Program logics, fourth lecture

Shared-memory concurrency: concurrent separation logic

Xavier Leroy
2021-03-25

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
Introduction:
Shared-memory parallel computing
Parallel computing

Use several processors (CPUs) together to perform a computation more quickly.

Two main models of parallel computing:

- **shared memory**
  - CPU CPU CPU CPU CPU
  - RAM
  - bus

- **distributed memory**
  - network
  - CPU
  - RAM

Many implementations that combine both models: multicore processors, multiprocessors, GPUs, clusters, grids, cloud computing, …
Milestones in parallel computing

1962  First symmetric multiprocessor: Burroughs D825 (1 to 4 CPUs sharing 1 to 16 memory modules).

1965  Start of the Multics project, the first modern operating system with multiprocessing support.

1973  Xerox PARC: Alto workstations + Ethernet network. First large distributed computation (image rendering).

1999  Launch of SETI@home and of Folding@home, two huge computations distributed over the Internet.

2006  First commonly-available multicore processors (Intel Core Duo and AMD Athlon 64 X2).

2012  (circa) All processors for PCs, tablets and smartphones are multicore.
Shared-memory concurrency

Features:

• Every processor has direct access to all the data.
• No need to duplicate data.
• Fast interprocess communications (through shared memory areas).

Challenges:

• Risk of interference between the actions of the processors.
• In particular: race conditions.
Race conditions

Several simultaneous accesses to the same memory location, including at least one write.

Case 1: two writes at the same time

\[
\text{set}(\ell, 1) \parallel \text{set}(\ell, 2)
\]

The program does not control which value ends up in location \( \ell \).

Case 2: one write and one read at the same time

\[
\text{set}(\ell, 1) \parallel \text{let} \; x = \text{get}(\ell)
\]

The program does not control which value is read in \( x \).
An example of race condition

\[ x := x + 1 \parallel x := x + 1 \]

Compiled to three instructions (read, compute, write):

\[
\begin{align*}
& \text{let } t = \text{get}(&x) \text{ in} \\
& \text{let } t = t + 1 \text{ in} \\
& \text{set}(&x, t) \\
& \text{let } t = \text{get}(&x) \text{ in} \\
& \text{let } t = t + 1 \text{ in} \\
& \text{set}(&x, t)
\end{align*}
\]
An example of race condition

\[ x := x + 1 \parallel x := x + 1 \]

One possible execution:

\[
\begin{align*}
\text{let } t &= \text{get}(&x) \text{ in} \\
\text{let } t &= t + 1 \text{ in} \\
\text{set}(&x, t)
\end{align*}
\]

With \( x = 0 \) initially, we end with \( x = 2 \).
An example of race condition

\[ x := x + 1 \parallel x := x + 1 \]

Another possible execution:

\[
\begin{align*}
\text{let } t &= \text{get}(&x) \text{ in} \\
\text{let } t &= t + 1 \text{ in} \\
\text{set}(&x, t) \\
\text{let } t &= \text{get}(&x) \text{ in} \\
\text{let } t &= t + 1 \text{ in} \\
\text{set}(&x, t)
\end{align*}
\]

With \( x = 0 \) initially, we end with \( x = 1 \).
A more realistic example

The “producer” part of a producer/consumer device: each process produces data $x$ and stores them in a shared buffer $T$ (an array of size $N$ indexed by $i$).

```plaintext
while $i \geq N$ do pause();
$T[i] := x;$
$i := i + 1;$
```
A more realistic example

With two producers in parallel:

$$\text{while } i \geq N \text{ do pause();}$$

$$T[i] := x_1;$$

$$i := i + 1;$$

$$\text{while } i \geq N \text{ do pause();}$$

$$T[i] := x_2; \quad \times$$

$$i := i + 1;$$

An out-of-bound array access is possible (if $i = N - 1$ initially).
A more realistic example

With two producers in parallel:

\[
\begin{align*}
\text{while } i \geq N & \text{ do pause();} \\
T[i] & := x_1; \\
i & := i + 1;
\end{align*}
\]

\[
\begin{align*}
\text{while } i \geq N & \text{ do pause();} \\
T[i] & := x_2; \\
i & := i + 1;
\end{align*}
\]

One of the two datum \(x_1, x_2\) is lost.

One entry of the buffer \((T[i - 1])\) is not initialized.
Synchronization using critical sections

In Java:

```java
synchronized (obj) {
    ...
}
```

In C:

```c
pthread_mutex_lock(mut);
...
pthread_mutex_unlock(mut);
```

Ensure **mutual exclusion**: at any time, at most one process is running inside the critical section.

