Hoare logic

Lecture 6: Examples in separation logic

Jean Pichon-Pharabod
University of Cambridge

CST Part II – 2017/18

A proof system for separation logic

Introduction

In the previous lecture, we saw how reasoning about pointers in Hoare logic was problematic, which motivated introducing separation logic. We looked at the concepts separation logic is based on, the new assertions that embody them, and the semantics of assertions and partial correctness triples in separation logic.

In this lecture, we will

- introduce a syntactic proof system for separation logic;
- use it to verify example programs, thereby illustrating the power of separation logic.

The lecture will be focused on partial correctness.

Separation logic

Separation logic inherits all the partial correctness rules from Hoare logic from the first lecture, and extends them with

- structural rules, including the frame rule;
- rules for each new heap-manipulating command.

As we saw last time, some of the rules that were admissible for plain Hoare logic, for example the rule of constancy, are no longer sound for separation logic.

We now want the rule of consequence to be able manipulate our extended assertion language, with our new assertions $P * Q$, $t_1 \mapsto t_2$, and *emp*, and not just first-order logic anymore.

The frame rule

The frame rule expresses that separation logic triples always preserve any assertion disjoint from the precondition:

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \mathit{mod}(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The second hypothesis ensures that the frame R does not refer to any program variables modified by the command C.

Why the frame rule matters

The frame rule is the core of separation logic.
As we saw last time, it builds in modularity and compositionality.

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \mathit{mod}(C) \cap FV(R) = \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

It is so central to separation logic that its soundness is built in the definition of the semantics of separation logic triples, making it sound by construction.

Other structural rules

Given the rules that we are going to consider for the heap-manipulating commands, we are going to need to include structural rules like the following:

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{\exists v.\, P\}\ C\ \{\exists v.\, Q\}}$$

$$\vdots$$

Rules like these were admissible in Hoare logic.

The heap assignment rule

Separation logic triples must assert ownership of any heap cells modified by the command. The heap assignment rule thus asserts ownership of the heap location being assigned:

$$\frac{}{\vdash \{E_1 \mapsto t\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}}$$

If expressions are allowed to fault, we need a more complex rule.

The heap dereference rule

Separation logic triples must ensure the command does not fault. The heap dereference rule thus asserts ownership of the given heap location to ensure the location is allocated in the heap:

$$\frac{}{\vdash \{E \mapsto u \wedge V = v\}\ V := [E]\ \{E[v/V] \mapsto u \wedge V = u\}}$$

Here, u and v are auxiliary variables, and v is used to refer to the initial value of program variable V in the postcondition.

Allocation and deallocation

The allocation rule introduces a new points-to assertion for each newly allocated location:

$$\frac{}{\vdash \{V = v\}\ V := \textbf{alloc}(E_0, ..., E_n)\ \{V \mapsto E_0[v/V], ..., E_n[v/V]\}}$$

The deallocation rule destroys the points-to assertion for the location to not be available anymore:

$$\frac{}{\vdash \{E \mapsto t\}\ \textbf{dispose}(E)\ \{emp\}}$$

Swap example

Specification of swap

To illustrate these rules, consider the following code snippet:

$$C_{swap} \equiv A := [X]; B := [Y]; [X] := B; [Y] := A;$$

We want to show that it swaps the values in the locations referenced by X and Y, when X and Y do not alias:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ C_{swap}\ \{X \mapsto n_2 * Y \mapsto n_1\}$$

Proof outline for swap

$\{X \mapsto n_1 * Y \mapsto n_2\}$

$A := [X];$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$

$B := [Y];$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\}$

$[X] := B;$

$\{(X \mapsto B * Y \mapsto n_2) \wedge A = n_1 \wedge B = n_2\}$

$[Y] := A;$

$\{(X \mapsto B * Y \mapsto A) \wedge A = n_1 \wedge B = n_2\}$

$\{X \mapsto n_2 * Y \mapsto n_1\}$

Justifying these individual steps is now considerably more involved than in Hoare logic.

