Shared-memory concurrency: concurrent separation logic

Xavier Leroy

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr

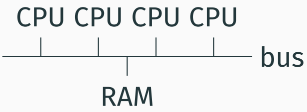
Introduction:
Shared-memory parallel computing

Bonus Bureau, Computing Divison, 11/24/1924

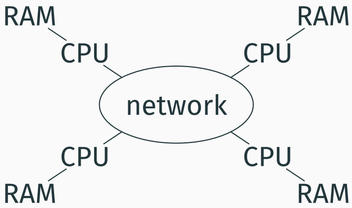
Use several processors (CPUs) together to perform a computation more quickly.

Two main models of parallel computing:

shared memory



distributed memory



Many implementation 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

$$\mathtt{set}(\ell, 1) \,\Big|\Big|\, \mathtt{set}(\ell, 2)$$

The program does not control which value ends up in location ℓ.

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

$$\mathtt{set}(\ell, 1) \,\Big|\Big|\, \mathtt{let}\ x = \mathtt{get}(\ell)$$

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):

$$\begin{array}{l|l} \texttt{let}\ t = \texttt{get}(\&x)\ \texttt{in} & \texttt{let}\ t = \texttt{get}(\&x)\ \texttt{in} \\ \texttt{let}\ t = t + 1\ \texttt{in} & \texttt{let}\ t = t + 1\ \texttt{in} \\ \texttt{set}(\&x, t) & \texttt{set}(\&x, t) \end{array}$$

An example of race condition

$$x := x + 1 \parallel x := x + 1$$

One possible execution:

$$\begin{array}{l|l}
\texttt{let } t = \texttt{get}(\&x) \texttt{ in} & \\
\texttt{let } t = t + 1 \texttt{ in} & \\
\texttt{set}(\&x, t) & \\
 & \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\
 & \texttt{let } t = t + 1 \texttt{ in} \\
 & \texttt{set}(\&x, t)
\end{array}$$

With $x = 0$ initially, we end with $x = 2$.

An example of race condition

$$x := x + 1 \parallel x := x + 1$$

Another possible execution:

$$\begin{array}{l|l}
\texttt{let } t = \texttt{get}(\&x) \texttt{ in} & \\
\texttt{let } t = t + 1 \texttt{ in} & \\
 & \texttt{let } t = \texttt{get}(\&x) \texttt{ in} \\
\texttt{set}(\&x, t) & \\
 & \texttt{let } t = t + 1 \texttt{ in} \\
 & \texttt{set}(\&x, t)
\end{array}$$

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).

$$\begin{array}{l} \texttt{while } i \geq N \texttt{ do } \texttt{pause}(); \\ T[i] := x; \\ i := i + 1; \end{array}$$

A more realistic example

With two producers in parallel:

$\texttt{while}\ i \geq N\ \texttt{do pause}();$	
	$\texttt{while}\ i \geq N\ \texttt{do pause}();$
	$T[i] := x_1;$
	$i := i + 1;$
$T[i] := x_2;$ ✘	
$i := i + 1;$	

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

A more realistic example

With two producers in parallel:

$\texttt{while}\ i \geq N\ \texttt{do}\ \texttt{pause}();$	
	$\texttt{while}\ i \geq N\ \texttt{do}\ \texttt{pause}();$
$T[i] := x_1;$	
	$T[i] := x_2;$
	$i := i + 1;$
$i := i + 1;$	

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:

