Resources, Concurrency, and Local Reasoning

Disjoint concurrency

Michiel Verloop
Loes Kruger

Type Theory and Coq

3rd December 2020

Outline

Introduction

Basic Principles
- Racy and Race-free Programs
- Cautious and Daring Concurrency
- Ownership and Separation
- Reasoning about programs

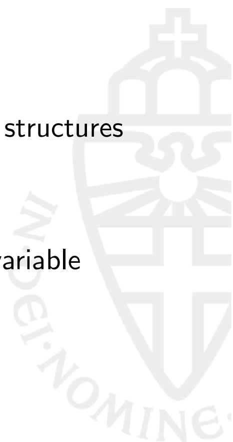
Disjoint Concurrency
- Simple examples
- Merge sort

Conclusion

Introduction: Resource Separation

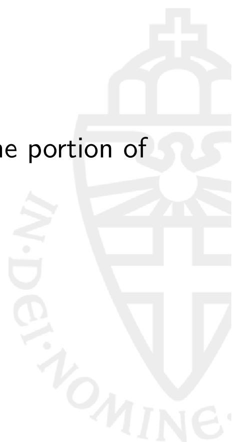
- Resources, Concurrency and Local Reasoning by Peter W. O'Hearn
- Correct resource usage is essential
- Resource separation
  - Reduces complexity
  - Less time-dependent errors

Introduction: Separation Logic



- Separation logic: accessing and modifying shared structures
- Separating conjunction: $p_1 * p_2$
- Concise and flexible description of structures
- Break state into chunks, can be used for shared-variable concurrency

Race-free



Race-free:
Concurrent processes do not attempt to share the same portion of state at the same time.

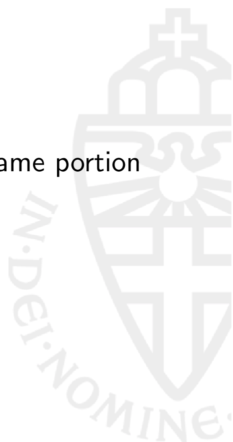
Race-free

Race-free:
Concurrent processes do not attempt to share the same portion of state at the same time.

$$x := y \parallel z := 3$$

Race-free only if z is not an alias of x or y, or vice versa.

Racy



Racy:
Multiple concurrent processes attempt to access the same portion of state at the same time.

$$w := y + x \parallel v := x \cdot z$$

Consistency

A program is consistent iff there is exactly one final state for all execution paths.

Consistency

A program is consistent iff there is exactly one final state for all execution paths.

- Consistent: $w := y + x \parallel v := x \cdot z$

Consistency

A program is consistent iff there is exactly one final state for all execution paths.

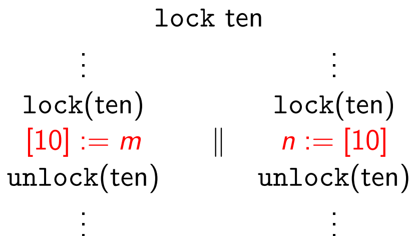
- Consistent: $w := y + x \parallel v := x \cdot z$
- Inconsistent: $x := y + x \parallel x := x \cdot z$

Racy vs Race-free

- No (clear) distinction in the literature
- In practice, the distinction is crucial
- Race-free: no complexity from interleavings

Mutual exclusion group

Mutual exclusion group:
Set of commands whose elements are required not to overlap in their executions.



Semaphores

Semaphore: Non-negative integer used to control access to a common resource by multiple processes in a concurrent system.

- P(s): $s - 1$ if $s > 0$ otherwise waiting
- V(s): $s + 1$

```
P(s)
critical piece of code
V(s)
```

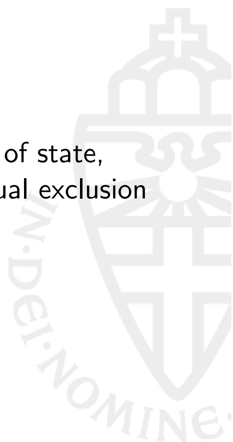
Semaphores

The following example uses two semaphores sem1 and sem2

$$\texttt{semaphore}\ \mathrm{sem1} := 1;\ \mathrm{sem2} := 0$$

$$\begin{array}{ccc} \vdots & & \vdots \\ \texttt{P}(\mathrm{sem1}) & & \texttt{P}(\mathrm{sem2}) \\ [10] := m & \| & n := [10] \\ \texttt{V}(\mathrm{sem2}) & & \texttt{V}(\mathrm{sem1}) \\ \vdots & & \vdots \end{array}$$

Cautious and Daring



Cautious:
Whenever concurrent processes access the same piece of state, they do so only within commands from the same mutual exclusion group.

