An Asynchronous Soundness Theorem for Concurrent Separation Logic

Paul-André Melliès **Léo Stefanesco**

Institut de Recherche en Informatique Fondamentale (IRIF)
CNRS & Université Paris Diderot

CRECOGI+ELICA+GDRILL Plenary Meeting, 2018, Paris

Concurrent Separation Logic (CSL)

Concurrent Separation Logic is a logic for

concurrent programs with shared memory and locks.

Hoare triples

$$\Gamma \vdash \{P\}\ C\ \{Q\}$$

are proved using **derivation trees**

$$\begin{array}{c} \vdots\ \pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array}$$

Soundness theorem

Soundness theorem A

Starting from a state $\mathfrak{s} \vDash P$,

no execution of C will crash

and if $(C, \mathfrak{s}) \Downarrow \mathfrak{s}'$, then $\mathfrak{s}' \vDash Q$.

Soundness theorem B

Starting from $\mathfrak{s} \vDash P$,

no execution of C will produce any data races.

Data races

A **data race** occurs when two instructions are executed **in parallel** and either:

- one **writes** to a location the other is **reading**,
- or both instructions **write** to the **same** location.

$$x := 89 \quad \Big\| \quad y := x + z$$

Soundness theorem

Soundness theorem A

Starting from a state $\mathfrak{s} \vDash P$,

no execution of C will crash

and if $(C, \mathfrak{s}) \Downarrow \mathfrak{s}'$, then $\mathfrak{s}' \vDash Q$.

- depends on **individual** instructions

Soundness theorem B

Starting from $\mathfrak{s} \vDash P$,

no execution of C will produce any data races.

- depends on **pairs** of instructions

Soundness theorem

Soundness theorem A

Starting from a state $\mathfrak{s} \vDash P$,

no execution of C will crash

and if $(C, \mathfrak{s}) \Downarrow \mathfrak{s}'$, then $\mathfrak{s}' \vDash Q$.

- depends on **individual** instructions
- will be seen as a **1-dimensional** property

Soundness theorem B

Starting from $\mathfrak{s} \vDash P$,

no execution of C will produce any data races.

- depends on **pairs** of instructions
- will be seen as a **2-dimensional** property

A Concurrent Shared Memory Language

$$
\begin{array}{l}
\text{resource } r \text{ do} \\
\quad \text{while } x > 0 \text{ do} \\
\quad\quad \left\{ \begin{array}{c} x := x - 1; \\ \begin{array}{c} \text{with } r \text{ do} \\ y := y + 1 \end{array} \;\Big\|\; \begin{array}{c} \text{with } r \text{ do} \\ y := y + 1 \end{array} \end{array} \right\}
\end{array}
$$

Static Semantics of Imperative Languages

- For non-concurrent languages, a good abstraction for a program C is that of a **state transformer**:

$$\begin{aligned} [\![C]\!] : \quad & \textbf{States} \longrightarrow \textbf{States} \\ & \text{initial state} \longmapsto \text{final state} \end{aligned}$$

  This is enough for sequential composition.

- For concurrent languages, **more information is needed**: the final value of y in

$$x := 0; y := x$$

  can be **any value**, depending on what the **environment** does:

$$x := 0; y := x \;\; \| \;\; x := 77$$

Stateful traces

The **traces** contain the **state** at each step of the execution

$$[\![C]\!] \ni \mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2 \xrightarrow{m_1} \mathfrak{s}_3 \xrightarrow{env} \mathfrak{s}_4 \xrightarrow{m_2} \mathfrak{s}_5 \xrightarrow{env} \mathfrak{s}_6$$

where:

- the m_i are **elementary instructions**,

$$\text{eg.} \qquad \mathfrak{s} \xrightarrow{x := y+4z} \mathfrak{s}', \quad \mathfrak{s} \xrightarrow{P(r)} \mathfrak{s}';$$

- the *env* transitions are played by the **Environment**.

Example:

$$[\![x := y]\!] = \left\{ \mathfrak{s}_1 \xrightarrow{env} \mathfrak{s}_2 \xrightarrow{x:=y} \mathfrak{s}_2[x := \mathfrak{s}_1(y)] \xrightarrow{env} \mathfrak{s}_3 \mid \forall \mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3 \right\}$$

Stateless traces

The **traces** are sequences of **events**

$$[\![C]\!] \ni a_1\ a_2\ a_3\ \ldots\ a_{n-1}\ a_n$$

where:

- the events a_i are **independent** from the state,
- the **Environment** implies **non sequentially consistent traces**.

$$\text{eg.} \qquad \mathrm{Wr}(x, 89) \cdot \mathrm{Rd}(x, 70)$$

Our approach in this talk

Interpret programs as a **2-dimensional asynchronous graphs**.

Each program C will have **two related semantics**, corresponding to the **stateful** and the **stateless** trace semantics:

$$\boxed{[\![C]\!]_S \xrightarrow{\mathscr{L}} [\![C]\!]_L}$$

Our approach in this talk

Interpret programs as a **2-dimensional asynchronous graphs**.

Each program C will have **two related semantics**, corresponding to the **stateful** and the **stateless** trace semantics:

$$\boxed{[\![C]\!]_S \xrightarrow{\mathscr{L}} [\![C]\!]_L}$$

Interpret **derivation trees** of CSL as **asynchronous graphs**:

$$\boxed{\left[\!\!\left[\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array} \right]\!\!\right]_{Sep} \xrightarrow{\mathscr{S}} [\![C]\!]_S}$$

Our approach in this talk

Interpret programs as a **2-dimensional asynchronous graphs**.

Each program C will have **two related semantics**, corresponding to the **stateful** and the **stateless** trace semantics:

$$\boxed{[\![C]\!]_S \xrightarrow{\mathscr{L}} [\![C]\!]_L}$$

Interpret **derivation trees** of CSL as **asynchronous graphs**:

$$\boxed{\left[\!\!\left[\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array} \right]\!\!\right]_{Sep} \xrightarrow{\mathscr{S}} [\![C]\!]_S}$$

Soundness = properties of these maps of **asynchronous graphs**.

Asynchronous graphs

Definition

An **asynchronous graph** is

- a **graph** $G = (V, E, s, t)$, with $s, t : E \to V$,
- a **relation** $\diamond$ between paths $f, g : x \twoheadrightarrow y$ of length 2, with the same source and target nodes.

Two related paths form a **tile**.