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

In 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.

```
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:

$$
\begin{aligned}
c \mathrel{:=}\ & \dots \\
\mid\ & c_1 \parallel c_2 \quad \text{execute } c_1 \text{ and } c_2 \text{ in parallel}
\end{aligned}
$$

Semantics:: an interleaving of the reductions of c_1 and c_2.

$$
\begin{array}{ll}
(a_1 \parallel a_2)/h \to 0/h & \text{(or any combination of } a_1 \text{ and } a_2\text{)} \\
(c_1 \parallel c_2)/h \to (c_1' \parallel c_2)/h' & \text{if } c_1/h \to c_1'/h' \\
(c_1 \parallel c_2)/h \to (c_1 \parallel c_2')/h' & \text{if } c_2/h \to c_2'/h' \\
(c_1 \parallel c_2)/h \to \mathtt{err} & \text{if } c_1/h \to \mathtt{err} \text{ or } c_2/h \to \mathtt{err}
\end{array}
$$

Separation logic rule for parallel execution

$$\frac{\{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \,\} \qquad \{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_2 \,\}}{\{\, P_1 \ast P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda\_.\, Q_1 \ast Q_2 \,\}}$$

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

$$\frac{\{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \,\} \qquad \{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_2 \,\}}{\{\, P_1 \ast P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda\_.\, Q_1 \ast Q_2 \,\}}$$

Alternate intuition: the precondition $P_1 \ast P_2$ 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 $c_2; c_1$.

$$\frac{\dfrac{\{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \,\}}{\{\, P_1 \ast P_2 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \ast P_2 \,\}} \qquad \dfrac{\{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_2 \,\}}{\{\, Q_1 \ast P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_1 \ast Q_2 \,\}}}{\{\, P_1 \ast P_2 \,\}\; c_1; c_2 \;\{\, \lambda\_.\, Q_1 \ast Q_2 \,\}}$$

Parallelism between disjoint sub-arrays

Example: Quicksort.

$$
\begin{array}{l}
\mathit{quicksort}\ T\ l\ h = \\
\quad \texttt{if}\ h - l \leq 50\ \texttt{then} \\
\quad\quad \mathit{insertionsort}\ T\ l\ h \\
\quad \texttt{else} \\
\quad\quad \texttt{let}\ m = \mathit{partition}\ T\ l\ h\ \texttt{in} \\
\quad\quad \mathit{quicksort}\ T\ l\ m \parallel \mathit{quicksort}\ T\ (m+1)\ h
\end{array}
$$

$\mathit{quicksort}\ T\ l\ h$ modifies the sub-array $T[l \ldots h]$ 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

$$\begin{aligned} \mathit{tree}(\texttt{Leaf}, p) &= \langle p = \texttt{NULL} \rangle \\ \mathit{tree}(\texttt{Node}(t_1, x, t_2), p) &= \exists p_1, p_2,\ p \mapsto p_1 \ast p + 1 \mapsto x \ast p + 2 \mapsto p_2 \\ &\qquad \ast\ \mathit{tree}(t_1, p_1) \ast \mathit{tree}(t_2, p_2) \end{aligned}$$

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

$$\begin{array}{l} \mathit{incrtree}\ t\ \delta = \\ \quad \texttt{if}\ t \neq \texttt{NULL}\ \texttt{then} \\ \qquad \texttt{let}\ l = \texttt{get}(t)\ \texttt{and}\ n = \texttt{get}(t+1)\ \texttt{and}\ r = \texttt{get}(t+2)\ \texttt{in} \\ \qquad \texttt{set}(t+1, n+\delta); \\ \qquad \mathit{incrtree}\ l\ \delta \parallel \mathit{incrtree}\ r\ \delta \end{array}$$

Absence of race conditions

We add one reduction rule that signals an error when a race condition occurs:

$$(c_1 \parallel c_2)/h \to \texttt{err} \quad \text{if} \quad \mathit{Acc}(c_1) \cap \mathit{Acc}(c_2) \neq \emptyset$$

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

$$\begin{aligned}
\mathit{Acc}(\texttt{get}(a)) = \mathit{Acc}(\texttt{set}(a, a')) = \mathit{Acc}(\texttt{free}(a)) &= \{a\} \\
\mathit{Acc}(\texttt{let}\ x = c_1\ \texttt{in}\ c_2) &= \mathit{Acc}(c_1) \\
\mathit{Acc}(c_1 \parallel c_2) &= \mathit{Acc}(c_1) \cup \mathit{Acc}(c_2)
\end{aligned}$$

Absence of race conditions

It is easy to show that

$$c/h \not\to \texttt{err} \quad \Rightarrow \quad Acc(c) \subseteq Dom(h)$$

Therefore, if $c_1/h_1 \not\to \texttt{err}$ and $c_2/h_2 \not\to \texttt{err}$ and $h_1 \perp h_2$,

$$Acc(c_1) \cap Acc(c_2) \subseteq Dom(h_1) \cap Dom(h_2) = \emptyset$$

and $(c_1 \parallel c_2)/(h_1 \uplus h_2)$ cannot reduce to $\texttt{err}$ 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.

O'Hearn (2001–2002), *Notes on separation logic for shared-variable concurrency*, unpublished.

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

O'Hearn (2004), *Resources, Concurrency and Local Reasoning*. The key ideas + the main examples.

Brookes (2004), *A Semantics for Concurrent Separation Logic*. A semantic and a soundness proof for O'Hearn's logic.

Shared resources

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)

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

Example (shared doubly-linked list)