Example: a well-synchronized producer.

```java
synchronized (buff) {
    while (buff.i >= N) buff.wait();
    buff.T [ buff.i ] = x;
    buff.i ++ ;
}
```
Synchronization and program logics

Many synchronization mechanisms:

- mutual exclusion: semaphores, locks, mutexes, ...
- barriers;
- message passing;
- atomic processor instructions (→ lock-free algorithms)

Which program logics to reason about interference and guarantee correct synchronization, in particular absence of race conditions?
Concurrency without resource sharing
Executing two commands in parallel

Commands:

\[ c := \ldots \]

| \[ c_1 \parallel c_2 \parallel \text{execute } c_1 \text{ and } c_2 \text{ in parallel} \]

Semantics:: an \textbf{interleaving} of the reductions of \( c_1 \) and \( c_2 \).

\[
\begin{align*}
(a_1 \parallel a_2)/h & \rightarrow 0/h \quad \text{(or any combination of } a_1 \text{ and } a_2) \\
(c_1 \parallel c_2)/h & \rightarrow (c'_1 \parallel c_2)/h' \quad \text{if } c_1/h \rightarrow c'_1/h' \\
(c_1 \parallel c_2)/h & \rightarrow (c_1 \parallel c'_2)/h' \quad \text{if } c_2/h \rightarrow c'_2/h' \\
(c_1 \parallel c_2)/h & \rightarrow \text{err} \quad \text{if } c_1/h \rightarrow \text{err} \text{ or } c_2/h \rightarrow \text{err}
\end{align*}
\]
Separation logic rule for parallel execution

\[
\begin{align*}
\{ P_1 \} \ c_1 \ \{ \lambda_. \ Q_1 \} \quad \{ P_2 \} \ c_2 \ \{ \lambda_. \ Q_2 \} \\
\{ P_1 \star P_2 \} \ c_1 \parallel c_2 \ \{ \lambda_. \ Q_1 \star Q_2 \}
\end{align*}
\]

Intuition:

- the initial heap \( h \) can be decomposed as \( h_1 \uplus h_2 \) with \( h_1 \) satisfying \( P_1 \) and \( h_2 \) satisfying \( P_2 \);
- \( c_1 \) executes in \( h_1 \) without modifying \( h_2 \);
- \( c_2 \) executes in \( h_2 \) without modifying \( h_1 \);
- the final states \( h_1', h_2' \) satisfy \( Q_1, Q_2 \) and are disjoint.
Separation logic rule for parallel execution

\[
\begin{align*}
\{ P_1 \} & \ c_1 \ \{ \_ \_ \ Q_1 \} & \{ P_2 \} & \ c_2 \ \{ \_ \_ \ Q_2 \} \\
\hline
\{ P_1 \star P_2 \} & \ c_1 \ || \ c_2 \ \{ \_ \_ \ Q_1 \star Q_2 \}
\end{align*}
\]

Alternate intuition: the precondition \( P_1 \star P_2 \) guarantees that the commands \( c_1 \) and \( c_2 \) execute without interference.
Therefore, the execution is equivalent to a sequential execution \( c_1; c_2 \) or \( c_2; c_1 \).

\[
\begin{align*}
\{ P_1 \} & \ c_1 \ \{ \_ \_ \ Q_1 \} & \{ P_2 \} & \ c_2 \ \{ \_ \_ \ Q_2 \} \\
\hline
\{ P_1 \star P_2 \} & \ c_1 \ \{ \_ \_ \ Q_1 \star P_2 \} & \{ Q_1 \star P_2 \} & \ c_2 \ \{ \_ \_ \ Q_1 \star Q_2 \} \\
\hline
\{ P_1 \star P_2 \} & \ c_1; c_2 \ \{ \_ \_ \ Q_1 \star Q_2 \}
\end{align*}
\]
Parallelism between disjoint sub-arrays

Example: Quicksort.

\[
\text{quicksort } T \ l \ h = \\
\text{if } h - l \leq 50 \text{ then} \\
\text{insertionsort } T \ l \ h \\
\text{else} \\
\text{let } m = \text{partition } T \ l \ h \text{ in} \\
\text{quicksort } T \ l \ m \ || \ \text{quicksort } T \ (m + 1) \ h
\]

\text{quicksort } T \ l \ h \text{ modifies the sub-array } T[l \ldots h] \text{ of } T.

The two recursive calls operate on disjoint sub-arrays: \( T[l \ldots m] \) and \( T[m + 1 \ldots h] \).

Therefore, we can do them in sequence as well as in parallel.
Parallelism between disjoint subtrees

\[
\begin{align*}
\text{tree}(\text{Leaf}, p) & = \langle p = \text{NULL} \rangle \\
\text{tree}(\text{Node}(t_1, x, t_2), p) & = \exists p_1, p_2, p \mapsto p_1 \star p + 1 \mapsto x \star p + 2 \mapsto p_2 \\
& \quad \star \text{tree}(t_1, p_1) \star \text{tree}(t_2, p_2)
\end{align*}
\]

The representation predicate guarantees that the two subtrees are disjoint, and can therefore be traversed and modified in parallel.

\[
\text{incrtree } t \delta =
\begin{align*}
\text{if } t & \neq \text{NULL} \text{ then } \\
\text{let } l & = \text{get}(t) \text{ and } n = \text{get}(t + 1) \text{ and } r = \text{get}(t + 2) \text{ in } \\
\text{set}(t + 1, n + \delta); \\
\text{incrtree } l \delta \parallel \text{incrtree } r \delta
\end{align*}
\]
We add one reduction rule that signals an error when a race condition occurs:

\[(c_1 \parallel c_2)/h \rightarrow \text{err} \quad \text{if} \quad \text{Acc}(c_1) \cap \text{Acc}(c_2) \neq \emptyset\]

