Hoare logic

Lecture 6: Extending Hoare logic

Christopher Pulte cp526
University of Cambridge

CST Part II – 2022/2023

Recap

Last time, we looked at how separation logic enables modular reasoning about pointers and mutable data structures.

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) functions.
- We will look at how to reason about simple forms of concurrency.

Total correctness

Total correctness

So far, we have concerned ourselves only with partial correctness, and not with whether the program diverges.

However, in many contexts where we care about correctness enough to use Hoare logic for verification, we also care about termination.

Total correctness triples

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.

Semantics of total correctness triples

A total correctness triple asserts that when the given command is executed from an initial state that satisfies the precondition, then: every execution terminates and every terminal state satisfies the postcondition:

$$\models [P]\ C\ [Q] \stackrel{def}{=}$$

$$\forall s.\, s \in [\![P]\!] \Rightarrow \left(\begin{array}{l} \neg(\langle C, s\rangle \rightarrow^{\omega}) \wedge \\ (\forall s'.\, \langle C, s\rangle \rightarrow^{*} \langle \textbf{skip}, s'\rangle \Rightarrow s' \in [\![Q]\!]) \end{array} \right)$$

Semantics of total correctness triples. Updated wrt. handout

Since WHILE is **safe** and **deterministic**, this is equivalent to

$$\forall s.\, s \in [\![P]\!] \Rightarrow \exists s'.\, \langle C, s\rangle \rightarrow^* \langle \textbf{skip}, s'\rangle \wedge s' \in [\![Q]\!]$$

Assume $s \in [\![P]\!]$ and $\langle C, s\rangle \rightarrow^* \langle \textbf{skip}, s'\rangle$.

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

Examples of total correctness triples

- The following total correctness triple is valid:

$$\models [X \geq 0] \textbf{ while } X \neq 0 \textbf{ do } X := X - 1 \ [X = 0]$$

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

- The following total correctness triple is not valid:

$$\not\models [\top] \textbf{ while } X \neq 0 \textbf{ do } X := X - 1 \ [X = 0]$$

  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.

Both of the corresponding partial correctness triples hold.

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.

$[P]\ C\ [\bot]$

- this says that C always terminates when executed from an initial state satisfying P, and ends up in a state where $\bot$ holds, which means that no state can satisfy P.

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.

$$\frac{}{\vdash [P]\ \textbf{skip}\ [P]} \qquad \frac{}{\vdash [P[E/X]]\ X := E\ [P]}$$

$$\frac{\vdash [P]\ C_1\ [Q] \qquad \vdash [Q]\ C_2\ [R]}{\vdash [P]\ C_1; C_2\ [R]} \qquad \frac{\vdash [P \wedge B]\ C_1\ [Q] \qquad \vdash [P \wedge \neg B]\ C_2\ [Q]}{\vdash [P]\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ [Q]}$$

$$\frac{\vdash P_1 \Rightarrow P_2 \qquad \vdash [P_2]\ C\ [Q_2] \qquad \vdash Q_2 \Rightarrow Q_1}{\vdash [P_1]\ C\ [Q_1]}$$

Unsoundness of the partial correctness loop rule for total correctness

The loop rule that we have for partial correctness is not sound for total correctness:

$$\dfrac{\vdash_{FOL} \top \Rightarrow \top \qquad \dfrac{\dfrac{\vdots}{\vdash_{FOL} (\top \wedge \mathbf{T}) \Rightarrow \top} \qquad \dfrac{}{\vdash \{\top\}\ \mathbf{skip}\ \{\top\}} \qquad \dfrac{\vdots}{\vdash_{FOL} \top \Rightarrow \top}}{\vdash \{\top \wedge \mathbf{T}\}\ \mathbf{skip}\ \{\top\}} \qquad \dfrac{\vdots}{\vdash \top \wedge \neg \mathbf{T} \Rightarrow \bot}}{\vdash \{\top\}\ \mathbf{while\ T\ do\ skip}\ \{\top \wedge \neg \mathbf{T}\}} \qquad \dfrac{\vdots}{\vdash \top \wedge \neg \mathbf{T} \Rightarrow \bot}$$

$$\vdash \{\top\}\ \mathbf{while\ T\ do\ skip}\ \{\bot\}$$

If the loop rule were sound for total correctness, then this would show that **while T do skip** always terminates in a state satisfying $\bot$.

Loop variants

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

The idea is to require that on each iteration of the loop, some quantity that cannot decrease forever, the **variant**, decreases.

For example, there is no infinite descending chain of non-negative integers. We will restrict ourselves to non-negative integer variants.

Loop rule for total correctness

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

$$\frac{\vdash [P \wedge B \wedge (t = n)]\ C\ [P \wedge (t < n)] \qquad \vdash_{FOL} P \wedge B \Rightarrow t \geq 0}{\vdash [P]\ \textbf{while}\ B\ \textbf{do}\ C\ [P \wedge \neg B]}$$

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

The variant t does not have to occur in C.