```
class DList { DListCell first, last; }
class DListCell { Object data; DListCell prev, next; }
```

Shared resources in separation logic

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

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

in mutual exclusion with the other processes.

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

$$\frac{\{\, RI_r \star P \,\}\ c\ \{\, RI_r \star Q \,\}}{\{\, P \,\}\ \texttt{with}\ r\ \texttt{do}\ c\ \{\, Q \,\}}$$

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.

Conditional critical sections in separation logic

O'Hearn's original article considers conditional critical sections

$$\texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c$$

where c is executed only when the condition b is true.

The rule for c.c.s. is

$$\frac{\{\, \langle b \rangle \star RI_r \star P \,\}\ c\ \{\, RI_r \star Q \,\}}{\{\, P \,\}\ \texttt{with}\ r\ \texttt{when}\ b\ \texttt{do}\ c\ \{\, Q \,\}}$$

Example: decrementing a shared counter

The invariant is $RI_r = \exists n,\ p \mapsto n \star \langle n \geq 0 \rangle$.

$$\{ \texttt{emp} \}$$

$\texttt{with}\ r\ \texttt{do}$

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

$\quad \texttt{let}\ n = \texttt{get}(p)\ \texttt{in}$

$$\{ p \mapsto n \star \langle n \geq 0 \rangle \}$$

$\quad \texttt{if}\ n > 0\ \texttt{then}\ \texttt{set}(p, n-1)$

$$\{ \exists n',\ p \mapsto n' \star \langle n' \geq 0 \rangle \}$$

$\texttt{done}$

$$\{ \texttt{emp} \}$$

Example: insertion in a shared list

The invariant is $RI_r = \exists q, w,\ p \mapsto q \star list(w, q)$.

$$\{\, \mathtt{emp} \,\}$$

`with` r `do`

$$\{\, \exists q, w,\ p \mapsto q \star list(w, q) \,\}$$

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

$$\{\, p \mapsto q \star \exists w,\ list(w, q) \,\}$$

  `let` $a = cons(x, q)$ `in`

$$\{\, a \mapsto x \star a + 1 \mapsto q \star p \mapsto q \star \exists w,\ list(w, q) \,\}$$

  $\mathtt{set}(p, a)$

$$\{\, p \mapsto a \star a \mapsto x \star a + 1 \mapsto q \star \exists w,\ list(w, q) \,\}$$

$$\Rightarrow \{\, \exists q, w,\ p \mapsto q \star list(w, q) \,\}$$

`done`

$$\{\, \mathtt{emp} \,\}$$

Simplified formalization: atomic sections (Vafeiadis, 2011)

Commands:

$$
\begin{array}{lll}
c ::= \dots & & \\
\quad \mid c_1 \parallel c_2 & & \text{execute } c_1 \text{ and } c_2 \text{ in parallel} \\
\quad \mid \texttt{atomic}\ c & & \text{execute } c \text{ in one uninterruptible step}
\end{array}
$$

A "super-critical" section: during the execution of `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 atomic instructions of the processor.

Modeling atomic instructions provided by the processor

Atomic swap and its special cases:

$$\begin{aligned}
\mathit{swap}(p, n) &\stackrel{\mathit{def}}{=} \mathtt{atomic}(\mathtt{let}\ x = \mathtt{get}(p)\ \mathtt{in}\ \mathtt{set}(p, n); x) \\
\mathit{test_and_set}(p) &\stackrel{\mathit{def}}{=} \mathit{swap}(p, 1) \\
\mathit{read_and_clear}(p) &\stackrel{\mathit{def}}{=} \mathit{swap}(p, 0)
\end{aligned}$$

Atomic increment / decrement:

$$\mathit{fetch\_and\_add}(p, d) \stackrel{\mathit{def}}{=} \mathtt{atomic}(\mathtt{let}\ x = \mathtt{get}(p)\ \mathtt{in}\ \mathtt{set}(p, x + d); x)$$

Compare and swap:

$$\begin{aligned}
\mathit{CAS}(p, x, n) \stackrel{\mathit{def}}{=} \mathtt{atomic}(&\mathtt{let}\ c = \mathtt{get}(p)\ \mathtt{in} \\
&\qquad \mathtt{if}\ c = x\ \mathtt{then}\ (\mathtt{set}(p, n); 1)\ \mathtt{else}\ 0)
\end{aligned}$$

Operational semantics for atomic sections

$$(\texttt{atomic}\ c)/h \rightarrow a/h' \qquad \text{if } c/h \xrightarrow{*} a/h'$$

$$(\texttt{atomic}\ c)/h \rightarrow \texttt{err} \qquad \text{if } c/h \xrightarrow{*} \texttt{err}$$

Note: $\texttt{atomic}\ c_1 \parallel \texttt{atomic}\ c_2$ 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, $(\texttt{atomic}\ c)/h$ 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 `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:

$$\frac{\mathtt{emp} \vdash \{\, P \ast J \,\}\; c \;\{\, \lambda v.\, Q\, v \ast J \,\}}{J \vdash \{\, P \,\}\; \mathtt{atomic}\; c \;\{\, Q \,\}}$$

Sharing a resource J' :

$$\frac{J \ast J' \vdash \{\, P \,\}\; c \;\{\, Q \,\}}{J \vdash \{\, P \ast J' \,\}\; c \;\{\, \lambda v.\, Q\, v \ast J' \,\}}$$

Framing the invariant:

$$\frac{J \vdash \{\, P \,\}\; c \;\{\, Q \,\}}{J \ast J' \vdash \{\, P \,\}\; c \;\{\, Q \,\}}$$

The rules for control structures (reminder)

$$\frac{P \Rightarrow Q\ [\![a]\!]}{J \vdash \{\, P \,\}\ a\ \{\, Q \,\}}$$

$$\frac{J \vdash \{\, P \,\}\ c\ \{\, R \,\} \qquad \forall v,\ J \vdash \{\, R\ v \,\}\ c'[x \leftarrow v]\ \{\, Q \,\}}{J \vdash \{\, P \,\}\ \texttt{let}\ x = c\ \texttt{in}\ c'\ \{\, Q \,\}}$$

$$\frac{J \vdash \{\, \langle b \rangle \star P \,\}\ c_1\ \{\, Q \,\} \qquad J \vdash \{\, \langle \neg b \rangle \star P \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\, Q \,\}}$$

$$\frac{J \vdash \{\, P_1 \,\}\ c_1\ \{\, \lambda\_.\ Q_1 \,\} \qquad J \vdash \{\, P_2 \,\}\ c_2\ \{\, \lambda\_.\ Q_2 \,\}}{J \vdash \{\, P_1 \star P_2 \,\}\ c_1 \parallel c_2\ \{\, \lambda\_.\ Q_1 \star Q_2 \,\}}$$

The “small rules” for heap operations (reminder)

$$J \vdash \{\, \mathrm{emp} \,\}\ \texttt{alloc}(N)\ \{\, \lambda \ell.\, \ell \mapsto \_ \star \cdots \star \ell + N - 1 \mapsto \_ \,\}$$

$$J \vdash \{\, [\![a]\!] \mapsto x \,\}\ \texttt{get}(a)\ \{\, \lambda v.\, \langle v = x \rangle \star [\![a]\!] \mapsto x \,\}$$

$$J \vdash \{\, [\![a]\!] \mapsto \_ \,\}\ \texttt{set}(a, a')\ \{\, \lambda v.\, [\![a]\!] \mapsto [\![a']\!] \,\}$$

$$J \vdash \{\, [\![a]\!] \mapsto \_ \,\}\ \texttt{free}(a)\ \{\, \lambda v.\, \mathrm{emp} \,\}$$

The structural rules (watch out! there's a catch!)

$$\frac{J \vdash \{\, P \,\}\; c \;\{\, Q \,\}}{J \vdash \{\, P \ast R \,\}\; c \;\{\, \lambda v.\, Q\, v \ast R \,\}}\ \text{(frame)}$$

$$\frac{P \Rightarrow P' \qquad J \vdash \{\, P' \,\}\; c \;\{\, Q' \,\} \qquad \forall v,\; Q'\, v \Rightarrow Q\, v}{J \vdash \{\, P \,\}\; c \;\{\, Q \,\}}\ \text{(consequence)}$$

$$\frac{J \vdash \{\, P \,\}\; c \;\{\, Q \,\} \qquad J \vdash \{\, P' \,\}\; c \;\{\, Q' \,\}}{J \vdash \{\, P \vee P' \,\}\; c \;\{\, \lambda v.\, Q\, v \vee Q'\, v \,\}}\ \text{(disjunction)}$$