Acc(c) is the set of memory locations that command c can read or write at the next reduction step:

\[
\begin{align*}
\text{Acc}(\text{get}(a)) &= \text{Acc}(\text{set}(a, a')) = \text{Acc}(\text{free}(a)) = \{a\} \\
\text{Acc(}\text{let } x = c_1 \text{ in } c_2\text{)} &= \text{Acc}(c_1) \\
\text{Acc}(c_1 \parallel c_2) &= \text{Acc}(c_1) \cup \text{Acc}(c_2)
\end{align*}
\]
Absence of race conditions

It is easy to show that

\[ c/h \not\rightarrow \text{err} \Rightarrow \text{Acc}(c) \subseteq \text{Dom}(h) \]

Therefore, if \( c_1/h_1 \not\rightarrow \text{err} \) and \( c_2/h_2 \not\rightarrow \text{err} \) and \( h_1 \perp h_2 \),

\[ \text{Acc}(c_1) \cap \text{Acc}(c_2) \subseteq \text{Dom}(h_1) \cap \text{Dom}(h_2) = \emptyset \]

and \((c_1 \parallel c_2)/(h_1 \uplus h_2)\) cannot reduce to \text{err} because of a race.

The semantic soundness proof (at the end of this lecture) formalizes this argument and shows that if \( \{ P \} \ c \ \{ Q \} \), the command \( c \) executes without race conditions.
Concurrency and resource sharing
The birth of concurrent separation logic

O’Hearn, Reynolds, Yang (2001), *Local Reasoning about Programs that Alter Data Structures*. The modern presentation of (sequential) separation logic.


Reynolds (2002), *Separation Logic: A Logic for Shared Mutable Data Structures*. Shows the rule for disjoint parallelism and mentions O’Hearn’s ongoing work.


A resource comprises

- one or several memory locations: global variables, dynamically-allocated objects;
- a lock or other mutual exclusion device that regulates access to the memory locations.

**Example (shared counter)**

```java
class Counter { int val; }
```

**Example (shared doubly-linked list)**

```java
class DList { DListCell first, last; }
class DListCell { Object data; DListCell prev, next; }
```
O’Hearn’s wonderful idea: a shared resource can be described by a separation logic assertion $A$.

- The footprint of $A$ defines the set of memory locations that belong to the resource.
- The assertion $A$ specifies the structure of these locations (e.g. “doubly-linked list”) and other relevant invariants.

**Example (shared counter $p$)**

$\exists n, \; p \mapsto n \star \langle n \geq 0 \rangle$

**Example (shared doubly-linked list $p, q$)**

$\exists x, y, w, \; p \mapsto x \star q \mapsto y \star dlist(w, x, y)$
Critical sections in separation logic

A shared resource $r$ is accessed only in a critical section

$$\text{with } r \text{ do } c$$

in mutual exclusion with the other processes.

Write $RI_r$ the assertion (the resource invariant) associated with $r$:

$$\{ \{ R I_r \ast P \} \ c \ \{ R I_r \ast Q \} \}$$

$$\{ \{ P \} \ \text{with } r \text{ do } c \ \{ Q \} \}$$

When entering the critical section, the process gains permission to use the memory locations of the resource, as described by $RI_r$.

Before leaving the critical section, the process must re-establish the invariant $RI_r$, because other processes are about to enter the critical section.
O’Hearn’s original article considers conditional critical sections

\[
\text{with } r \text{ when } b \text{ do } c
\]

where \( c \) is executed only when the condition \( b \) is true.

The rule for c.c.s. is

\[
\{ \langle b \rangle \star Rl_r \star P \} \ c \ \{ Rl_r \star Q \}
\]

\[
\{ P \} \ \text{with } r \text{ when } b \text{ do } c \ \{ Q \}
\]
Example: decrementing a shared counter

The invariant is \( R_{I_r} = \exists n, \ p \mapsto n \ast \langle n \geq 0 \rangle. \)

\[
\begin{aligned}
\text{with } r \text{ do} \\
\quad \text{let } n = \text{get}(p) \text{ in} \\
\quad \text{if } n > 0 \text{ then set}(p, n - 1) \\
\text{done}
\end{aligned}
\]
Example: insertion in a shared list

The invariant is $R I_r = \exists q, w, \ p \mapsto q \ast list(w, q)$.

\[
\{ \text{emp} \} \\
\]

with $r$ do

\[
\{ \exists q, w, \ p \mapsto q \ast list(w, q) \} \\
\]

let $q = \text{get}(p)$ in

\[
\{ p \mapsto q \ast \exists w, \ list(w, q) \} \\
\]

let $a = \text{cons}(x, q)$ in

\[
\{ a \mapsto x \ast a + 1 \mapsto q \ast p \mapsto q \ast \exists w, \ list(w, q) \} \\
\]

set($p, a$)

\[
\{ p \mapsto a \ast a \mapsto x \ast a + 1 \mapsto q \ast \exists w, \ list(w, q) \} \\
\Rightarrow \{ \exists q, w, \ p \mapsto q \ast list(w, q) \} \\
\]

done

\[
\{ \text{emp} \} \\
\]
Commands:

\[ c ::= \ldots \]

\[ | c_1 \parallel c_2 \] execute \( c_1 \) and \( c_2 \) in parallel

\[ | \text{atomic } c \] execute \( c \) in one uninterruptible step

A “super-critical” section: during the execution of \text{atomic } c, all other processes are blocked and perform zero computation steps.