Daring:
Not cautious.

Daring

Daring:

$$\texttt{semaphore sem1} := 1;\ \text{sem2} := 0$$

$$\begin{array}{ccc} \vdots & & \vdots \\ \texttt{P}(\text{sem1}) & & \texttt{P}(\text{sem2}) \\ [10] := m & \| & n := [10] \\ \texttt{V}(\text{sem2}) & & \texttt{V}(\text{sem1}) \\ \vdots & & \vdots \end{array}$$

Daring

Daring and inconsistent:

$$\texttt{semaphore}\ \text{sem1} := 1;\ \text{sem2} := 0$$

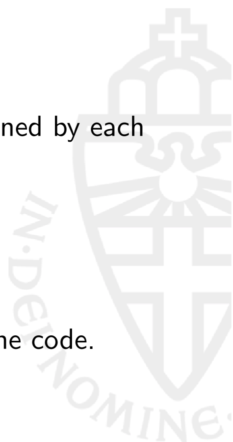
$$\begin{array}{ccccc} \vdots & & \vdots & & \\ & & & & \vdots \\ \texttt{P}(\text{sem1}) & & \texttt{P}(\text{sem2}) & & \\ [10] := m & \parallel & n := [10] & \parallel & [10] := 9 \\ \texttt{V}(\text{sem2}) & & \texttt{V}(\text{sem1}) & & \vdots \\ \vdots & & \vdots & & \end{array}$$

Ownership Hypothesis

Ownership Hypothesis:
A code fragment can access only those portions of state that it owns.

- $10 \mapsto -$:
  I own the cell at address 10.
  The value of the cell is unknown, indicated by $-$.
- $10 \mapsto 3$:
  I own the cell at address 10.
  The value of the cell is 3.

Separation Property



Separation Property:
At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

- $\{\texttt{emp}\}\texttt{P}(\texttt{sem1})\{10 \mapsto -\}$
  Cell at address 10 can be accessed in the code.
- $\{10 \mapsto -\}\texttt{V}(\texttt{sem2})\{\texttt{emp}\}$
  Cell at address 10 can no longer be accessed in the code.

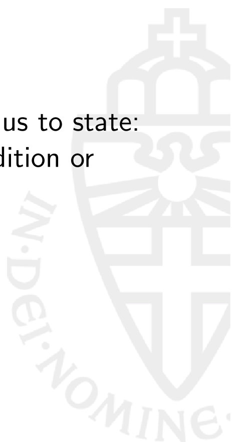
Separation Property

Separation Property:
At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

$$\texttt{semaphore}\ \mathsf{sem1} := 1;\ \mathsf{sem2} := 0$$

$$\begin{array}{ccc} \vdots & & \vdots \\ \{\texttt{emp}\} & & \{\texttt{emp}\} \\ \texttt{P}(\mathsf{sem1}) & & \texttt{P}(\mathsf{sem2}) \\ \{10 \mapsto -\} & & \{10 \mapsto -\} \\ {[10]} := m & \parallel & n := [10] \\ \{10 \mapsto -\} & & \{10 \mapsto -\} \\ \texttt{V}(\mathsf{sem2}) & & \texttt{V}(\mathsf{sem1}) \\ \{\texttt{emp}\} & & \{\texttt{emp}\} \\ \vdots & & \vdots \end{array}$$

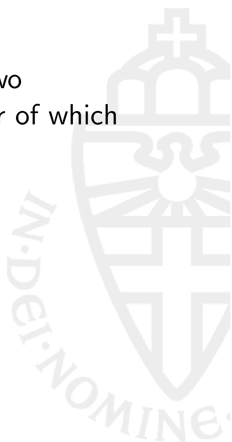
Reasoning about programs



Extending separation logic with this hypothesis allows us to state: If $\{P\}C\{Q\}$ can be proven, then there is no race condition or attempt to access a dangling pointer.

$$\{10 \mapsto 3\}x := [10]\{(10 \mapsto 3) \wedge x = 3\}$$

Disjoint Concurrency Rule

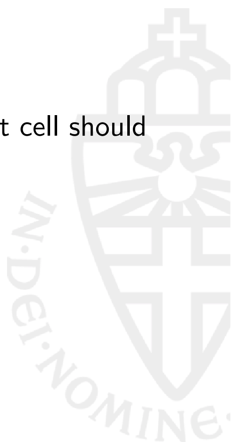


Recall, $P * Q$ means that the heap can be split into two components, one of which makes P true and the other of which makes Q true.

