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

Introduction

In the previous lecture we saw the informal concepts that Separation Logic is based on.

This lecture will

- introduce a formal proof system for Separation logic
- present examples to illustrate the power of Separation logic

The lecture will be focused on partial correctness.

A proof system for Separation logic

Separation Logic

Separation logic inherits all the partial correctness rules from Hoare logic that you have already seen and extends them with

- the frame rule
- rules for each new heap-primitive

Some of the derived rules for plain Hoare logic no longer hold for separation logic (e.g., the rule of constancy).

The frame rule

The frame rule expresses that Separation logic triples always preserve any resources disjoint from the precondition.

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

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

The heap assignment rule

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

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

Here we use $E_1 \mapsto \_$ as shorthand for $\exists v.\, E_1 \mapsto v$.

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.

$$\vdash \{E \mapsto v \wedge X = x\}\ X := [E]\ \{E[x/X] \mapsto v \wedge X = v\}$$

Here the auxiliary variable x is used to refer to the initial value of X in the postcondition.

Separation logic

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

$$\vdash \{X = x\}\ X := \mathit{cons}(E_1, ..., E_n)\ \{X \mapsto E_1[x/X], ..., E_n[x/X]\}$$

The deallocation rule destroys the points-to assertion for the location to be deallocated:

$$\vdash \{E \mapsto \_\}\ \mathbf{dispose}(E)\ \{\mathit{emp}\}$$

Swap example

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:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ C_{swap}\ \{X \mapsto v_2 * Y \mapsto v_1\}$$

Swap example

Below is a proof-outline of the main steps:

$$\{X \mapsto v_1 * Y \mapsto v_2\}$$

$$A := [X];$$

$$\{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1\}$$

$$B := [Y];$$

$$\{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1 \wedge B = v_2\}$$

$$[X] := B;$$

$$\{X \mapsto B * Y \mapsto v_2 \wedge A = v_1 \wedge B = v_2\}$$

$$[Y] := A;$$

$$\{X \mapsto B * Y \mapsto A \wedge A = v_1 \wedge B = v_2\}$$

$$\{X \mapsto v_2 * Y \mapsto v_1\}$$

Swap example

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

$$\{X \mapsto v_1 \wedge A = a\}\ A := [X]\ \{X[a/A] \mapsto v_1 \wedge A = v_1\}$$

Then we existentially quantify the auxiliary variable a:

$$\{\exists a.\, X \mapsto v_1 \wedge A = a\}\ A := [X]\ \{\exists a.\, X[a/A] \mapsto v_1 \wedge A = v_1\}$$

Applying the rule-of-consequence we obtain:

$$\{X \mapsto v_1\}\ A := [X]\ \{X \mapsto v_1 \wedge A = v_1\}$$

Since $A := [X]$ does not modify Y we can frame on $Y \mapsto v_2$:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ A := [X]\ \{(X \mapsto v_1 \wedge A = v_1) * Y \mapsto v_2\}$$

Lastly, by the rule-of-consequence we obtain:

$$\{X \mapsto v_1 * Y \mapsto v_2\}\ A := [X]\ \{X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1\}$$

Swap example

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

$$\vdash (X \mapsto v_1 \wedge A = v_1) * Y \mapsto v_2 \Rightarrow X \mapsto v_1 * Y \mapsto v_2 \wedge A = v_1$$

To prove this we need proof rules for the new separation logic primitives.

Separation logic assertions

Separation conjunction is commutative and associative operator with *emp* as a neutral element:

$$\vdash P * Q \Leftrightarrow Q * P$$

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

$$\vdash P * emp \Leftrightarrow P$$

Separation conjunction is monotone with respect to implication:

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

Separation logic assertions

Separating conjunction distributes over disjunction and semi-distributes over conjunction:

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

Taking $R \equiv X \mapsto 1 \vee Y \mapsto 1$, $P \equiv X \mapsto 1$ and $Q \equiv X \mapsto 1$ yields a counterexample to distributivity over conjunction in the other direction:

$$\not\vdash (P * R) \wedge (Q * R) \Rightarrow (P \wedge Q) * R$$

Separation logic assertions

An assertion is **pure** if it does not contain *emp*, $\mapsto$ or $\hookrightarrow$.