Practical relevance:

- In case of time sharing on a monoprocessor: atomic section \( \approx \) block interrupts and prevent preemption
- A good model for the \text{atomic instructions} of the processor.
Modeling atomic instructions provided by the processor

Atomic swap and its special cases:

\[
\text{swap}(p, n) \overset{\text{def}}{=} \text{atomic}(\text{let } x = \text{get}(p) \text{ in } \text{set}(p, n); x)
\]

\[
\text{test\_and\_set}(p) \overset{\text{def}}{=} \text{swap}(p, 1)
\]

\[
\text{read\_and\_clear}(p) \overset{\text{def}}{=} \text{swap}(p, 0)
\]

Atomic increment / decrement:

\[
\text{fetch\_and\_add}(p, d) \overset{\text{def}}{=} \text{atomic}(\text{let } x = \text{get}(p) \text{ in } \text{set}(p, x + d); x)
\]

Compare and swap:

\[
\text{CAS}(p, x, n) \overset{\text{def}}{=} \text{atomic}(\text{let } c = \text{get}(p) \text{ in }
\text{if } c = x \text{ then } (\text{set}(p, n); 1) \text{ else } 0)
\]
Operational semantics for atomic sections

$$(\text{atomic } c)/h \rightarrow a/h' \quad \text{if } c/h \rightarrow^* a/h'$$

$$(\text{atomic } c)/h \rightarrow \text{err} \quad \text{if } c/h \rightarrow^* \text{err}$$

Note: atomic $c_1 \parallel$ atomic $c_2$ is equivalent to $c_1; c_2$ or $c_2; c_1$. There is no interleaving between the reduction steps of $c_1$ and those of $c_2$.

Note: if $c/h$ diverges, $(\text{atomic } c)/h$ is stuck.
In practice, $c$ contains no loops and always terminates.
A “triple” for concurrency with critical sections

\[ J \vdash \{ P \} c \{ Q \} \]

The assertion \( J \) is an invariant on the shared memory (accessible only inside atomic sections \( \text{atomic} \ c \)).

The precondition \( P \) and the postcondition \( Q \) describe the private memory for the command \( c \).
The rules for atomic sections

Executing an atomic section:

\[
\text{emp} \vdash \{ P \star J \} c \{ \lambda v. Q v \star J \}
\]

\[ J \vdash \{ P \} \text{atomic } c \{ Q \} \]

Sharing a resource \( J' \):

\[
J \star J' \vdash \{ P \} c \{ Q \}
\]

\[ J \vdash \{ P \star J' \} c \{ \lambda v. Q v \star J' \} \]

Framing the invariant:

\[
J \vdash \{ P \} c \{ Q \}
\]

\[ J \star J' \vdash \{ P \} c \{ Q \} \]
The rules for control structures (reminder)

\[
P \Rightarrow Q [a] \\
J \vdash \{ P \} a \{ Q \} \\
J \vdash \{ P \} c \{ R \} \quad \forall v, \ J \vdash \{ R v \} c'[x \leftarrow v] \{ Q \} \\
J \vdash \{ P \} \text{let } x = c \text{ in } c' \{ Q \} \\
J \vdash \{ \langle b \rangle \star P \} c_1 \{ Q \} \quad J \vdash \{ \langle \neg b \rangle \star P \} c_2 \{ Q \} \\
\{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{ Q \} \\
J \vdash \{ P_1 \} c_1 \{ \lambda_{=} . \ Q_1 \} \quad J \vdash \{ P_2 \} c_2 \{ \lambda_{=} . \ Q_2 \} \\
J \vdash \{ P_1 \star P_2 \} c_1 \parallel c_2 \{ \lambda_{=} . \ Q_1 \star Q_2 \}
The “small rules” for heap operations (reminder)

\[
\begin{align*}
J \vdash \{ \text{emp} \} \quad &\text{alloc}(N) \quad \{ \lambda l. l \mapsto _* \cdots _* l + N - 1 \mapsto _* \} \\
J \vdash \{ [a] \mapsto x \} \quad &\text{get}(a) \quad \{ \lambda v. \langle v = x \rangle \star [a] \mapsto x \} \\
J \vdash \{ [a] \mapsto _* \} \quad &\text{set}(a, a') \quad \{ \lambda v. [a] \mapsto [a'] \} \\
J \vdash \{ [a] \mapsto _* \} \quad &\text{free}(a) \quad \{ \lambda v. \text{emp} \}
\end{align*}
\]
The structural rules (watch out! there’s a catch!)

\[
J \vdash \{ P \} c \{ Q \}
\]

(frame)

\[
J \vdash \{ P \ast R \} c \{ \lambda v. Q v \ast R \}
\]

\[
\begin{align*}
P \Rightarrow P' & \quad J \vdash \{ P' \} c \{ Q' \} \quad \forall v, \ Q' v \Rightarrow Q v \\
J \vdash \{ P \} c \{ Q \}
\end{align*}
\]