The rule for disjoint concurrency is defined as follows:

$$\frac{\{P\}C\{Q\} \quad \{P'\}C'\{Q'\}}{\{P * P'\}C \parallel C'\{Q * Q'\}}$$

Dangling pointers



If the statement accesses a cell, then ownership of that cell should be in the precondition.

$$\{P\}[10] := 42\{Q\}$$

Here, P must imply the assertion $10 \mapsto - * \texttt{true}$

Bad example

$$\{10 \mapsto -\}[10] := 42 \parallel [10] := 6\{???\}$$

This would require:

$$10 \mapsto - \implies (10 \mapsto - * \texttt{true}) * (10 \mapsto - * \texttt{true})$$

Because of the separation property violation, this simplifies to:

$$10 \mapsto - \implies \texttt{false}$$

Simple example

We can prove this program is race-free, as long as that race is ruled out by the precondition.

$$\frac{\{x \mapsto 3\}[x] := 4\{x \mapsto 4\} \quad \{y \mapsto 3\}[y] := 5\{y \mapsto 5\}}{\{x \mapsto 3 * y \mapsto 3\}[x] := 4 \parallel [y] := 5\{x \mapsto 4 * y \mapsto 5\}}$$

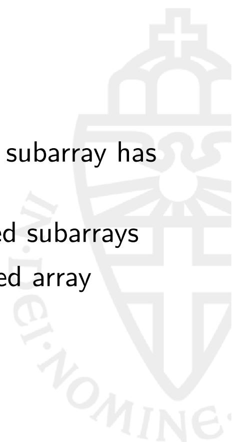
Alternative notation

$$\frac{\{x \mapsto 3\}[x] := 4\{x \mapsto 4\} \quad \{y \mapsto 3\}[y] := 5\{y \mapsto 5\}}{\{x \mapsto 3 * y \mapsto 3\}[x] := 4 \parallel [y] := 5\{x \mapsto 4 * y \mapsto 5\}}$$

Can be written as

$$\begin{array}{c} \{x \mapsto 3 * y \mapsto 3\} \\ \begin{array}{lcl} \{x \mapsto 3\} & & \{y \mapsto 3\} \\ [x] := 4 & \parallel & [y] := 5 \\ \{x \mapsto 4\} & & \{y \mapsto 5\} \end{array} \\ \{x \mapsto 4 * y \mapsto 5\} \end{array}$$

Merge sort



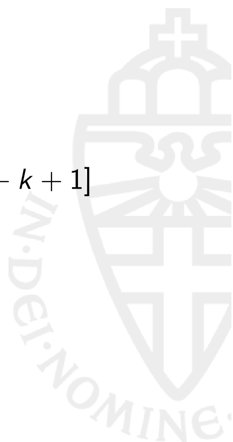
Merge sort is an efficient sorting algorithm:

1. Repeatedly split unsorted array in half until every subarray has 1 element
2. Repeatedly merge subarrays to produce new sorted subarrays
3. If only one subarray remains, this is the final sorted array

Good for parallelization

Merge sort predicates

$$
\begin{aligned}
\mathit{array}(a,i,j) &\triangleq \forall k. i \leq k \leq j \implies a+i \hookrightarrow - \\
\mathit{sorted}(a,i,j) &\triangleq \forall k. i \leq k < j \implies [a+k] \leq [a+k+1] \\
[E] \leq [F] &\triangleq \exists ab. E \hookrightarrow a \land F \hookrightarrow b \land a \leq b \\
E \hookrightarrow F &\triangleq E \mapsto F * \texttt{true}
\end{aligned}
$$



Merge sort

Parallel merge sort algorithm:

$\{array(a, i, j)\}$
$\texttt{procedure ms}(a, i, j);$
$\texttt{newvar}\ m := (i + j)/2;$
$\texttt{if}\ i < j\ \texttt{then}$
$\quad \texttt{ms}(a, i, m) \parallel \texttt{ms}(a, m + 1, j);$
$\quad \texttt{merge}(a, i, m + 1, j);$
$\{sorted(a, i, j)\}$

Merge sort race-free

$$
\begin{array}{ccc}
\multicolumn{3}{c}{\{array(a,i,m) * array(a,m+1,j)\}} \\
\{array(a,i,m)\} & & \{array(a,m+1,j)\} \\
\mathtt{ms}(a,i,m) & \| & \mathtt{ms}(a,m+1,j) \\
\{sorted(a,i,m)\} & & \{sorted(a,m+1,j)\} \\
\multicolumn{3}{c}{\{sorted(a,i,m) * sorted(a,m+1,j)\}}
\end{array}
$$

Ownership is granted per cell rather than per array.

Conclusion

- Racy vs race-free
- Daring vs Cautious
- Ownership hypothesis and Separation property
- Disjoint concurrency rule
- Examples

Thank you for your attention
Are there any questions?