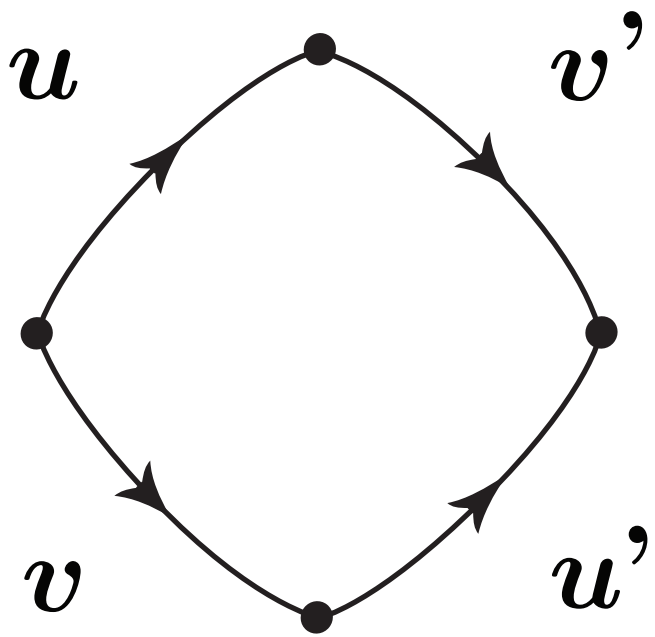
Asynchronous graphs

Definition

An **asynchronous graph** is

- a **graph** $G = (V, E, s, t)$, with $s, t : E \to V$,
- a **relation** $\diamond$ between paths $f, g : x \twoheadrightarrow y$ of length 2, with the same source and target nodes.

Two related paths form a **tile**.



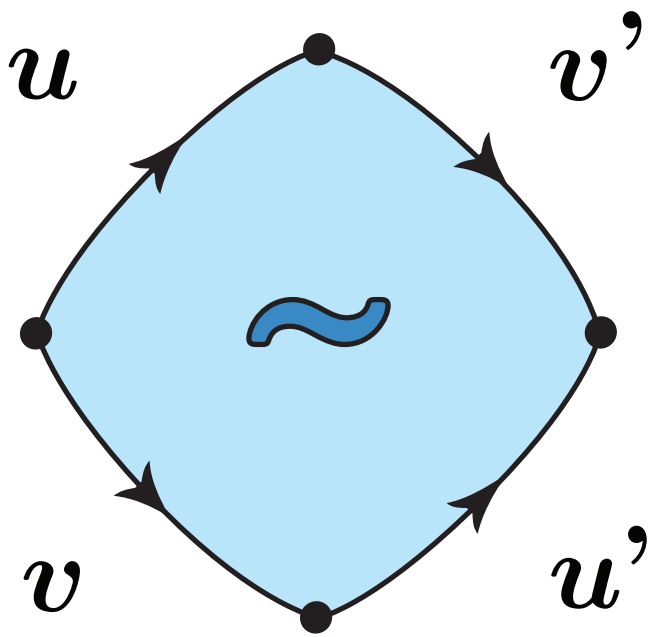
Asynchronous graphs

Definition

An **asynchronous graph** is

- a **graph** $G = (V, E, s, t)$, with $s, t : E \to V$,
- a **relation** $\diamond$ between paths $f, g : x \twoheadrightarrow y$ of length 2, with the same source and target nodes.

Two related paths form a **tile**.



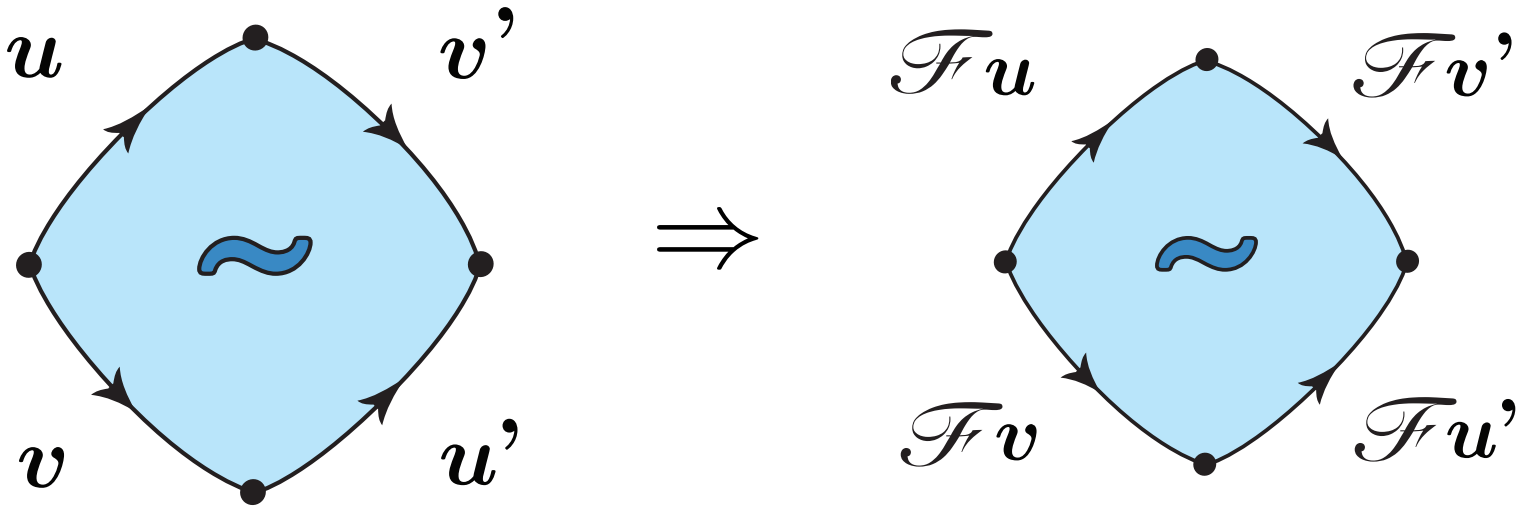
Maps between asynchronous graphs

Definition

A **map of asynchronous graphs**

$$\mathscr{F} \quad : \quad (G, \diamond) \longrightarrow (G', \diamond')$$

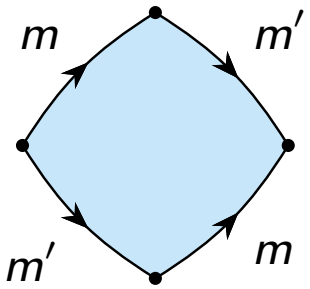
is a **graph homomorphism** $\mathscr{F} : G \to G'$ that maps **tiles into tiles**.



Can be seen as a **labeling** for G.