(consequence)

\[
\begin{align*}
J \vdash \{ P \} c \{ Q \} & \quad J \vdash \{ P' \} c \{ Q' \} \\
\quad & \quad (disjunction) \\
J \vdash \{ P \lor P' \} c \{ \lambda v. Q v \lor Q' v \}
\end{align*}
\]

\[
\begin{align*}
J \text{ precise} & \quad J \vdash \{ P \} c \{ Q \} \quad J \vdash \{ P' \} c \{ Q' \} \\
\quad & \quad (conjunction) \\
J \vdash \{ P \land P' \} c \{ \lambda v. Q v \land Q' v \}
\end{align*}
\]
The conjunction rule and Reynold’s counterexample

Take $J = \text{true}$ (the assertion $\lambda h. \top \text{ true}$ for all heaps). Take $\text{one} = 1 \mapsto \_$. We have $\text{one} \star \text{true} \Rightarrow \text{true}$, hence

\[
\text{emp} \vdash \{ \text{one} \star \text{true} \} \ 0 \ \{ \lambda_. \text{emp} \star \text{true} \}
\]
\[
\text{emp} \vdash \{ \text{one} \star \text{true} \} \ 0 \ \{ \lambda_. \text{one} \star \text{true} \}
\]

and, by application of the atomic rule,

\[
J \vdash \{ \text{one} \} \text{ atomic} \ 0 \ \{ \lambda_. \text{emp} \}
\]
\[
J \vdash \{ \text{one} \} \text{ atomic} \ 0 \ \{ \lambda_. \text{one} \}
\]

If the conjunction rule was true for all $J$, we could conclude

\[
J \vdash \{ \text{one} \& \text{one} \} \text{ atomic} \ 0 \ \{ \lambda_. \text{emp} \& \text{one} \}
\]

yet the postcondition $\text{emp} \& \text{one}$ is always false.
Precise assertions

Intuitively: an assertion $P$ is precise if its memory footprint is uniquely defined.

Formally: if $P$ cuts a sub-heap $h_1$ out of a given heap $h$, this sub-heap is uniquely determined:

$$h = h_1 \uplus h_2 = h'_1 \uplus h'_2 \land P h_1 \land P h'_1 \Rightarrow h_1 = h'_1$$
### Examples of precise / imprecise assertions

<table>
<thead>
<tr>
<th>Precise assertions</th>
<th>Imprecise assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>emp</code></td>
<td><code>true</code></td>
</tr>
<tr>
<td><code>ℓ ↦ → _</code></td>
<td><code>∃ ℓ, ℓ ↦ → _</code></td>
</tr>
<tr>
<td><code>ℓ ↦ → v</code></td>
<td><code>∃ ℓ, ℓ ↦ → v</code></td>
</tr>
<tr>
<td><code>∃ v, ℓ ↦ → v ⊕ R(v)</code></td>
<td></td>
</tr>
<tr>
<td><code>P ⊕ Q</code></td>
<td><code>P ⊕ true</code></td>
</tr>
<tr>
<td><code>⟨b⟩ ⊕ P ∨ ⟨¬b⟩ ⊕ Q</code></td>
<td><code>emp ∨ ℓ ↦ → _</code></td>
</tr>
</tbody>
</table>

(assuming `P, Q, R(v)` to be precise)
Binary semaphores and applications
A binary semaphore = a memory location \( p \) containing 0 (meaning “busy”) or 1 (meaning “available”).

The operations \( P \) (take) and \( V \) (release):

\[
V(sem) = \text{atomic(set}(sem, 1))
\]
\[
P(sem) = \text{let } x = \text{swap}(sem, 0) \text{ in if } x = 1 \text{ then } 0 \text{ else } P(sem)
\]

where

\[
\text{swap}(p, n) = \text{atomic(let } x = \text{get}(p) \text{ in set}(p, n); x)
\]

Note: \( P(sem) \) is busy-waiting and can fail to terminate, but the loop is outside the atomic section.
The rules for binary semaphores

Let $RI$ be the assertion describing the resources associated with the semaphore. We assume $RI$ precise.

As invariant on the shared memory, take

$$J(sem, RI) \overset{\text{def}}{=} \exists n. \text{sem} \mapsto n \Star (\langle n = 0 \rangle \lor \langle n = 1 \rangle \Star RI)$$

that is: “if the semaphore is available, the resources $RI$ are in the shared memory”. We can then derive:

$$J(sem, RI) \vdash \{ RI \} V(sem) \{ \text{emp} \}$$
$$J(sem, RI) \vdash \{ \text{emp} \} P(sem) \{ RI \}$$

In other words: releasing $p$ is putting $RI$ in the shared memory, and taking $p$ is getting $RI$ from the shared memory.
Synchronization with a semaphore

Consider the assertion $RI = \exists n, x \mapsto n \ast \langle n \text{ premier} \rangle$, “variable $x$ contains a prime number”.