Total correctness: factorial example

Consider the factorial computation we looked at before:

$$[X = x \wedge X \geq 0 \wedge Y = 1]$$
$$\textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1)$$
$$[Y = x!]$$

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

$$[X = x \wedge X \geq 0 \wedge Y = 1]$$
$$\textbf{while } X \neq 0 \textbf{ do } (Y := Y \times X; X := X - 1)$$
$$[Y = x!]$$

Take the invariant I to be $Y \times X! = x! \wedge X \geq 0$, and the variant t to be X.

Then we have to show that

- $\vdash_{FOL} (X = x \wedge X \geq 0 \wedge Y = 1) \Rightarrow I$
- $\vdash [I \wedge X \neq 0 \wedge (X = n)]\ Y := Y \times X; X := X - 1\ [I \wedge (X < n)]$
- $\vdash_{FOL} (I \wedge \neg(X \neq 0)) \Rightarrow Y = x!$
- $\vdash_{FOL} (I \wedge X \neq 0) \Rightarrow X \geq 0$

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.

Showing termination separately

Termination is usually straightforward to show, but there are examples where it is not.

For example, no one knows whether the program below terminates for all values of X:

while $X > 1$ **do**

  if $ODD(X)$ **then** $X := 3 \times X + 1$ **else** $X := X \ DIV \ 2$

(The Collatz conjecture is that this terminates with $X = 1$.)

Microsoft's T2 tool is used to prove termination of systems code.

Summary of total correctness

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 variant.

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)

Functions

Consider an extension of our language with the following crude form of functions where arguments are passed by reference :

$$C ::= \ldots \mid \textbf{let } F(X_1, \ldots, X_n) = C_1 \textbf{ in } C_2 \mid F(X_1, \ldots, X_n)$$

$\{X = x \wedge x > 0\}$
let $F(X, N) =$
  (**if** $X > 1$ **then** $(X := X - 1; N := N \times X; F(X, N))$
  else skip) **in**
$N := X;$
$F(X, N)$
$\{N = x!\}$

Hoare Logic rules for functions

We need to extend our judgment $\vdash$ with a component $\mathcal{F}$ to keep track of the pre- and postconditions of functions:

$$\frac{\mathcal{F}(F) = \langle P, Q \rangle \qquad \ldots}{\vdash_{\mathcal{F}} \{P[Z_1/X_1, \ldots, Z_n/X_n]\}\ F(Z_1, \ldots, Z_n)\ \{Q[Z_1/X_1, \ldots, Z_n/X_n]\}}$$

$$\frac{\vdash_{\mathcal{F}[F \mapsto \langle P', Q' \rangle]} \{P'\}\ C_1\ \{Q'\} \qquad \vdash_{\mathcal{F}[F \mapsto \langle P', Q' \rangle]} \{P\}\ C_2\ \{Q\} \qquad \ldots}{\vdash_{\mathcal{F}} \{P\}\ \textbf{let}\ F(X_1, \ldots, X_n) = C_1\ \textbf{in}\ C_2\ \{Q\}}$$

We need to be careful to not have aliasing between program variables. We assume that the ... assumptions deal with that.

Verifying an example using functions

$$
\begin{array}{l}
\{X = x \wedge x > 0\} \\
\textbf{let } F(X, N) = \\
\quad \{X > 0 \wedge N = X \times \ldots \times x\} \\
\quad \left(\begin{array}{l}
\quad \textbf{if } X > 1 \textbf{ then} \\
\qquad \{X > 0 \wedge N = X \times \ldots \times x \wedge X > 1\} \\
\qquad \{X - 1 > 0 \wedge N \times (X - 1) = (X - 1) \times \ldots \times x\} \\
\qquad X := X - 1; \\
\qquad \{X > 0 \wedge N \times X = X \times \ldots \times x\} \\
\qquad N := N \times X; \\
\qquad \{X > 0 \wedge N = X \times \ldots \times x\} \\
\qquad F(X, N) \\
\qquad \{N = 1 \times \ldots \times x\} \\
\quad \textbf{else} \\
\qquad \{X > 0 \wedge N = X \times \ldots \times x \wedge \neg(X > 1)\} \\
\qquad \textbf{skip} \\
\qquad \{N = 1 \times \ldots \times x\}
\end{array} \right) \\
\quad \{N = 1 \times \ldots \times x\} \textbf{ in} \\
\{X = x \wedge x > 0\} \\
\{X > 0 \wedge X = X \times \ldots \times x\} \\
N := X; \\
\{X > 0 \wedge N = X \times \ldots \times x\} \\
F(X, N) \\
\{N = x!\}
\end{array}
$$

Summary of functions

Hoare triples are a natural fit for specifying and verifying functions.