The stateful machine model

The **stateful machine model** $\mathcal{H}_S$ is defined as follows:

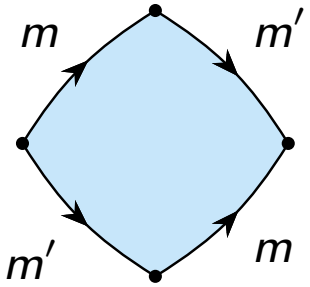
- its **nodes** are machine states $\mathfrak{s} \in (\mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val}) \times \mathscr{P}(Locks)$,
- there is an **edge** $\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$ whenever $[\![m]\!](\mathfrak{s}) = \mathfrak{s}'$,
- a square  is a **tile** when

1. m and m' **do not synchronize**: $\mathsf{lock}(m) \cap \mathsf{lock}(m') = \emptyset$
2. and do **not** induce a **data race**:

$$\begin{aligned} (\,\mathrm{rd}(m\,) \cup \mathrm{wr}(m\,)\,) \cap \mathrm{wr}(m') &= \emptyset \\ (\,\mathrm{rd}(m') \cup \mathrm{wr}(m')\,) \cap \mathrm{wr}(m\,) &= \emptyset \end{aligned}$$

The stateless machine model

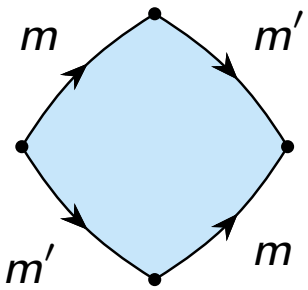
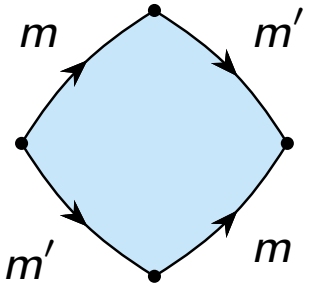
The **stateless machine model** $\pm_L$ is defined as follows:

- its **nodes** are lock states $L \subseteq Locks$,
- there is an **edge** $L \xrightarrow{m} L'$ whenever $[\![m]\!](L) = L'$,
- a square 

is a **tile** when

  m and m' **do not synchronize**: $\mathrm{lock}(m) \cap \mathrm{lock}(m') = \emptyset$.

Machine Models

	$\pitchfork_S$	$\pitchfork_L$
Nodes	**machine states** $\mathfrak{s} = (\mu, L)$ with • *memory state* $\mu : \mathbf{Loc} \rightharpoonup_{fin} \mathbf{Val}$ • *lock state* $L \subseteq Locks$	**lock states** $L \subseteq Locks$
Edges	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$ for each $[\![m]\!](\mathfrak{s}) = \mathfrak{s}'$	$L \xrightarrow{m} L'$ for $[\![m]\!](L) = L'$
Tiles	**data race-freedom**  is a tile when: $\mathsf{lock}(m) \cap \mathsf{lock}(m') = \emptyset$ $(\mathrm{rd}(m) \cup \mathrm{wr}(m)) \cap \mathrm{wr}(m') = \emptyset$ $(\mathrm{rd}(m') \cup \mathrm{wr}(m')) \cap \mathrm{wr}(m) = \emptyset$	**parallelism**  is a tile when: $\mathsf{lock}(m) \cap \mathsf{lock}(m') = \emptyset$

Asynchronous Transition Systems (ATS)

Definition (ATS)

An **ATS** over a **machine model** $\mathcal{M}$ is an **asynchronous graph** G together with a **map of asynchronous graphs**

$$\lambda \quad : \quad G \quad \longrightarrow \quad \mathcal{M}$$

with a partition on G's edges, for the **Code** and the **Environment**

- **nodes** are labeled by **states**,
- **edges** are labeled by **instructions** m,
- **labels** are consistent with the semantics of **instructions**:

$$x \xrightarrow{m} y \qquad \Longrightarrow \qquad [\![m]\!]\big(\lambda(x)\big) = \lambda(y)$$

Semantics of the Code

The semantics of a **program** C is a pair of **related ATSs**.

$$[\![C]\!]_S : \quad G_S(C) \longrightarrow \pm_S$$

$$[\![C]\!]_L : \quad G_L(C) \longrightarrow \pm_L$$

Definition

The semantics is defined by **induction of the structure of the Code**:

$$[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!], \quad [\![C_1 \parallel C_2]\!] = [\![C_1]\!] \parallel [\![C_2]\!]$$

where $[\![\cdot]\!]$ is either stateful $[\![\cdot]\!]_S$ or stateless $[\![\cdot]\!]_L$

Semantics of the Code

The semantics of a **program** C is a pair of **related ATSs**.

$$
\begin{tikzcd}
[\![C]\!]_S : & G_S(C) \arrow[r] \arrow[d, dashed, "\mathscr{L}"'] & \text{⅄}_S \arrow[d, dashed, "(\mu,L)\mapsto L"] \\
[\![C]\!]_L : & G_L(C) \arrow[r] & \text{⅄}_L
\end{tikzcd}
$$

Definition

The semantics is defined by **induction of the structure of the Code**:

$$[\![C_1 \,;\, C_2]\!] = [\![C_1]\!] \,;\, [\![C_2]\!], \quad [\![C_1 \parallel C_2]\!] = [\![C_1]\!] \parallel [\![C_2]\!]$$

where $[\![\cdot]\!]$ is either stateful $[\![\cdot]\!]_S$ or stateless $[\![\cdot]\!]_L$

The parallel product

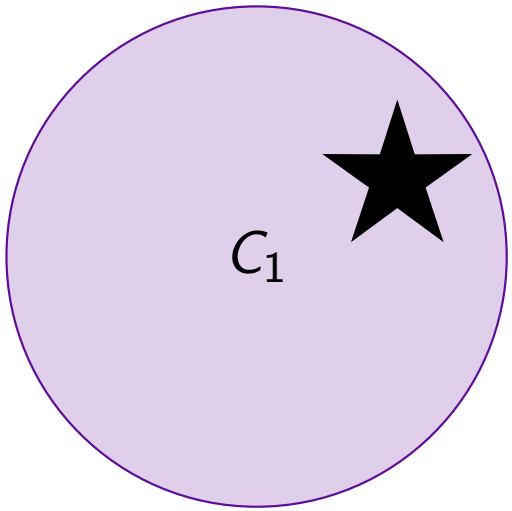
C_1 $\qquad$ $C_1 \parallel C_2$ $\qquad$ C_2