$$\frac{J \text{ precise} \qquad J \vdash \{\, P \,\}\; c \;\{\, Q \,\} \qquad J \vdash \{\, P' \,\}\; c \;\{\, Q' \,\}}{J \vdash \{\, P \wedge P' \,\}\; c \;\{\, \lambda v.\, Q\, v \wedge Q'\, v \,\}}\ \text{(conjunction)}$$

The conjunction rule and Reynold's counterexample

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

$$\texttt{emp} \vdash \{\, \texttt{one} \ast \texttt{true} \,\}\ 0\ \{\, \lambda\_.\texttt{emp} \ast \texttt{true} \,\}$$

$$\texttt{emp} \vdash \{\, \texttt{one} \ast \texttt{true} \,\}\ 0\ \{\, \lambda\_.\texttt{one} \ast \texttt{true} \,\}$$

and, by application of the $\texttt{atomic}$ rule,

$$J \vdash \{\, \texttt{one} \,\}\ \texttt{atomic}\ 0\ \{\, \lambda\_.\texttt{emp} \,\}$$

$$J \vdash \{\, \texttt{one} \,\}\ \texttt{atomic}\ 0\ \{\, \lambda\_.\texttt{one} \,\}$$

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

$$J \vdash \{\, \texttt{one} \wedge \texttt{one} \,\}\ \texttt{atomic}\ 0\ \{\, \lambda\_.\texttt{emp} \wedge \texttt{one} \,\}$$

yet the postcondition $\texttt{emp} \wedge \texttt{one}$ 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:

$$h = h_1 \uplus h_2 = h'_1 \uplus h'_2 \quad \wedge \quad P\,h_1 \quad \wedge \quad P\,h'_1 \quad \Rightarrow \quad h_1 = h'_1$$

Examples of precise / imprecise assertions

Precise assertions	Imprecise assertions
$\texttt{emp}$	$\texttt{true}$
$\ell \mapsto \_$	$\exists \ell,\ \ell \mapsto \_$
$\ell \mapsto v$	$\exists \ell,\ \ell \mapsto v$
$\exists v, \ell \mapsto v \star R(v)$	
$P \star Q$	$P \star \texttt{true}$
$\langle b \rangle \star P \vee \langle \neg b \rangle \star Q$	$\texttt{emp} \vee \ell \mapsto \_$

(assuming P, Q, $R(v)$ to be precise)

Binary semaphores and applications

Implementing binary semaphores

A binary semaphore = a memory location p containing 0 (meaning "busy") or 1 (meaning "available").

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

$$
\begin{aligned}
V(sem) &= \mathtt{atomic}(\mathtt{set}(sem, 1)) \\
P(sem) &= \mathtt{let}\ x = swap(sem, 0)\ \mathtt{in} \\
&\qquad\quad \mathtt{if}\ x = 1\ \mathtt{then}\ 0\ \mathtt{else}\ P(sem)
\end{aligned}
$$

where

$$swap(p, n) = \mathtt{atomic}(\mathtt{let}\ x = \mathtt{get}(p)\ \mathtt{in}\ \mathtt{set}(p, n); x)$$

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

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

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

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

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 $RI = \exists n, x \mapsto n \star \langle n \text{ premier} \rangle$, "variable x contains a prime number".

$$\{\, sem \mapsto 0 \star x \mapsto \_ \,\}$$

$$
\begin{array}{l||l}
\{\, x \mapsto \_ \,\} & \quad\quad \{\, \texttt{emp} \,\} \\
\quad\quad \texttt{set}(x, 53); & P(sem); \\
\{\, x \mapsto 53 \,\} \Rightarrow \{\, RI \,\} & \quad\quad \{\, RI \,\} \\
\quad\quad V(sem) & \texttt{let}\ n = \texttt{get}(x)\ \texttt{in} \\
\{\, \texttt{emp} \,\} & \quad\quad \{\, x \mapsto n \star \langle n \text{ prime} \rangle \,\} \\