Separation conjunction and conjunction collapses for pure assertions:

$\vdash P \wedge Q \Rightarrow P * Q$	when P or Q is pure
$\vdash P * Q \Rightarrow P \wedge Q$	when P and Q are pure
$\vdash (P \wedge Q) * R \Leftrightarrow P \wedge (Q * R)$	when P is pure

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.

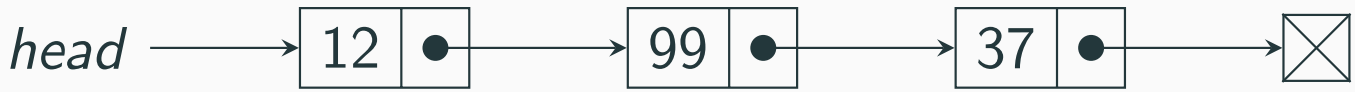
To illustrate we will specify and verify a library for working with linked lists in Separation Logic.

A linked list library

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

We will use a singly-linked list, starting from some designated *head* variable that refers to the first element of the list and terminating with a *null*-pointer.

For instance, we will represent a list containing the values $12, 99$ and 37 as follows



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 memory representations.

For our example, we want a predicate $\mathit{list}(\mathit{head}, \alpha)$ that relates a mathematical list, α, with its memory representation.

To define such a predicate formally, we need to extend the assertion logic to reason about mathematical lists, support for predicates and inductive definitions. We will elide these details.

Representation predicates

We are going to define the $\mathit{list}(\mathit{head}, \alpha)$ predicate by induction on the list α. We need to consider two cases: the empty list and an element x appended to a list β.

An empty list is represented as a *null*-pointer

$$\mathit{list}(\mathit{head}, []) \stackrel{\mathit{def}}{=} \mathit{head} = \mathit{null}$$

The list $x :: \beta$ is represented by a reference to two consecutive heap-cells that contain the value x and a representation of the rest of the list, respectively

$$\mathit{list}(\mathit{head}, x :: \beta) \stackrel{\mathit{def}}{=} \exists y.\, \mathit{head} \mapsto x * (\mathit{head} + 1) \mapsto y * \mathit{list}(y, \beta)$$

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

Imagine C_{push} is an implementation of an push operation that pushes the value stored in variable X to the front of the list referenced by variable *HEAD* and stores a reference to the new list in *HEAD*

We can specify this operation in terms of its behaviour on the abstract state of the list as follows

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

Representation predicates

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

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

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

$$\{list(HEAD, x :: \alpha)\} \quad C_{pop} \quad \{list(HEAD, \alpha) \wedge RET = x\}$$

$$\{list(HEAD, [])\} \quad C_{pop} \quad \{list(HEAD, []) \wedge RET = null\}$$

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

Implementation of *push*

The *push* operation stores the *HEAD* pointer pointer into a temporary variable Y before allocating two consecutive heap-cells for the new list element and updating *HEAP*:

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

We wish to prove it satisfies the following specification:

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

Proof outline for *push*

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

$$\{\mathit{list}(\mathit{HEAD}, \alpha) \wedge X = x\}$$
$$Y := \mathit{HEAD}$$
$$\{\mathit{list}(Y, \alpha) \wedge X = x\}$$
$$\mathit{HEAD} := \mathbf{cons}(X, Y)$$
$$\{\mathit{list}(Y, \alpha) * \mathit{HEAD} \mapsto X, Y \wedge X = x\}$$
$$\{\mathit{list}(\mathit{HEAD}, X :: \alpha) \wedge X = x\}$$
$$\{\mathit{list}(\mathit{HEAD}, x :: \alpha)\}$$

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

Implementation of *delete*

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

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

To prove that *delete* satisfies its intended specification,

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

we need a suitable invariant: that we own the rest of the list.