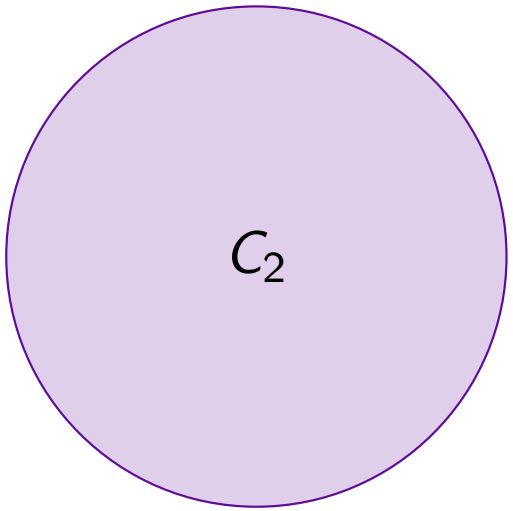
The parallel product

C_1 $\qquad$ $C_1 \parallel C_2$ $\qquad$ C_2

$$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$$

The parallel product





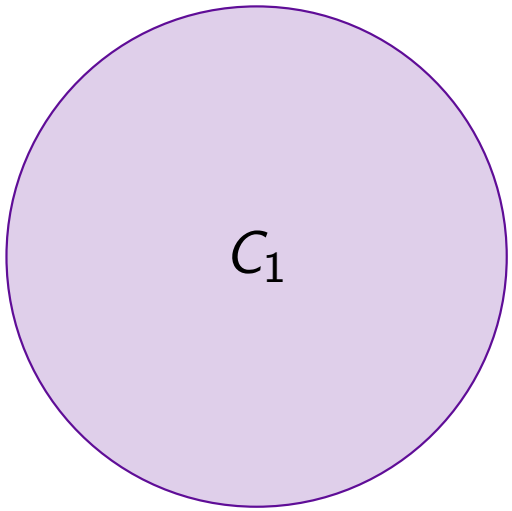
C_1 $\qquad$ $C_1 \parallel C_2$ $\qquad$ C_2

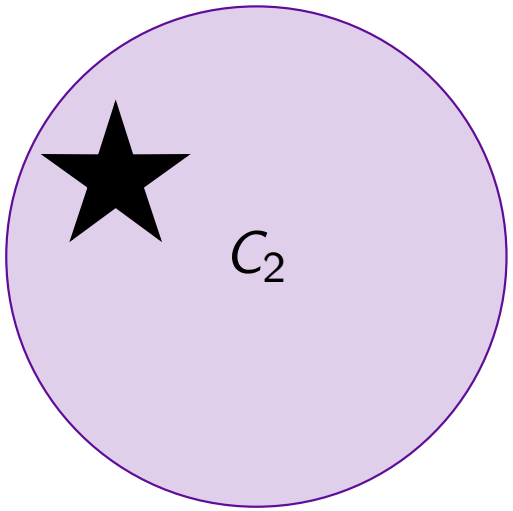
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$ $\qquad$ $\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$

The parallel product

C_1	$C_1 \parallel C_2$	C_2
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$