 & \texttt{print}(n)
\end{array}
$$

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.

Synchronization and resource transfer with a semaphore

Consider the assertion $RI = \exists p, x \mapsto p \ast p \mapsto \_$
"variable x points to a valid memory location".

$$\{\, sem \mapsto 0 \ast x \mapsto \_ \,\}$$

$$
\begin{array}{l|l}
\{\, x \mapsto \_ \,\} & \qquad \{\, \texttt{emp} \,\} \\
\qquad \texttt{let } p = \texttt{alloc}(1) \texttt{ in} & P(sem); \\
\{\, x \mapsto \_ \ast p \mapsto \_ \,\} & \qquad \{\, RI \,\} \\
\qquad \texttt{set}(x, p); & \texttt{let } p = \texttt{get}(x) \texttt{ in} \\
\{\, x \mapsto p \ast p \mapsto \_ \,\} \Rightarrow \{\, RI \,\} & \qquad \{\, x \mapsto p \ast p \mapsto \_ \,\} \\
\qquad V(sem) & \texttt{free}(p) \\
\{\, \texttt{emp} \,\} & \qquad \{\, x \mapsto \_ \,\}
\end{array}
$$

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:

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

For $swap(sem, 0)$, we have the triple

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

$P(sem)$ iterates $swap(sem, 0)$ until the result is 1, hence

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

Derivation of the rule for V

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

It suffices to show

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

to obtain $\mathtt{emp} \vdash \{\, RI \star J(sem, RI) \,\}\ \mathtt{set}(sem, 1)\ \{\, J(sem, RI) \,\}$
and therefore $J(sem, RI) \vdash \{\, RI \,\}\ V(sem)\ \{\, \mathtt{emp} \,\}$.

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

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

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

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:

$$\texttt{with}\ r\ \texttt{do}\ c \quad \stackrel{def}{=} \quad P(r); c; V(r)$$

where each critical section r is identified by the location of a semaphore, initialized to 1.

Implementing critical sections

If RI_r is the resource invariant for r, the shared memory invariant is the conjunction of the invariants of the associated semaphores:

$$J_{\mathcal{R}} \quad = \quad \mathop{\bigstar}_{r \in \mathcal{R}} J(r, RI_r)$$

This implementation validates the rule for critical sections:

$$\frac{r \in \mathcal{R} \qquad J_{\mathcal{R} \setminus \{r\}} \vdash \{\, RI_r \star P \,\}\; c \;\{\, RI_r \star Q \,\}}{J_{\mathcal{R}} \vdash \{\, P \,\}\; \texttt{with}\; r \;\texttt{do}\; c \;\{\, Q \,\}}$$

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.

$$\texttt{with}\ r\ \texttt{when}\ c_b\ \texttt{do}\ c \quad \stackrel{def}{=} \quad P(r);\ wait(r, c_b);\ c;\ V(r)$$

where *wait* is the following busy-waiting loop:

$$\begin{aligned} wait(r, c_b) \quad = \quad & \texttt{let}\ b = c_b\ \texttt{in} \\ & \texttt{if}\ b\ \texttt{then}\ 0\ \texttt{else}\ (V(r);\ P(r);\ wait(r, c_b)) \end{aligned}$$

We can derive the following rule:

$$\frac{\begin{array}{c} r \in \mathcal{R} \\ J_{\mathcal{R}\setminus\{r\}} \vdash \{\, RI_r \star P \,\}\ c_b\ \{\, \lambda b.\ \langle b \rangle \star B \vee \langle \neg b \rangle \star RI_r \star P \,\} \\ J_{\mathcal{R}\setminus\{r\}} \vdash \{\, B \,\}\ c\ \{\, RI_r \star Q \,\} \end{array}}{J_{\mathcal{R}} \vdash \{\, P \,\}\ \texttt{with}\ r\ \texttt{when}\ c_b\ \texttt{do}\ c\ \{\, Q \,\}}$$

The producer/consumer device

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

$$
\begin{array}{l}
\texttt{while true do} \\
\quad \text{compute } x; \\
\quad \mathit{produce}(x); \\
\texttt{done}
\end{array}
\;\Bigg\|\;
\begin{array}{l}
\texttt{while true do} \\
\quad \texttt{let } y = \mathit{consume}() \texttt{ in} \\
\quad \text{use } y \\
\texttt{done}
\end{array}
$$

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:

$$\begin{aligned} \mathit{produce}(b, s_0, s_1, x) &= P(s_0);\ \texttt{set}(b, x);\ V(s_1) \\ \mathit{consume}(b, s_0, s_1) &= P(s_1);\ \texttt{let}\ x = \texttt{get}(b)\ \texttt{in}\ V(s_0);\ x \end{aligned}$$

Specification and verification of producer/consumer

Write $RI(x)$ the resource invariant associated with datum x.

Specification of *produce* and *consume*:

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

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

$$J(b) \quad \stackrel{def}{=} \quad J(s_0, b \mapsto \_) \star J(s_1, \exists x,\ b \mapsto x \star RI(x))$$

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

Semantic soundness of concurrent separation logic

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 $\mathtt{Safe}^n\ c\ h$.
- The conjunction rule is the only one that demands precise resource invariants.

Some intuitions

$$J \vdash \{ P \} \; c \; \{ Q \}$$

Deductive intuition: it's like $\{ P \ast J \} \; c \; \{ Q \ast J \}$
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

$$J \models \{\{\, P \,\}\}\; c\; \{\{\, Q \,\}\} \quad \stackrel{def}{=} \quad \forall n, h,\; P\; h \Rightarrow \mathtt{Safe}^n\; c\; h\; Q\; J$$

The inductive predicate $\mathtt{Safe}^n\; c\; h\; Q\; J$ 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.

$$\mathtt{Safe}^0\; c\; h\; Q\; J \qquad \frac{Q\; [\![a]\!]\; h}{\mathtt{Safe}^{n+1}\; a\; h\; Q\; J} \qquad \frac{(\forall a, c \neq a) \qquad \cdots}{\mathtt{Safe}^{n+1}\; c\; h\; Q\; J}$$

A weak semantic triple with step indexing

$$\dfrac{\begin{array}{l}\forall a, c \neq a \\ \forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\to \texttt{err} \\ \forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \to c'/h' \Rightarrow \\ \qquad\qquad \exists h_1', h_j',\ h' = h_1' \uplus h_j' \uplus h_f \wedge J\ h_j' \wedge \texttt{Safe}^n\ c'\ h_1'\ Q\end{array}}{\texttt{Safe}^{n+1}\ c\ h_1\ Q}$$

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

A weak semantic triple with step indexing

$$
\frac{
\begin{array}{l}
\forall a, c \neq a \\
\forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\rightarrow \texttt{err} \\
\forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \rightarrow c'/h' \Rightarrow \\
\qquad\qquad \exists h_1', h_j',\ h' = h_1' \uplus h_j' \uplus h_f \wedge J\ h_j' \wedge \texttt{Safe}^n\ c'\ h_1'\ Q
\end{array}
}{
\texttt{Safe}^{n+1}\ c\ h_1\ Q
}
$$