Proof outline for *delete*

$\{\mathit{list}(\mathit{HEAD}, \alpha)\}$

$X := \mathit{HEAD};$

$\{\mathit{list}(X, \alpha)\}$

$\{\exists \alpha.\, \mathit{list}(X, \alpha)\}$

while $X \neq \mathit{NULL}$ **do**

$\{\exists \alpha.\, \mathit{list}(X, \alpha) \wedge X \neq \mathit{NULL}\}$

$(Y := [X + 1]; \mathbf{dispose}(X); \mathbf{dispose}(X + 1); X := Y)$

$\{\exists \alpha.\, \mathit{list}(X, \alpha)\}$

$\{\mathit{list}(X, \alpha) \wedge \neg(X \neq \mathit{NULL})\}$

$\{\mathit{emp}\}$

Proof outline for loop-body of *delete*

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

$$\{\exists\alpha.\, \mathit{list}(X,\alpha) \wedge X \neq \mathit{NULL}\}$$
$$\{\exists v, t, \alpha.\, X \mapsto v, t * \mathit{list}(t,\alpha)\}$$
$$Y := [X+1];$$
$$\{\exists v, \alpha.\, X \mapsto v, Y * \mathit{list}(Y,\alpha)\}$$
$$\mathbf{dispose}(X); \mathbf{dispose}(X+1);$$
$$\{\exists\alpha.\, \mathit{list}(Y,\alpha)\}$$
$$X := Y$$
$$\{\exists\alpha.\, \mathit{list}(X,\alpha)\}$$

Concurrency (not examinable)

Concurrency

Imagine extending our WHILE_p language with a parallel composition construct, $C_1||C_2$, which executes the two statements C_1 and C_2 in parallel.

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

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

Concurrency

Adding parallelism 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**
- **well-synchronised shared state**

Disjoint concurrency

Disjoint concurrency refers to multiple commands potentially executing in parallel 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 resources into two disjoint parts, P_1 and P_2, and give each parallel command ownership of one of them.

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

The third hypothesis ensures C_1 does not modify any program variables used in the specification of C_2 and vice versa.

Disjoint concurrency example

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

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

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

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

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.

$$I \vdash \{emp\}\ \textbf{acquire}\ \{I * locked\}$$

$$I \vdash \{I * locked\}\ \textbf{release}\ \{emp\}$$

where I is not allowed to refer to any program variables.

The *locked* resource ensures the lock can only be released 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 in parallel.

Thread A increments the number by 2 and thread B multiplies the number by 10. The threads use a lock to ensure their accesses are well-synchronised.

Assuming x initially contains an even number, we wish to prove that x is still even after the two parallel threads have terminated.

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:

$$I \stackrel{def}{=} \exists v.\, x \mapsto v * even(v)$$

...

Assuming the lock invariant I is $\exists v.\, x \mapsto v * \mathit{even}(v)$, we have:

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

$\{X = x \wedge \mathit{emp}\}$		$\{X = x \wedge \mathit{emp}\}$
acquire;		**acquire**;
$\{X = x \wedge I * \mathit{locked}\}$		$\{X = x \wedge I * \mathit{locked}\}$
$A := [X]; [X] := A + 2;$	$\|\|$	$B := [X]; [X] := B * 10;$
$\{X = x \wedge I * \mathit{locked}\}$		$\{X = x \wedge I * \mathit{locked}\}$
release;		**release**;
$\{X = x \wedge \mathit{emp}\}$		$\{X = x \wedge \mathit{emp}\}$

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

Summary

Abstract data types are specified using representation predicates which relate an abstract model of the state of the data structure with a concrete memory representation.

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

Suggested reading:

- Peter O'Hearn. Resources, Concurrency and Local Reasoning.