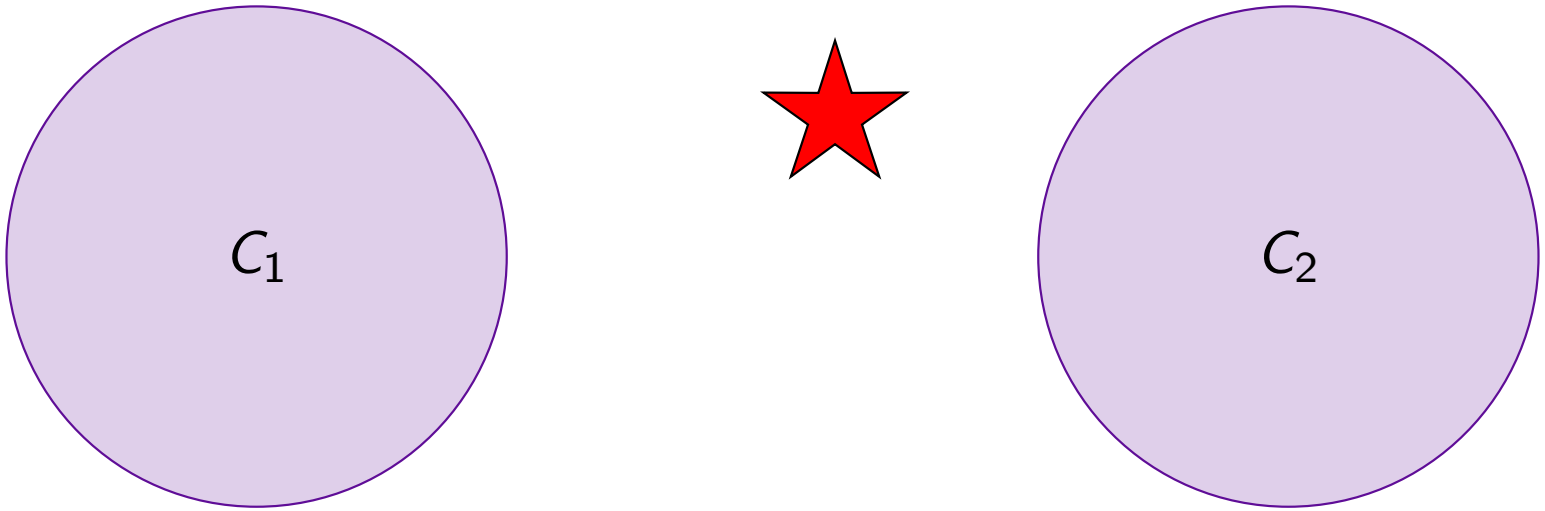
The parallel product





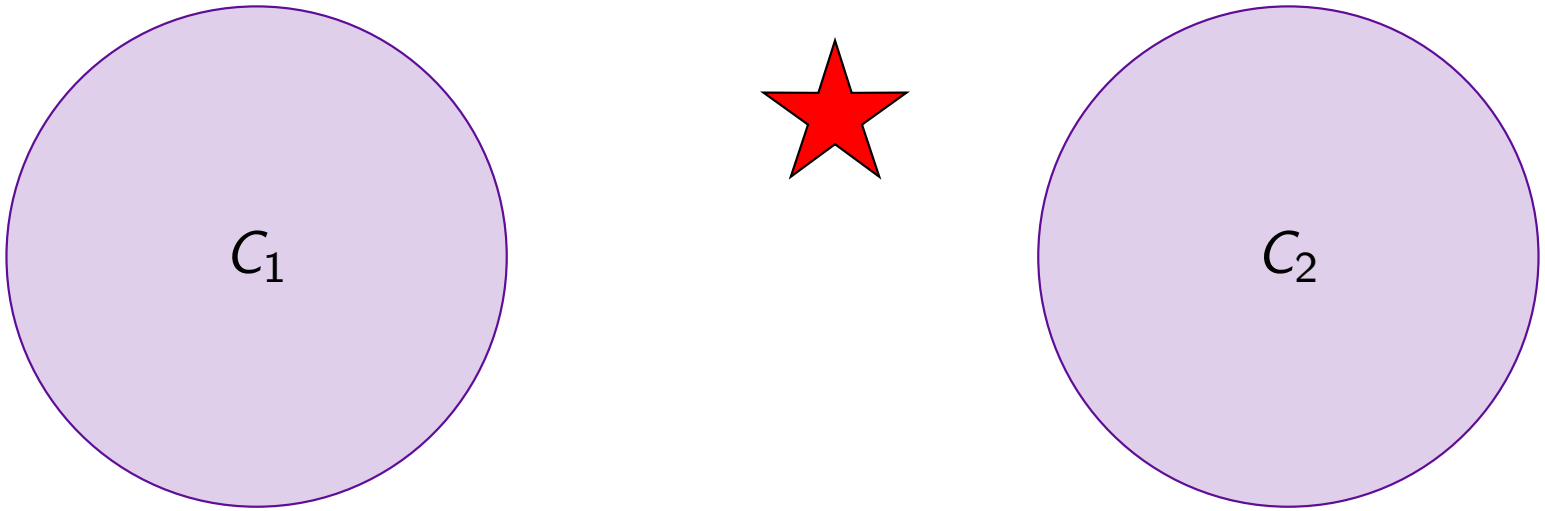
C_1	$C_1 \parallel C_2$	C_2
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$

The parallel product



C_1	$C_1 \parallel C_2$	C_2
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$		$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$

The parallel product



C_1	$C_1 \parallel C_2$	C_2
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$
$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$	$\mathfrak{s} \xrightarrow{m} \mathfrak{s}'$

The parallel product

$$G_1 \xrightarrow{\lambda_1} \perp,\ G_2 \xrightarrow{\lambda_2} \perp \quad \longmapsto \quad G_1 \parallel G_2 \xrightarrow{\lambda_{1\|2}} \perp$$

Nodes of $G_1 \parallel G_2$: $x_1|x_2 \in G_1 \times G_2$, such that $\lambda_1(x_1) = \lambda_2(x_2)$

$$\lambda_{1\|2}(x_1|x_2) := \lambda_1(x_1) = \lambda_2(x_2)$$

Three types of transitions, e.g. $x_1|x_2 \xrightarrow{m|m} x_1'|x_2'$ is a pair:

$$x_1 \xrightarrow{m} x_1' \in G_1 \quad \text{and} \quad x_2 \xrightarrow{m} x_2' \in G_2$$

This transition is mapped by $\lambda_{1\|2}$ to $\lambda_{1\|2}(x_1|x_2) \xrightarrow{m} \lambda_{1\|2}(x_1'|x_2')$

Tiles of $G_1 \parallel G_2$ are tiles of G_1, of G_2, or made of instructions of the form

$$m_1|m_1 \text{ and } m_2|m_2$$

Data races in the stateless semantics $[\![C]\!]_L$

A **data race** needs two **unsynchronized** instructions in some trace

$$\cdots \rightarrow \bullet \xrightarrow{m_1} \bullet \xrightarrow{m_2} \bullet \rightarrow \cdots$$

Data races in the stateless semantics $[\![C]\!]_L$

A **data race** needs two **unsynchronized** instructions in some trace

$$\cdots \rightarrow \bullet \xrightarrow{m_1} \bullet \xrightarrow{m_2} \bullet \rightarrow \cdots$$

thus coming with **another schedule** of m_1 and m_2:

$$\cdots \rightarrow \bullet \xrightarrow{m_2} \bullet \xrightarrow{m_1} \bullet \rightarrow \cdots$$

Data races in the stateless semantics $[\![C]\!]_L$

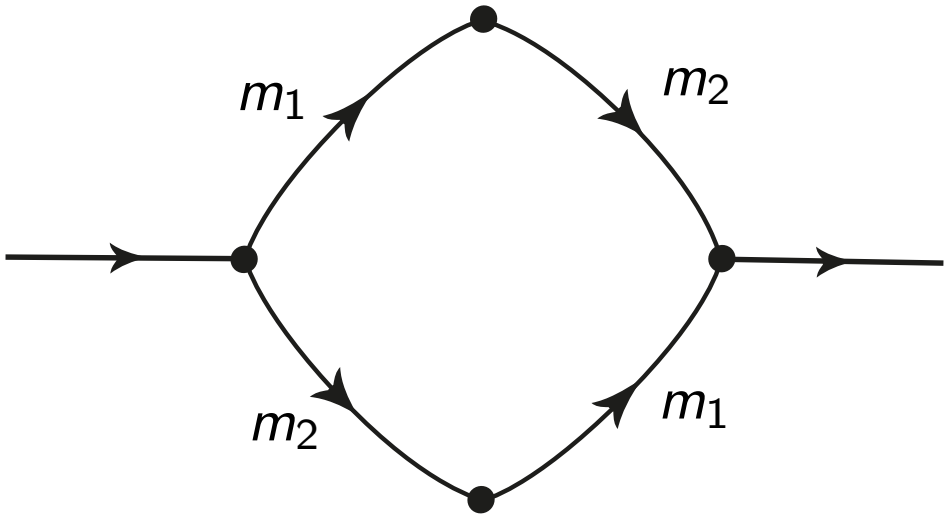
A **data race** needs two **unsynchronized** instructions in some trace

$$\cdots \rightarrow \bullet \xrightarrow{m_1} \bullet \xrightarrow{m_2} \bullet \rightarrow \cdots$$

thus coming with **another schedule** of m_1 and m_2:

$$\cdots \rightarrow \bullet \xrightarrow{m_2} \bullet \xrightarrow{m_1} \bullet \rightarrow \cdots$$