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

- in every heap h of the shape $h_1 \uplus h_j \uplus h_f$ with h_j satisfying J, c/h causes no errors, and …

A weak semantic triple with step indexing

$$\frac{\begin{array}{l}\forall a, c \neq a \\ \forall h_j, h_f,\ J\ h_j \Rightarrow c/h_1 \uplus h_j \uplus h_f \not\to \texttt{err} \\ \forall h_j, h_f, c', h',\ J\ h_j \wedge c/h_1 \uplus h_j \uplus h_f \to c'/h' \Rightarrow \\ \qquad\qquad \exists h_1', h_j',\ h' = h_1' \uplus h_j' \uplus h_f \wedge J\ h_j' \wedge \texttt{Safe}^n\ c'\ h_1'\ Q\end{array}}{\texttt{Safe}^{n+1}\ c\ h_1\ Q}$$

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

- in every heap h of the shape $h_1 \uplus h_j \uplus h_f$ with h_j satisfying J, c/h causes no errors, and …
- for every reduction $c/h \to c'/h'$, the heap h' decomposes as $h_1' \uplus h_j' \uplus h_f$ 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 $h = h_1 \uplus h_j \uplus h_f$ to be used for validating the main rules:

$$\frac{\mathtt{emp} \vdash \{\, P \ast J \,\}\; c\; \{\, Q \ast J \,\}}{J \vdash \{\, P \,\}\; \mathtt{atomic}\; c\; \{\, Q \,\}} \qquad \frac{(h_1 \uplus h_j) \quad \uplus \quad \emptyset \quad \uplus \quad h_f}{h_1 \quad \uplus \quad h_j \quad \uplus \quad h_f}$$

$$\frac{J \ast J' \vdash \{\, P \,\}\; c\; \{\, Q \,\}}{J \vdash \{\, P \ast J' \,\}\; c\; \{\, \lambda v.\, Q\, v \ast J' \,\}} \qquad \frac{h_1 \quad \uplus \quad (h_j \uplus h_2) \quad \uplus \quad h_f}{(h_1 \uplus h_2) \quad \uplus \quad h_j \quad \uplus \quad h_f}$$

Semantic soundness and heap decompositions

$$\frac{J \vdash \{\, P_1 \,\}\; c_1 \;\{\, \lambda\_.\, Q_1 \,\} \qquad J \vdash \{\, P_2 \,\}\; c_2 \;\{\, \lambda\_.\, Q_2 \,\}}{J \vdash \{\, P_1 \star P_2 \,\}\; c_1 \parallel c_2 \;\{\, \lambda\_.\, Q_1 \star Q_2 \,\}}$$

$$\frac{\begin{array}{r} h_1 \uplus h_j \uplus (h_f \uplus h_2) \\ \text{or } h_2 \uplus h_j \uplus (h_f \uplus h_1) \end{array}}{(h_1 \uplus h_2) \uplus h_j \uplus h_f}$$

$$\frac{J \vdash \{\, P \,\}\; c \;\{\, Q \,\}}{J \star J' \vdash \{\, P \,\}\; c \;\{\, Q \,\}} \qquad \frac{h_1 \uplus h_j \uplus (h_f \uplus h_j')}{h_1 \uplus (h_j \uplus h_j') \uplus h_f}$$

Absence of race conditions

$$(c_1 \parallel c_2)/h \to \texttt{err} \quad \text{if} \quad Acc(c_1) \cap Acc(c_2) \neq \emptyset$$

If we add the error rule above and take

$$Acc(\texttt{atomic}\ c) = \emptyset,$$

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: $\texttt{atomic}(\texttt{set}(p,1)) \parallel \texttt{atomic}(\texttt{set}(p,2))$ is provable but is not considered as a race condition.

Summary

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...

$$\{\, x = 0 \,\}\ \texttt{atomic}(x := x + 1) \parallel \texttt{atomic}(x := x + 1)\ \{\, x = 2 \,\}$$

References

References

A reference book on shared-memory concurrency:

- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*, Morgan Kaufman, 2012.

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

- P. O'Hearn, *Resources, Concurrency and Local Reasoning*, Theor. Comp. Sci, 2007.

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/`