Pre- and postconditions of recursive functions are like loop invariants, but with a "gap" between the entry and exit of the function (the only gap between an iteration of a loop and the next is the guard):

$$
\begin{array}{l}
\textbf{let } F(\ldots) = \\
\quad \{P\} \\
\quad \ldots \\
\quad \{P[\ldots/\ldots]\} \\
\quad F(\ldots) \\
\quad \{Q[\ldots/\ldots]\} \\
\quad \ldots \\
\quad \{Q\} \textbf{ in} \\
\ldots \\
\{P[\ldots/\ldots]\} \\
F(\ldots)
\end{array}
$$

Concurrency (not examinable)

Concurrent composition. Small update wrt handout

Consider an extension of our WHILE_p language with a concurrent composition construct (also "parallel composition"), $C_1 \,||\, C_2$.

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

$$\frac{\langle C_1, \langle s, h\rangle\rangle \to \langle C_1', \langle s', h'\rangle\rangle}{\langle C_1 \,||\, C_2, \langle s, h\rangle\rangle \to \langle C_1' \,||\, C_2, \langle s', h'\rangle\rangle}$$

$$\frac{\langle C_2, \langle s, h\rangle\rangle \to \langle C_2', \langle s', h'\rangle\rangle}{\langle C_1 \,||\, C_2, \langle s, h\rangle\rangle \to \langle C_1 \,||\, C_2', \langle s', h'\rangle\rangle}$$

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

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

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.

Disjoint concurrency

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: (taking $FV(C)$ for some command C to be the set of program variables mentioned by C)

$$\frac{\begin{array}{c}\vdash \{P_1\}\ C_1\ \{Q_1\} \qquad \vdash \{P_2\}\ C_2\ \{Q_2\} \\ mod(C_1) \cap FV(P_2, C_2, Q_2) = \emptyset \\ mod(C_2) \cap FV(P_1, C_1, Q_1) = \emptyset\end{array}}{\vdash \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}$$

The third hypothesis ensures that C_1 does not modify any program variables used by C_2 or its specification, the fourth hypothesis ensures the symmetric.

Disjoint concurrency example

Here is a simple example to illustrate two parallel increment operations that operate on disjoint parts of the heap:

$$\{X \mapsto 3 * Y \mapsto 4\}$$

$\{X \mapsto 3\}$	$\{Y \mapsto 4\}$
$A := [X]; [X] := A + 1$	$B := [Y]; [Y] := B + 1$
$\{X \mapsto 4\}$	$\{Y \mapsto 5\}$

$$\{X \mapsto 4 * Y \mapsto 5\}$$

Well-synchronised concurrency

Well-synchronised shared state

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.

Well-synchronised shared state

To illustrate, consider a simplified setting with a single global lock.

We write $\vdash_I \{P\}\ C\ \{Q\}$ to indicate that we can derive the given triple assuming the lock invariant is I. We have the following rules (where PV is the set of program variables in an assertion):

$$\frac{PV(I) = \emptyset}{\vdash_I \{emp\}\ \textbf{lock}\ \{I * locked\}} \qquad \frac{PV(I) = \emptyset}{\vdash_I \{I * locked\}\ \textbf{unlock}\ \{emp\}}$$

The *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 a shared heap cell at location X concurrently.

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

lock;	**lock**;
$A := [X]; [X] := A + 1;$	$C := [X]; [X] := C + 2;$
$B := [X]; [X] := B + 1;$	
unlock	**unlock**

Well-synchronised shared state example

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 being odd.

lock;	**lock**;
$A := [X]; [X] := A + 1;$	$C := [X]; [X] := C + 2;$
$B := [X]; [X] := B + 1;$	
unlock	**unlock**

Well-synchronised shared state example

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.

Well-synchronised shared state example. Updated wrt handout

$$\{X = x \wedge emp\}$$

$\{X = x \wedge emp\}$	$\{X = x \wedge emp\}$
lock;	**lock**;
$\{X = x \wedge (I * locked)\}$	$\{X = x \wedge (I * locked)\}$
$\{X = x \wedge ((\exists n.\, x \mapsto 2 \times n) * locked)\}$	
$A := [X]; [X] := A + 1;$	
$\{X = x \wedge ((\exists n.\, x \mapsto 2 \times n + 1) * locked)\}$	$C := [X]; [X] := C + 2;$
$B := [X]; [X] := B + 1;$	
$\{X = x \wedge ((\exists n.\, x \mapsto 2 \times n) * locked)\}$	
$\{X = x \wedge (I * locked)\}$	$\{X = x \wedge (I * locked)\}$
unlock	**unlock**
$\{X = x \wedge emp\}$	$\{X = x \wedge emp\}$

$$\{X = x \wedge emp\}$$

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

Overall summary

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**!