Detailed proof outline fo the first triple of swap

$\{X \mapsto n_1 * Y \mapsto n_2\}$

$\{\exists a.\, ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = a)\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = a\}$

$\{(X \mapsto n_1 \wedge A = a) * Y \mapsto n_2\}$

$\{X \mapsto n_1 \wedge A = a\}$

$A := [X]$

$\{X[a/A] \mapsto n_1 \wedge A = n_1\}$

$\{X \mapsto n_1 \wedge A = n_1\}$

$\{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$

$\{\exists a.\, ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)\}$

$\{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$

For reference: proof of the first triple of swap

Put another way:

To prove this first triple, we use the heap dereference rule to derive:

$$\{X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{X[a/A] \mapsto n_1 \wedge A = n_1\}$$

Then we existentially quantify the auxiliary variable a:

$$\{\exists a.\, X \mapsto n_1 \wedge A = a\}\ A := [X]\ \{\exists a.\, X[a/A] \mapsto n_1 \wedge A = n_1\}$$

Applying the rule of consequence, we obtain:

$$\{X \mapsto n_1\}\ A := [X]\ \{X \mapsto n_1 \wedge A = n_1\}$$

Since $A := [X]$ does not modify Y, we can frame on $Y \mapsto n_2$:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ A := [X]\ \{(X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2\}$$

Lastly, by the rule of consequence, we obtain:

$$\{X \mapsto n_1 * Y \mapsto n_2\}\ A := [X]\ \{(X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1\}$$

Proof of the first triple of swap (continued)

We relied on many properties of our assertion logic.

For example, to justify the first application of consequence, we need to show that

$$\vdash P \Rightarrow \exists a.\, (P \wedge A = a)$$

and to justify the last application of consequence, we need to show that:

$$\vdash ((X \mapsto n_1 \wedge A = n_1) * Y \mapsto n_2) \Rightarrow ((X \mapsto n_1 * Y \mapsto n_2) \wedge A = n_1)$$

Properties of separation logic assertions

Syntax of assertions in separation logic

We now have an extended language of assertions, with a new connective, the separating conjunction $*$:

$$\begin{array}{rcll} P, Q & ::= & \bot \mid \top \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q & \\ & \mid & P * Q \mid emp & \\ & \mid & \forall v.\, P \mid \exists v.\, P \mid t_1 = t_2 \mid p(t_1, \ldots, t_n) & n \geq 0 \end{array}$$

$\mapsto$ is a predicate symbol of arity 2.

This is not just usual first-order logic anymore: this is an instance of the classical first-order logic of bunched implication (which is related to linear logic).
We will also require inductive predicates later.

We will take an informal look at what kind of properties hold and do not hold in this logic. Using the semantics, we can prove the properties we need as we go.

Properties of separating conjunction

Separating conjunction is a commutative and associative operator with *emp* as a neutral element (like $\wedge$ was with $\top$):

$$\vdash P * Q \Leftrightarrow Q * P$$
$$\vdash (P * Q) * R \Leftrightarrow P * (Q * R)$$
$$\vdash P * emp \Leftrightarrow P$$

Separating conjunction is monotone with respect to implication:

$$\frac{\vdash P_1 \Rightarrow Q_1 \qquad \vdash P_2 \Rightarrow Q_2}{\vdash P_1 * P_2 \Rightarrow Q_1 * Q_2}$$

Separating conjunction distributes over disjunction:

$$\vdash (P \vee Q) * R \Leftrightarrow (P * R) \vee (Q * R)$$

Properties of separating conjunction (continued)

Separating conjunction semi-distributes over conjunction (but not the other direction in general):

$$\vdash (P \wedge Q) * R \Rightarrow (P * R) \wedge (Q * R)$$

In classical separation logic, $\top$ is not a neutral element for the separating conjunction: we only have

$$\vdash P \Rightarrow P * \top$$

but not the other direction in general. This means that we cannot “forget” about allocated locations (this is where classical separation logic differs from intuitionistic separation logic): we have $\vdash P * Q \Rightarrow P * \top$, but not $\vdash P * Q \Rightarrow P$ in general. To actually get rid of Q, we have to deallocate the corresponding locations.

Properties of pure assertions

An assertion is **pure** when it does not talk about the heap. Syntactically, this means it does not contain *emp* or $\mapsto$.