$$\{ \text{sem} \mapsto 0 \ast x \mapsto _- \}$$

$$\{ x \mapsto _- \}$$

set($x$, 53);

$$\{ x \mapsto 53 \} \Rightarrow \{ RI \}$$

$V(\text{sem})$

$$\{ \text{emp} \}$$

$P(\text{sem})$

let $n = \text{get}(x)$ in

$$\{ x \mapsto n \ast \langle n \text{ prime} \rangle \}$$

print($n$)

The $P$ and $V$ operations ensure that the right process never reads $x$ before the left process has initialized. They transfer the permission to access $x$ from the left process to the right process.
Consider the assertion \( RI = \exists p, x \mapsto p \star p \mapsto _ - \) 
“variable \( x \) points to a valid memory location”.

\[
\{ \text{sem} \mapsto 0 \star x \mapsto _ - \} \\
\{ x \mapsto _ - \} \\
\text{let } p = \text{alloc}(1) \text{ in} \\
\{ x \mapsto _ - \star p \mapsto _ - \} \\
\text{set}(x, p); \\
\{ x \mapsto p \star p \mapsto _ - \} \implies \{ RI \} \\
\text{V(sem)} \\
\{ \text{emp} \} \\
\text{P(sem);} \\
\{ RI \} \\
\text{let } p = \text{get}(x) \text{ in} \\
\{ x \mapsto p \star p \mapsto _ - \} \\
\text{free}(p) \\
\{ x \mapsto _ - \} \\
\{ \text{emp} \} \\
\]

The memory location that was allocated by the left process is transferred and safely deallocated by the right process.
Derivation of the rule for $P$

Recall the invariant on the shared memory:

$$ J(sem, RI) \overset{\text{def}}{=} \exists n. \, sem \mapsto n \star (\langle n = 0 \rangle \lor \langle n = 1 \rangle \star RI) $$

For $\text{swap}(sem, 0)$, we have the triple

$$ J(sem, RI) \vdash \{ emp \} \text{swap}(sem, 0) \{ \lambda n. \langle n = 0 \rangle \lor \langle n = 1 \rangle \star RI \} $$

$P(sem)$ iterates $\text{swap}(sem, 0)$ until the result is 1, hence

$$ J(sem, RI) \vdash \{ emp \} \, P(sem) \{ RI \} $$
Derivation of the rule for $V$

$$J(sem, RI) \overset{\text{def}}{=} \exists n. \; sem \mapsto n \star (\langle n = 0 \rangle \lor \langle n = 1 \rangle \star RI)$$

It suffices to show

$$\text{emp} \vdash \{ RI \star J(sem, RI) \} \; \text{set}(sem, 1) \; \{ sem \mapsto 1 \star RI \}$$

to obtain $\text{emp} \vdash \{ RI \star J(sem, RI) \} \; \text{set}(sem, 1) \; \{ J(sem, RI) \}$

and therefore $J(sem, RI) \vdash \{ RI \} \; V(sem) \; \{ \text{emp} \}$.

But we do not know the status of the semaphore (busy or available):

$$\text{emp} \vdash \{ RI \star sem \mapsto 0 \} \; \text{set}(sem, 1) \; \{ sem \mapsto 1 \star RI \} \; (\text{available})$$

$$\text{emp} \vdash \{ RI \star sem \mapsto 1 \star RI \} \; \text{set}(sem, 1) \; \{ sem \mapsto 1 \star RI \} \; (\text{busy})$$

In the second case, we need $RI \star RI \Rightarrow RI$, which is true if $RI$ is precise.
Implementing critical sections

We can use a semaphore as a lock: $P$ acquires the lock, $V$ releases the lock.

This gives a simple implementation of critical sections:

$$\text{with } r \text{ do } c \overset{\text{def}}{=} P(r); c; V(r)$$

where each critical section $r$ is identified by the location of a semaphore, initialized to 1.
If $RI_r$ is the resource invariant for $r$, the shared memory invariant is the conjunction of the invariants of the associated semaphores:

$$J_{\mathcal{R}} = \bigstar_{r \in \mathcal{R}} J(r, RI_r)$$

This implementation validates the rule for critical sections:

$${r \in \mathcal{R} \quad J_{\mathcal{R}\{r\}} \vdash \{ RI_r \star P \} \cdot \{ RI_r \star Q \}} \quad \frac{J_{\mathcal{R}} \vdash \{ P \} \text{ with } r \text{ do } \{ Q \}}{J_{\mathcal{R}} \vdash \{ P \} \text{ with } r \text{ do } \{ Q \}}$$
Implementing conditional critical sections

In our PTR language, the condition $c_b$ of a c.c.s. is necessarily a command that evaluates to a Boolean.

$$\text{with } r \text{ when } c_b \text{ do } c \overset{\text{def}}{=} P(r); \text{wait}(r, c_b); \ c; \ V(r)$$

where \textit{wait} is the following busy-waiting loop:

$$\text{wait}(r, c_b) = \text{let } b = c_b \text{ in}$$

$$\text{if } b \text{ then } 0 \text{ else } (V(r); P(r); \text{wait}(r, c_b))$$

We can derive the following rule:

$$r \in R$$

$$J_{R\setminus\{r\}} \vdash \{ RL_r \star P \} \ c_b \ \{ \lambda b. \langle b \rangle \star B \lor \langle \neg b \rangle \star RL_r \star P \}$$

$$J_{R\setminus\{r\}} \vdash \{ B \} \ c \ \{ RL_r \star Q \}$$

$$J_R \vdash \{ P \} \text{ with } r \text{ when } c_b \text{ do } c \ \{ Q \}$$
The producer/consumer device

A generalization of the “synchronization and resource transfer” example, where several resources are transferred one after the other.

```
while true do
  compute x;
  produce(x);
  done

while true do
  let y = consume() in
  use y
  done
```

The already produced but not yet consumed resources are stored in a buffer in shared memory.

