a “gap”.

Concurrency (not examinable)
Concurrent composition

Consider an extension of our WHILE language with a concurrent composition construct (also “parallel composition”), $C_1 \parallel C_2$, which executes the two statements $C_1$ and $C_2$ concurrently.

For our simple form of concurrency, the statement $C_1 \parallel C_2$ reduces by interleaving execution steps of $C_1$ and $C_2$, until both have terminated:

\[
\langle C_1, \langle s, h \rangle \rangle \rightarrow \langle C'_1, \langle s', h' \rangle \rangle \\
\langle C_1 \parallel C_2, (s, h) \rangle \rightarrow \langle C'_1 \parallel C_2, \langle s', h' \rangle \rangle \\
\langle C_2, \langle s, h \rangle \rangle \rightarrow \langle C'_2, \langle s', h' \rangle \rangle \\
\langle C_1 \parallel C_2, \langle s, h \rangle \rangle \rightarrow \langle C_1 \parallel C'_2, \langle s', h' \rangle \rangle
\]

For instance, $(X := 0 \parallel X := 1); \text{print}(X)$ is allowed to print 0 or 1.

Final states are now of the form $F ::= \textbf{skip} | F_1 \parallel F_2$. 
Concurrency disciplines

Adding concurrency complicates reasoning by introducing the possibility of concurrent interference on shared state.

While separation logic does extend to reason about general concurrent interference, we will focus on two common idioms of concurrent programming with limited forms of interference:

- disjoint concurrency, and
- well-synchronised shared state.
Disjoint concurrency
Disjoint concurrency

Disjoint concurrency refers to multiple commands potentially executing concurrently, but all working on disjoint state.

Parallel implementations of divide-and-conquer algorithms can often be expressed using disjoint concurrency.

For instance, in a parallel merge sort, the recursive calls to merge sort operate on disjoint parts of the underlying array.
The proof rule for disjoint concurrency requires us to split our assertions into two disjoint parts, $P_1$ and $P_2$, and give each parallel command ownership of one of them:

$$\frac{\vdash \{P_1\} C_1 \{Q_1\} \quad \vdash \{P_2\} C_2 \{Q_2\}}{mod(C_1) \cap FV(P_2, Q_2) = \emptyset \quad mod(C_2) \cap FV(P_1, Q_1) = \emptyset \quad \vdash \{P_1 \ast P_2\} C_1 || C_2 \{Q_1 \ast Q_2\}}$$

The third hypothesis ensures that $C_1$ does not modify any program variables used in the specification of $C_2$, the fourth hypothesis ensures the symmetric.
Here is a simple example to illustrate two parallel increment operations that operate on disjoint parts of the heap:

\[
\begin{align*}
&\{X \mapsto 3 \ast Y \mapsto 4\} \\
\text{\text{\{\text{\{X \mapsto 3\}}}} & \quad \text{\text{\text{\{Y \mapsto 4\}}} \\
A := [X]; [X] := A + 1 & B := [Y]; [Y] := B + 1 \\
\{X \mapsto 4\} & \quad \{Y \mapsto 5\} \\
\text{\text{\text{\{X \mapsto 4 \ast Y \mapsto 5\}}}
\end{align*}
\]
Well-synchronised concurrency
Well-synchronised shared state refers to the common concurrency idiom of using locks to ensure exclusive access to state shared between multiple threads.

To reason about locking, concurrent separation logic extends separation logic with **lock invariants** that describe the resources protected by locks.

When acquiring a lock, the acquiring thread takes ownership of the lock invariant and when releasing the lock, must give back ownership of the lock invariant.
To illustrate, consider a simplified setting with a single global lock. We write $I \vdash \{P\} C \{Q\}$ to indicate that we can derive the given triple assuming the lock invariant is $I$. We have the following rules:

\[
\begin{align*}
FV(I) = \emptyset \\
I \vdash \{\text{emp}\} \text{lock} \{I \ast \text{locked}\} & \quad FV(I) = \emptyset \\
I \vdash \{I \ast \text{locked}\} \text{unlock} \{\text{emp}\}
\end{align*}
\]

The \textit{locked} resource ensures the lock can only be unlocked by the thread that currently has the lock.
Well-synchronised shared state example

To illustrate, consider a program with two threads that both access a number stored in shared heap cell at location $X$ concurrently.

Thread $A$ increments $X$ by 1 twice, and thread $B$ increments $X$ by 2. The threads use a lock to ensure their accesses are well-synchronised.

Assuming that location $X$ initially contains an even number, we wish to prove that the contents of location $X$ is still even after the two concurrent threads have terminated.

A non-synchronised interleaving would allow $X$ to end up odd.
First, we need to define a lock invariant.

The lock invariant needs to own the shared heap cell at location $X$ and should express that it always contains an even number:

$$I \equiv \exists n. x \mapsto 2 \times n$$

We have to use an indirection through $X = x$ because $I$ is not allowed to mention program variables.
We can temporarily violate the invariant when holding the lock.
Summary of concurrent separation logic

We have seen how concurrent separation logic supports reasoning about concurrent programs. The rule for disjoint concurrency enables reasoning about the parts of the state that are not shared, and the rules for locks enable reasoning about the parts of the state that are shared but guarded by locks.

Concurrent separation logic can also be extended to support reasoning about general concurrency interference.

Papers of historical interest:

- Peter O’Hearn. Resources, Concurrency and Local Reasoning.
Conclusion
• Verification of the seL4 microkernel assembly:
• The RustBelt project:
  https://plv.mpi-sws.org/rustbelt/
• The iGPS logic for relaxed memory concurrency:
  http://plv.mpi-sws.org/igps/
• The Iris higher-order concurrent separation logic framework,
  implemented and verified in a proof assistant:
  http://iris-project.org/
• Facebook’s bug-finding Infer tool:
  http://fbinfer.com/
We have seen that Hoare logic (separation logic, when we have pointers) enables specifying and reasoning about programs.

Reasoning remains close to the syntax, and captures the intuitions we have about why programs are correct.

It’s all about invariants!