Separating conjunction and conjunction become more similar when they involve pure assertions:

$$\begin{array}{ll}
\vdash P \wedge Q \Rightarrow P * Q & \text{when } P \text{ or } Q \text{ is pure} \\
\vdash P * Q \Rightarrow P \wedge Q & \text{when } P \text{ and } Q \text{ are pure} \\
\vdash (P \wedge Q) * R \Leftrightarrow P \wedge (Q * R) & \text{when } P \text{ is pure}
\end{array}$$

Axioms for the points-to assertion

null cannot point to anything:

$$\vdash \forall t_1, t_2.\, t_1 \mapsto t_2 \Rightarrow (t_1 \mapsto t_2 * t_1 \neq \mathbf{null})$$

locations combined by $*$ are disjoint:

$$\vdash \forall t_1, t_2, t_3, t_4.\, (t_1 \mapsto t_2 * t_3 \mapsto t_4) \Rightarrow (t_1 \mapsto t_2 * t_3 \mapsto t_4 * t_1 \neq t_3)$$

$$\vdots$$

Assertions in separation logic are not freely duplicable in general: we do not have $\vdash P \Rightarrow P * P$ for all P. Therefore, in general, we need to repeat the assertions on the right-hand side of the implication to not "lose" them.

Verifying abstract data types

Verifying ADTs

Separation logic is very well-suited for specifying and reasoning about data structures typically found in standard libraries such as lists, queues, stacks, etc.

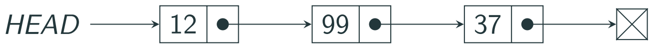
To illustrate this, we will specify and verify a library for working with lists, implemented using null-terminated singly-linked lists, in separation logic.

A list library implemented using singly-linked lists

First, we need to define a memory representation for our lists.

We will use null-terminated singly-linked list, starting from some designated *HEAD* program variable that refers to the first element of the linked list.

For instance, we will represent the mathematical list $[12, 99, 37]$ as we did in the previous lecture:



Representation predicates

To formalise the memory representation, separation logic uses **representation predicates** that relate an abstract description of the state of the data structure with its concrete memory representations.

For our example, we want a predicate $\mathit{list}(t, \alpha)$ that relates a mathematical list, α, with its memory representation starting at location t (here, $\alpha, \beta, \dots$ are just terms, but we write them differently to clarify the fact that they refer to mathematical lists).

To define such a predicate formally, we need to extend the assertion logic to reason about inductively defined predicates. We probably also want to extend it to reason about mathematical lists directly rather than through encodings. We will elide these details.

Representation predicates

We are going to define the $\mathit{list}(t, \alpha)$ predicate by induction on the list α:

- The empty list [] is represented as a **null** pointer:

$$\mathit{list}(t, []) \overset{def}{=} t = \textbf{null}$$

- The list $h :: \alpha$ (again, h is just a term) is represented by a pointer to two consecutive heap cells that contain the head h of the list and the location of the representation of the tail α of the list, respectively:

$$\mathit{list}(t, h :: \alpha) \overset{def}{=} \exists y.\, t \mapsto h * (t + 1) \mapsto y * \mathit{list}(y, \alpha)$$

(recall that $t \mapsto h \Rightarrow (t \mapsto h * t \neq \textbf{null})$)

Representation predicates

The representation predicate allows us to specify the behaviour of the list operations by their effect on the abstract state of the list.

For example, assuming that we represent the mathematical list α at location *HEAD*, we can specify a push operation C_{push} that pushes the value of program variable X onto the list in terms of its behaviour on the abstract state of the list as follows:

$$\{\mathit{list}(\mathit{HEAD}, \alpha) \land X = x\}\ C_{push}\ \{\mathit{list}(\mathit{HEAD}, x :: \alpha)\}$$

Representation predicates

We can specify all the operations of the library in a similar manner:

$$\{emp\} \;\; C_{new} \;\; \{list(HEAD, [])\}$$

$$\left\{\begin{array}{l} list(HEAD, \alpha) \wedge \\ X = x \end{array}\right\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

$$\{list(HEAD, \alpha)\} \;\; C_{pop} \left\{\begin{array}{l} \left(\begin{array}{l} list(HEAD, []) \wedge \\ \alpha = [] \wedge ERR = 1 \end{array}\right) \vee \\ \left(\exists h, \beta. \begin{array}{l} \alpha = h :: \beta \wedge \\ list(HEAD, \beta) \wedge \\ RET = h \wedge ERR = 0 \end{array}\right) \end{array}\right\}$$

$$\{list(HEAD, \alpha)\} \; C_{delete} \; \{emp\}$$

$$\vdots$$

The *emp* in the postcondition of C_{delete} ensures that the locations of the precondition have been deallocated.