Note: we can have several producer processes and several consumer processes running concurrently.
A solution with a buffer of size 1 and two semaphores

Three variables in shared memory:

- $b$: location of the buffer (one memory cell)
- $s_1$: a semaphore that is 1 when the buffer is full (the buffer contains a produced but not yet consumed datum)
- $s_0$: a semaphore that is 1 when the buffer is empty (contains no produced but not yet consumed datum)

Implementation:

$$\text{produce}(b, s_0, s_1, x) = P(s_0); \text{set}(b, x); V(s_1)$$

$$\text{consume}(b, s_0, s_1) = P(s_1); \text{let } x = \text{get}(b) \text{ in } V(s_0); x$$
Write $RI(x)$ the resource invariant associated with datum $x$.

Specification of *produce* and *consume*:

$$J(b) \vdash \{ RI(x) \} \text{produce}(b, s_0, s_1, x) \{ \text{emp} \}$$

$$J(b) \vdash \{ \text{emp} \} \text{consume}(b, s_0, s_1) \{ \lambda x. RI(x) \}$$

The verification goes through by taking $J$ as shared memory invariant:

$$J(b) \overset{\text{def}}{=} J(s_0, b \mapsto \_ ) \ast J(s_1, \exists x, b \mapsto x \ast RI(x))$$

In other words: when semaphore $s_0$ is 1, $b$ is valid (we can write into it); when semaphore $s_1$ is 1, $b$ contains a datum $x$ such that $RI(x)$ holds.
Semantic soundness
The original proof of Brookes (2004):

- Denotational semantics for commands, as action traces.
- A “local” semantics for actions and traces that identifies resource ownership and resource transfers at critical sections.
- An hypothesis: all resource invariants are precise.

The simplified proof of Vafeiadis (2011):

- Direct, elementary reasoning about reduction sequences, using a step-indexed predicate $\text{Safe}^n c h$.
- The conjunction rule is the only one that demands precise resource invariants.
Some intuitions

\[ J \vdash \{ P \} c \{ Q \} \]

Deductive intuition: it’s like \( \{ P \star J \} c \{ Q \star J \} \)
plus invariance of \( J \), that is, all triples appearing in the derivation
have the shape above.

Operational intuition: at every step of the evaluation, the current
heap \( h \) decomposes in three disjoint parts:

\[ h = h_1 \uplus h_j \uplus h_f \]

\( h_1 \) is the private memory for \( c \).
\( h_j \) is the shared memory accessible to atomic sections.
\( h_f \) is the “frame” memory, including the private memories of the
processes that execute in parallel with \( c \).
A weak semantic triple with step indexing

Define the semantic triple \( J \models \{\{P\}\} c \{\{Q\}\} \) by

\[
J \models \{\{P\}\} c \{\{Q\}\} \overset{\text{def}}{=} \forall n, h, P h \Rightarrow \text{Safe}^n c h Q J
\]

The inductive predicate \( \text{Safe}^n c h Q J \) means that the executions of \( c \) in the private memory \( h \)
– do not cause errors in the first \( n \) execution steps;
– satisfy \( Q \) if they terminate in at most \( n \) steps;
– preserve the shared-memory invariant \( J \).

\[
\begin{align*}
\text{Safe}^0 c h Q J & \quad Q [a] h \\
\text{Safe}^{n+1} a h Q J & \quad (\forall a, c \neq a) \quad \cdots \\
\text{Safe}^{n+1} c h Q J & \quad \cdots 
\end{align*}
\]
A weak semantic triple with step indexing

∀a, c ≠ a

∀h_j, h_f, J h_j ⇒ c/h_1 ∪ h_j ∪ h_f ⊕ err

∀h_j, h_f, c', h', J h_j ∧ c/h_1 ∪ h_j ∪ h_f → c'/h' ⇒

∃h'_1, h'_j, h' = h'_1 ∪ h'_j ∪ h_f ∧ J h'_j ∧ Safe^n c' h'_1 Q

Safe^{n+1} c h_1 Q

The inductive case: c in h_1 is safe for n + 1 steps if
A weak semantic triple with step indexing

∀a, c ≠ a

∀h_j, h_f, J h_j ⇒ c/h_1 ⊎ h_j ⊎ h_f ↛ err

∀h_j, h_f, c', h', J h_j ∧ c/h_1 ⊎ h_j ⊎ h_f → c'/h' ⇒

∃h', h'_j, h' = h'_1 ⊎ h'_j ⊎ h_f ∧ J h'_j ∧ Safe^n c' h'_1 Q

Safe^{n+1} c h_1 Q

The inductive case: c in h_1 is safe for n + 1 steps if
• in every heap h of the shape h_1 ⊎ h_j ⊎ h_f with h_j satisfying J, c/h causes no errors, and …
∀a, c ≠ a