In our approach, we turn it into a square



Data races in the stateless semantics $[\![C]\!]_L$

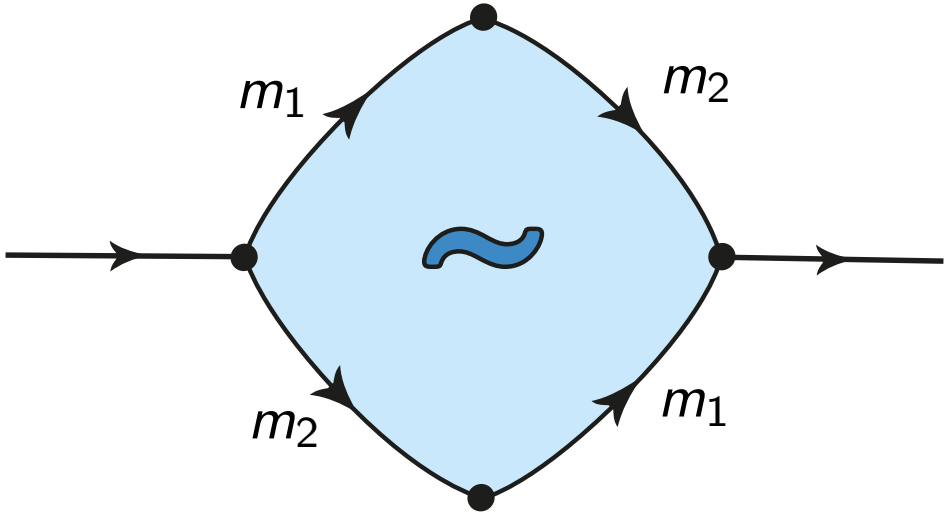
A **data race** needs two **unsynchronized** instructions in some trace

$$\cdots \rightarrow \bullet \xrightarrow{m_1} \bullet \xrightarrow{m_2} \bullet \rightarrow \cdots$$

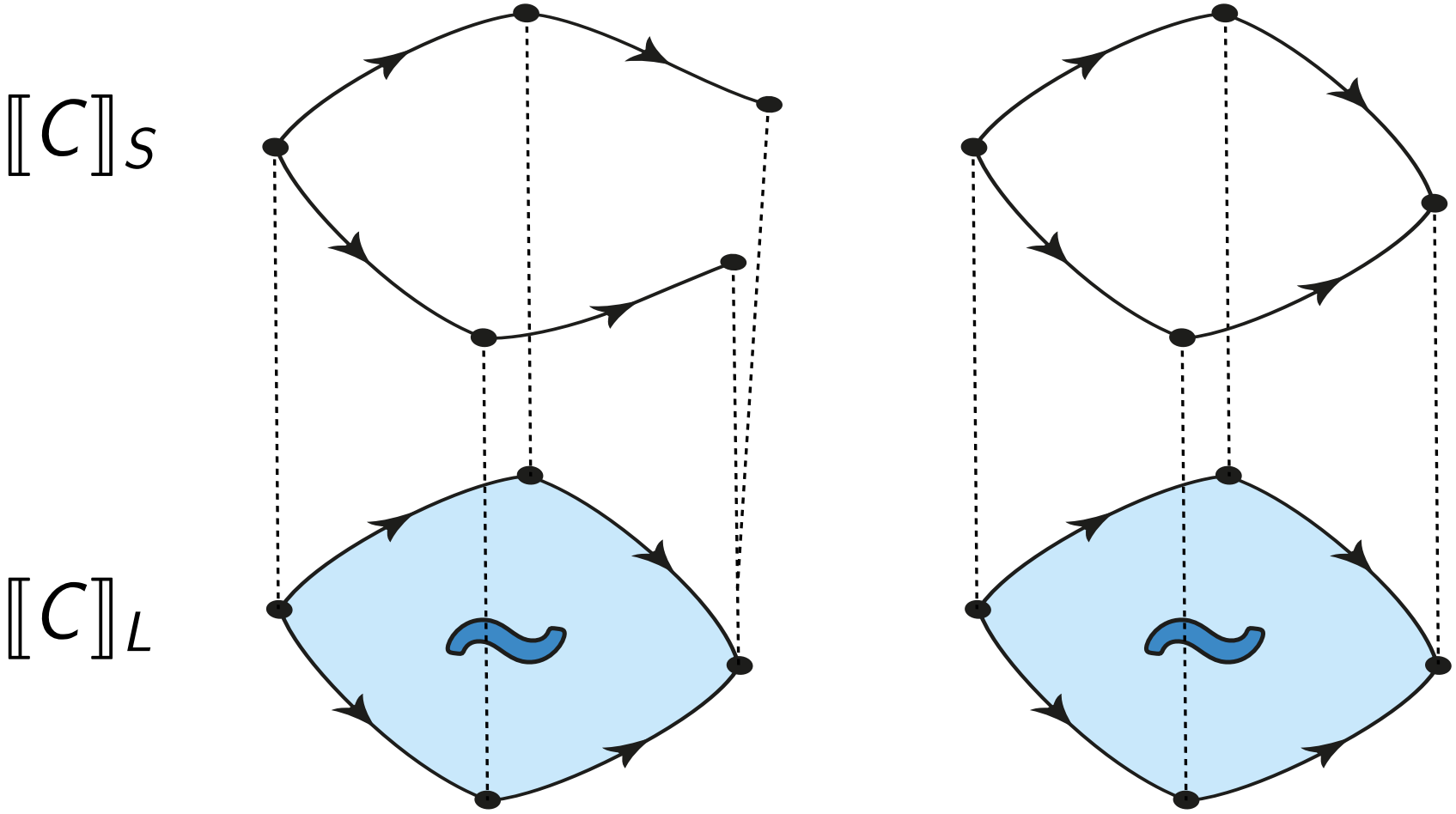
thus coming with **another schedule** of m_1 and m_2:

$$\cdots \rightarrow \bullet \xrightarrow{m_2} \bullet \xrightarrow{m_1} \bullet \rightarrow \cdots$$

In our approach, we turn it into a **tile**



A topological account of data races



Data race = **stateful hole** above a **stateless tile**

Separation Logic (sequential)

Hoare logic with extended predicates:

$$\vdash \{P\}\ C\ \{Q\}$$

Meaning of triple: $\forall \sigma, \sigma', (\sigma \vDash P \ \wedge\ C, \sigma \Downarrow \sigma') \ \Rightarrow\ \sigma' \vDash Q$

Predicates on the memory

$P, Q, J ::= \top \mid \bot \mid P \vee Q \mid P \wedge Q \mid \ \forall v.P \mid \ \exists v.P \mid \ P * Q \mid \mathbf{emp} \mid v \mapsto w$

$$\begin{aligned}
\sigma \vDash P \wedge Q &\iff \sigma \vDash P \text{ and } \sigma \vDash Q \\
\sigma \vDash P * Q &\iff \exists \sigma_1 \sigma_2,\ \sigma = \sigma_1 \uplus \sigma_2 \text{ and } \sigma_1 \vDash P \text{ and } \sigma_2 \vDash Q \\
\sigma \vDash \mathbf{emp} &\iff \sigma = \emptyset \\
\sigma \vDash v \mapsto w &\iff \sigma = [v \mapsto w]
\end{aligned}$$

The Frame Rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

C depends only on the **resource** P

The Frame Rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

C depends only on the **resource** P

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \quad \vdash \{P_2\}\ C_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