Implementation of *push*

The *push* operation stores the *HEAD* pointer into a temporary variable *Y* before allocating two consecutive locations for the new list element, storing the start-of-block location to *HEAD*:

$$C_{push} \;\equiv\; Y := HEAD; HEAD := \mathbf{alloc}(X, Y)$$

We wish to prove that C_{push} satisfies its intended specification:

$$\{list(HEAD, \alpha) \wedge X = x\} \; C_{push} \; \{list(HEAD, x :: \alpha)\}$$

(We could use $HEAD := \mathbf{alloc}(X, HEAD)$ instead.)

Proof outline for *push*

Here is a proof outline for the *push* operation:

$$\begin{array}{l}
\{list(HEAD, \alpha) \wedge X = x\} \\
Y := HEAD \\
\{list(Y, \alpha) \wedge X = x\} \\
HEAD := \mathbf{alloc}(X, Y) \\
\{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x\} \\
\{list(HEAD, X :: \alpha) \wedge X = x\} \\
\{list(HEAD, x :: \alpha)\}
\end{array}$$

For the **alloc** step, we frame off $list(Y, \alpha) \wedge X = x$.

For reference: detailed proof outline for the allocation

$$\begin{array}{l}
\{list(Y, \alpha) \wedge X = x\} \\
\{\exists z. (list(Y, \alpha) \wedge X = x) \wedge HEAD = z\} \\
\quad \{(list(Y, \alpha) \wedge X = x) \wedge HEAD = z\} \\
\quad \{(list(Y, \alpha) \wedge X = x) * HEAD = z\} \\
\qquad \{HEAD = z\} \\
HEAD := \mathbf{alloc}(X, Y) \\
\qquad \{HEAD \mapsto X[z/HEAD], Y[z/HEAD]\} \\
\qquad \{HEAD \mapsto X, Y\} \\
\quad \{(list(Y, \alpha) \wedge X = x) * HEAD \mapsto X, Y\} \\
\quad \{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x)\} \\
\{\exists z. (list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x)\} \\
\{(list(Y, \alpha) * HEAD \mapsto X, Y) \wedge X = x\}
\end{array}$$

Implementation of *delete*

The *delete* operation iterates down over the list, deallocating nodes until it reaches the end of the list.

$$
\begin{aligned}
C_{delete} \equiv\ & X := HEAD;\\
& \textbf{while } X \neq \textbf{null do}\\
& \quad (Y := [X+1]; \textbf{dispose}(X); \textbf{dispose}(X+1); X := Y)
\end{aligned}
$$

We wish to prove that C_{delete} satisfies its intended specification:

$$\{list(HEAD, \alpha)\}\ C_{delete}\ \{emp\}$$

For that, we need a suitable loop invariant.
To execute safely, X effectively needs to point to **a** list (which is α only at the start).

Proof outline for *delete*

We can pick the invariant that we own the rest of the list:

$$
\begin{aligned}
& \{list(HEAD, \alpha)\}\\
& X := HEAD;\\
& \{list(X, \alpha)\}\\
& \{\exists \beta.\ list(X, \beta)\}\\
& \textbf{while } X \neq \textbf{null do}\\
& \quad \{\exists \beta.\ list(X, \beta) \land X \neq \textbf{null}\}\\
& \quad (Y := [X+1]; \textbf{dispose}(X); \textbf{dispose}(X+1); X := Y)\\
& \quad \{\exists \beta.\ list(X, \beta)\}\\
& \{\exists \beta.\ list(X, \beta) \land \neg(X \neq \textbf{null})\}\\
& \{emp\}
\end{aligned}
$$

We need to complete the proof outline for the body of the loop.

Proof outline for the loop body of *delete*

To verify the loop body, we need a lemma to unfold the list representation predicate in the non-null case:

$$
\begin{aligned}
& \{\exists \beta.\ list(X, \beta) \land X \neq \textbf{null}\}\\
& \{\exists h, y, \gamma.\ X \mapsto h, y * list(y, \gamma)\}\\
& Y := [X+1];\\
& \{\exists h, \gamma.\ X \mapsto h, Y * list(Y, \gamma)\}\\
& \textbf{dispose}(X); \textbf{dispose}(X+1);\\
& \{\exists \gamma.\ list(Y, \gamma)\}\\
& X := Y\\
& \{\exists \gamma.\ list(X, \gamma)\}\\
& \{\exists \beta.\ list(X, \beta)\}
\end{aligned}
$$

Classical separation logic and deallocation

Classical separation logic forces us to deallocate.