∀h_j, h_f, J h_j ⇒ c/h_1 ⊕ h_j ⊕ h_f ↛ err

∀h_j, h_f, c', h', J h_j ∧ c/h_1 ⊕ h_j ⊕ h_f → c'/h' ⇒

∃h'_1, h'_j, h' = h'_1 ⊕ h'_j ⊕ h_f ∧ J h'_j ∧ Safe^n c' h'_1 Q

Safe^{n+1} c h_1 Q

The inductive case: c in h_1 is safe for n + 1 steps if

• in every heap h of the shape h_1 ⊕ h_j ⊕ h_f with h_j satisfying J, c/h causes no errors, and ...

• for every reduction c/h → c'/h', the heap h' decomposes as h'_1 ⊕ h'_j ⊕ h_f with h'_j satisfying J, and moreover c' in h'_1 is safe for the remaining n steps.
Semantic soundness and heap decompositions

It is relatively easy to show that this semantic triple $J \models \{ \{ P \} \} c \{ \{ Q \} \}$ validates the rules of concurrent separation logic.

Below, we illustrate the decomposition $h = h_1 \cup h_j \cup h_f$ to be used for validating the main rules:

\[
\begin{align*}
\text{emp} & \vdash \{ P \ast J \} c \{ Q \ast J \} \\
\hline
J & \vdash \{ P \} \text{ atomic } c \{ Q \} \\
J \ast J' & \vdash \{ P \} c \{ Q \} \\
\hline
J & \vdash \{ P \ast J' \} c \{ \lambda v. Q \, v \ast J' \}
\end{align*}
\]

\[
\begin{align*}
\hline
(h_1 \cup h_j) \cup \emptyset \cup h_f & \\
\hline
h_1 \cup h_j \cup h_f & \\
(h_1 \cup h_j) \cup (h_j \cup h_2) \cup h_f & \\
(h_1 \cup h_2) \cup h_j \cup h_f &
\end{align*}
\]
Semantic soundness and heap decompositions

\[
\begin{align*}
J \vdash \{ P_1 \} c_1 \{ \lambda_. \ Q_1 \} \\
J \vdash \{ P_2 \} c_2 \{ \lambda_. \ Q_2 \}
\end{align*}
\]

\[
\frac{J \vdash \{ P_1 \ast P_2 \} c_1 \parallel c_2 \{ \lambda_. \ Q_1 \ast Q_2 \}}{
J \vdash \{ P_1 \ast P_2 \} \{ \lambda_. \ Q_1 \ast Q_2 \}}
\]

\[
\begin{align*}
h_1 \uplus h_j \uplus (h_f \uplus h_2) \\
or h_2 \uplus h_j \uplus (h_f \uplus h_1)
\end{align*}
\]

\[
\frac{(h_1 \uplus h_2) \uplus h_j \uplus h_f}{(h_1 \uplus h_2) \uplus h_j \uplus h_f}
\]

\[
\begin{align*}
J \vdash \{ P \} c \{ Q \} \\
J \ast J' \vdash \{ P \} c \{ Q \}
\end{align*}
\]

\[
\frac{h_1 \uplus h_j \uplus (h_f \uplus h_j')}{h_1 \uplus (h_j \uplus h_j') \uplus h_f}
\]

\[
\begin{align*}
J \vdash \{ P \} c \{ Q \}
\end{align*}
\]

\[
J \vdash \{ P \ast R \} c \{ \lambda v. \ Q \ast v \ast R \}
\]
Absence of race conditions

\[(c_1 \parallel c_2)/h \rightarrow \text{err} \quad \text{if} \quad \text{Acc}(c_1) \cap \text{Acc}(c_2) \neq \emptyset\]

If we add the error rule above and take

\[\text{Acc(atomic } c) = \emptyset,\]

the proof of semantic soundness still works. This shows:

*Every command c provable in concurrent separation logic contains no race conditions between non-atomic memory accesses.*

Note: \(\text{atomic(set}(p, 1)) \parallel \text{atomic(set}(p, 2))\) is provable but is not considered as a race condition.
Summary
After the lightning strike that was separation logic in 2001, concurrent separation logic in 2004 was a resounding thunderclap.

Compared with earlier logics for concurrency (e.g. Owicki & Gries, 1976), concurrent separation logic was a huge step forward to prove safety properties of parallel computations:

- absence of race conditions;
- memory safety (no use after free, no double free);
- integrity of data structures;
- data transfers between processes.

Still not obvious how to prove functional correctness...

\[
\{ x = 0 \} \text{atomic}(x := x + 1) \parallel \text{atomic}(x := x + 1) \{ x = 2 \}
\]
References
References

A reference book on shared-memory concurrency:


The paper that introduced concurrent separation logic (revised version):


The simple proof of semantic soundness:

- V. Vafeiadis, *Concurrent separation logic and operational semantics*, MFPS 2011

Mechanizations:

- The companion Coq development for this lecture: https://github.com/xavierleroy/cdf-program-logics
- The Iris framework: https://iris-project.org/