The Frame Rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

C depends only on the **resource** P

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \quad \vdash \{P_2\}\ C_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}$$

But C_1 and C_2 **cannot communicate!**

Concurrent Separation Logic

Associate an **invariant** J to each **lock** r in a context:

$$r_1{:}J_1, \ldots, r_n{:}J_n \vdash \{P\}\ C\ \{Q\}$$

Inference rules:

$$\frac{\Gamma \vdash \{P * J\}\ C\ \{Q * J\}}{\Gamma, r : J \vdash \{P\}\ \texttt{with}\ r\ \texttt{do}\ C\ \{Q\}}$$

$$\frac{\Gamma, r : J \vdash \{P\}\ C\ \{Q\}}{\Gamma \vdash \{P * J\}\ \texttt{resource}\ r\ \texttt{do}\ C\ \{Q * J\}}$$

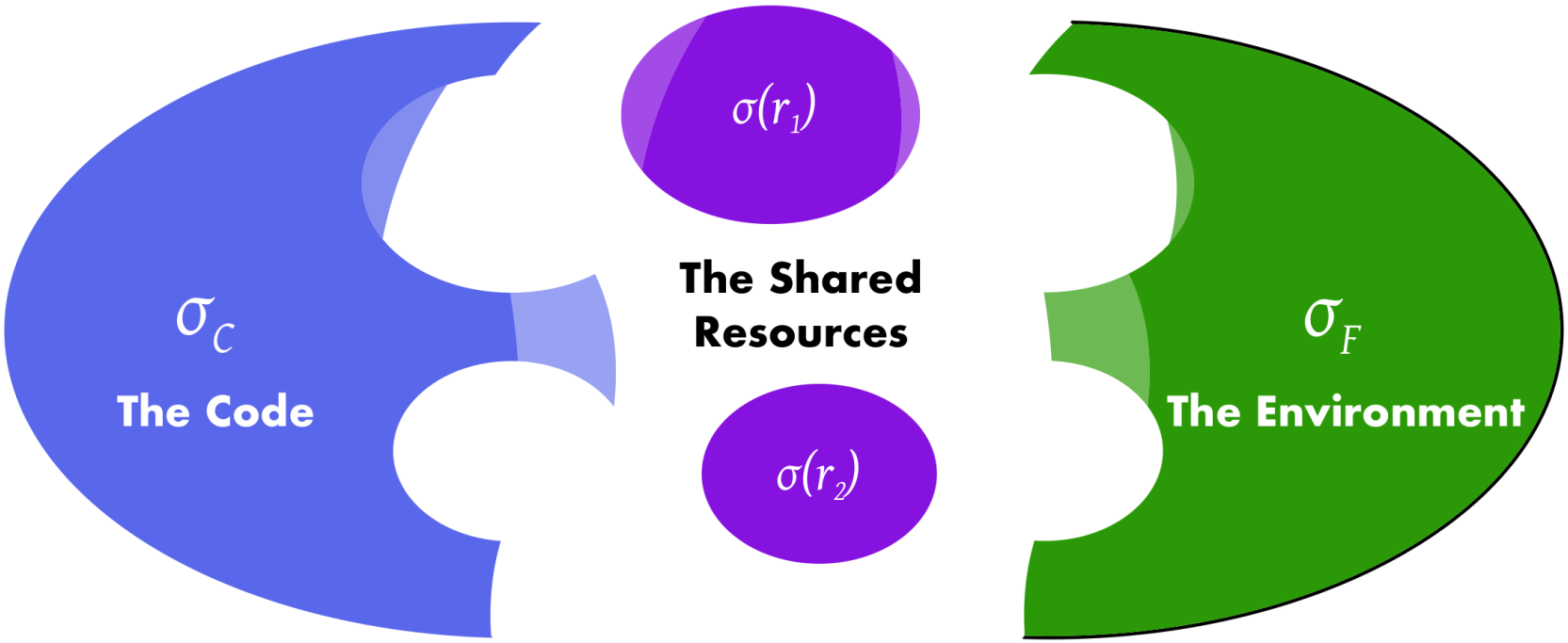
Separated States

A **separated state** is a triple

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$$

$$\pitchfork$$

$$\textbf{States} \times (\mathit{Locks} \to \textbf{States} + \{C, F\}) \times \textbf{States}$$