If we did not have the two deallocations in the body of the loop, we would have to do something with

$$X \mapsto h * X + 1 \mapsto Y$$

We can at best weaken that assertion to $\top$, but not fully eliminate it.

We could weaken our loop invariant to $\exists \beta.\ list(X, \beta) * \top$:
the $\top$ would indicate the memory leak.

Implementation of *max*

The *max* operation iterates over a non-empty list, computing its maximum element:

```
$C_{max} \equiv$
  $X := [HEAD + 1]; M := [HEAD];$
  $\textbf{while } X \neq \textbf{null do}$
    $(E := [X]; (\textbf{if } E > M \textbf{ then } M := E \textbf{ else skip}); X := [X + 1])$
```

We wish to prove that C_{max} satisfies its intended specification:

$$\{list(HEAD, h :: \alpha)\}\ C_{max}\ \{list(HEAD, h :: \alpha) * M = max(h :: \alpha)\}$$

For that, we need a suitable loop invariant.
The lists represented starting at *HEAD* and *X* are not disjoint.

Proof outline for *max*

We can define an auxiliary predicate $plist(t_1, \alpha, t_2)$ inductively:

$$plist(t_1, [], t_2) \overset{def}{=} (t_1 = t_2)$$

$$plist(t_1, h :: \alpha, t_2) \overset{def}{=} (\exists y.\ t_1 \mapsto h, y * plist(y, \alpha, t_2))$$

such that $list(t, \alpha +\!\!+ \beta) \Leftrightarrow \exists y.\ plist(t, \alpha, y) * list(y, \beta)$, and use it to express our invariant:

```
$\{list(HEAD, h :: \alpha)\}$
$X := [HEAD + 1]; M := [HEAD];$
$\{plist(HEAD, [h], X) * list(X, \alpha) * M = max([h])\}$
$\{\exists \beta, \gamma.\ h :: \alpha = \beta +\!\!+ \gamma * plist(HEAD, \beta, X) * list(X, \gamma) * M = max(\beta)\}$
$\textbf{while } X \neq \textbf{null do}$
  $(E := [X]; (\textbf{if } E > M \textbf{ then } M := E \textbf{ else skip}); X := [X + 1])$
$\{list(HEAD, h :: \alpha) * M = max(h :: \alpha)\}$
```

Summary of examples in separation logic

We can specify abstract data types using representation predicates which relate an abstract model of the state of the data structure with a concrete memory representation.

Justification of individual steps has to be made quite carefully given the unfamiliar interaction of connectives in separation logic, but proof outlines remain very readable.

Concurrency (not examinable)

Concurrent composition

Imagine extending our WHILE_p language with a concurrent composition construct (also "parallel composition"), $C_1||C_2$, which executes the two statements C_1 and C_2 concurrently.

The statement $C_1||C_2$ reduces by interleaving execution steps of C_1 and C_2, until both have terminated.

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

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:

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

The third hypothesis ensures that C_1 does not modify any program variables used in the specification of C_2, 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:

$$\begin{array}{c} \{X \mapsto 3 * Y \mapsto 4\} \\ \hline \begin{array}{c||c} \{X \mapsto 3\} & \{Y \mapsto 4\} \\ A := [X]; [X] := A + 1 & B := [Y]; [Y] := B + 1 \\ \{X \mapsto 4\} & \{Y \mapsto 5\} \end{array} \\ \hline \{X \mapsto 4 * Y \mapsto 5\} \end{array}$$

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 $I \vdash \{P\}\ C\ \{Q\}$ to indicate that we can derive the given triple assuming the lock invariant is I. We have the following rules:

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

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 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 being odd.

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

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

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

Concurrent separation logic supports disjoint concurrency by splitting resources into disjoint parts and distributing them to non-interacting commands.

Concurrent separation logic also supports reasoning about well-synchronised concurrent programs, using lock invariants to guard access to shared state.

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.

Perspectives

- Verification of the seL4 microkernel assembly: `https://entropy2018.sciencesconf.org/data/myreen.pdf`
- 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/`

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**!*

*Or recursive function pre- and postconditions, which amount to the same thing.

The heap assignment rule for when expressions can fault

$$\vdash \{E_1 \mapsto t_1 * E_2 = t_2\}\ [E_1] := E_2\ \{E_1 \mapsto E_2\}$$

It also requires that evaluating E_2 does not fault.
Exercise: Why is $E_1 = t_1$ not necessary in the precondition?