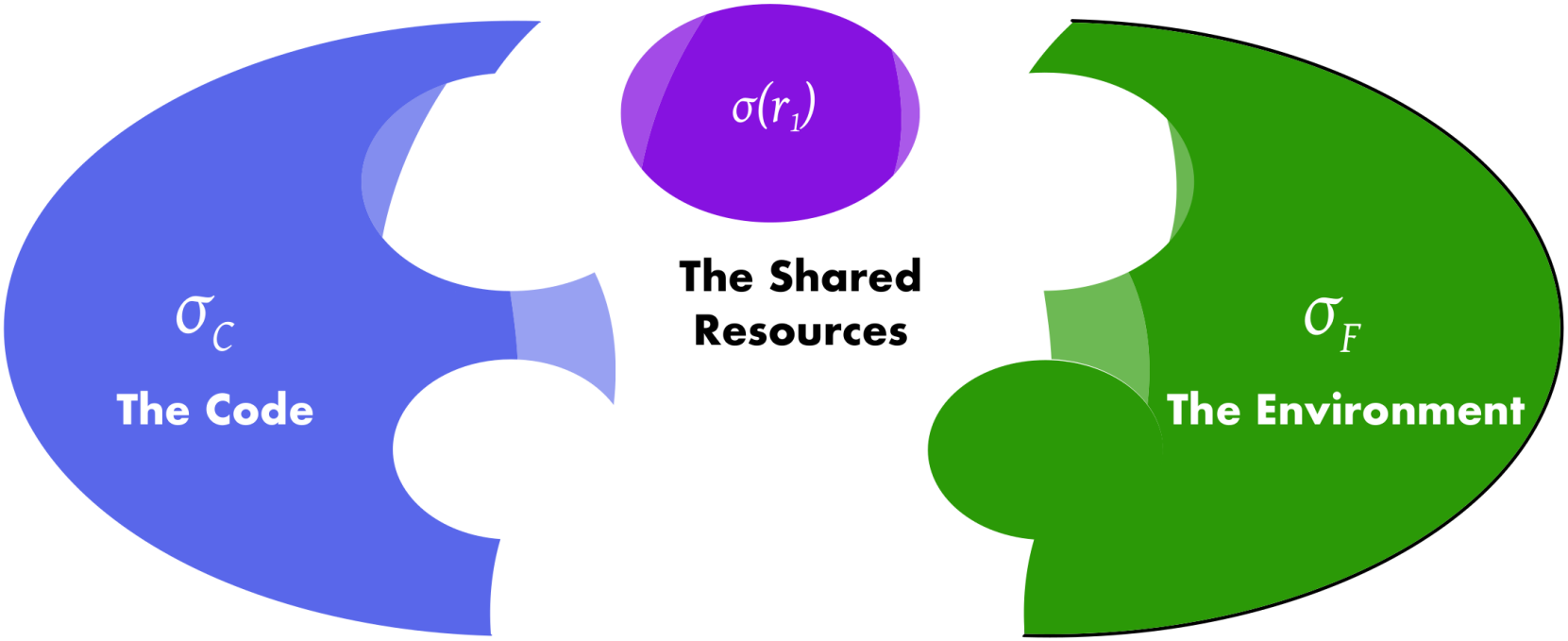
Separated States

A **separated state** is a triple

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$$

$$\pitchfork$$

$$\textbf{States} \times (\mathit{Locks} \to \textbf{States} + \{C, F\}) \times \textbf{States}$$



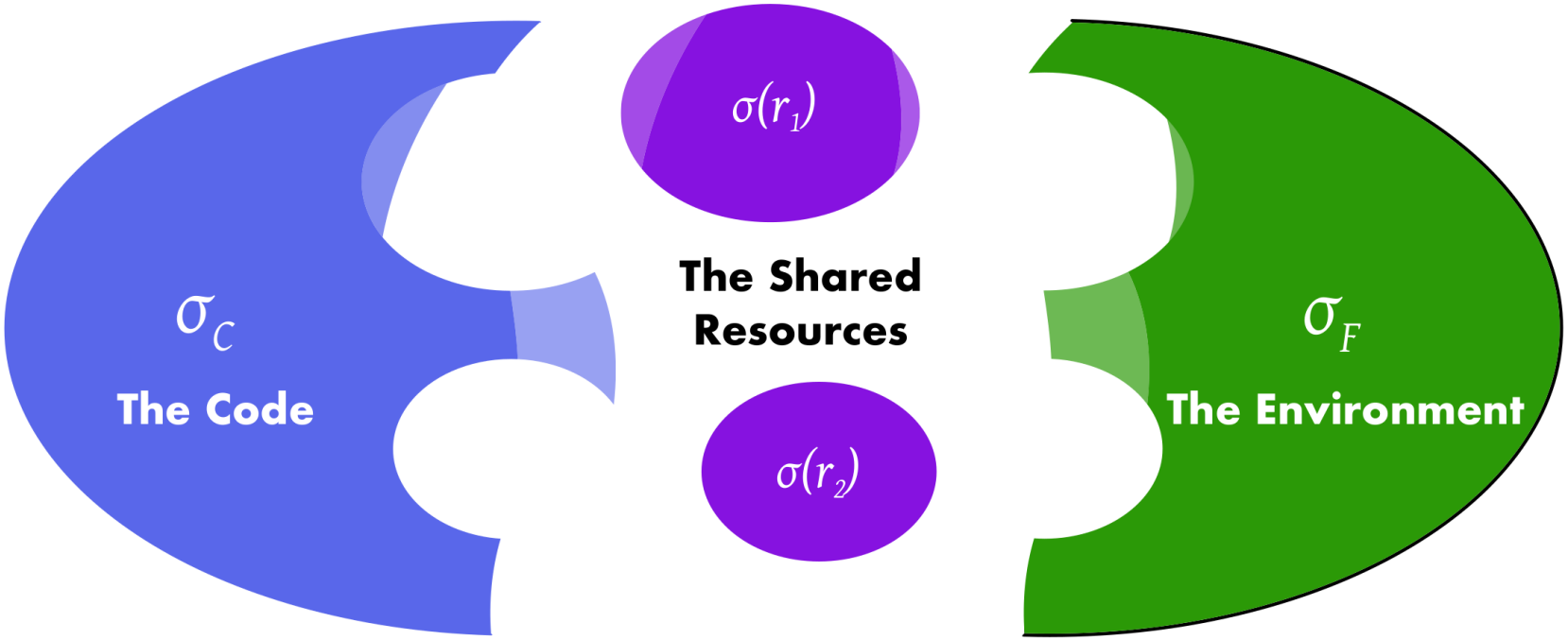
Separated States

A **separated state** is a triple

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$$

$$\Cap$$

$$\textbf{States} \times (\mathit{Locks} \to \textbf{States} + \{C, F\}) \times \textbf{States}$$



The separated machine model

The **separated machine model** $\pitchfork_{Sep}$ is defined in a similar fashion.

Its nodes are **separated states** $(\sigma_C, \boldsymbol{\sigma}, \sigma_F)$,

and there are two kinds of **transitions**

$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{m} (\sigma'_C, \boldsymbol{\sigma}', \sigma_F)$$
$$(\sigma_C, \boldsymbol{\sigma}, \sigma_F) \xrightarrow{m} (\sigma_C, \boldsymbol{\sigma}', \sigma'_F)$$

There is a **map of asynchronous graphs**

$$\pitchfork_{Sep} \longrightarrow \pitchfork_S$$

Semantics of the derivation trees

Every **CSL derivation tree** $\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array}$ is interpreted as an **ATS**:

$$[\![\pi]\!]_{Sep} : \quad G_{Sep}(\pi) \longrightarrow \mathbb{t}_{Sep}$$

$$\begin{tikzcd}
{[\![C]\!]_S :} & G_S(C) \arrow[r] \arrow[d, dashed, "\mathscr{L}_G"'] & \mathbb{t}_S \arrow[d, dashed, "(\mu,L)\mapsto L"] \\
{[\![C]\!]_L :} & G_L(C) \arrow[r] & \mathbb{t}_L
\end{tikzcd}$$

$[\![C]\!]_S \xrightarrow{\mathscr{L}} [\![C]\!]_L$

Semantics of the derivation trees

Every **CSL derivation tree** $\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array}$ is interpreted as an **ATS**:

$$
\begin{tikzcd}
[\![\pi]\!]_{Sep} : & G_{Sep}(\pi) \arrow[r] \arrow[d, dashed, "\mathscr{S}"'] & \mathfrak{t}_{Sep} \arrow[d, dashed] \\
[\![C]\!]_{S} : & G_{S}(C) \arrow[r] \arrow[d, dashed, "\mathscr{L}"'] & \mathfrak{t}_{S} \arrow[d, dashed, "(\mu,L)\mapsto L"] \\
[\![C]\!]_{L} : & G_{L}(C) \arrow[r] & \mathfrak{t}_{L}
\end{tikzcd}
$$

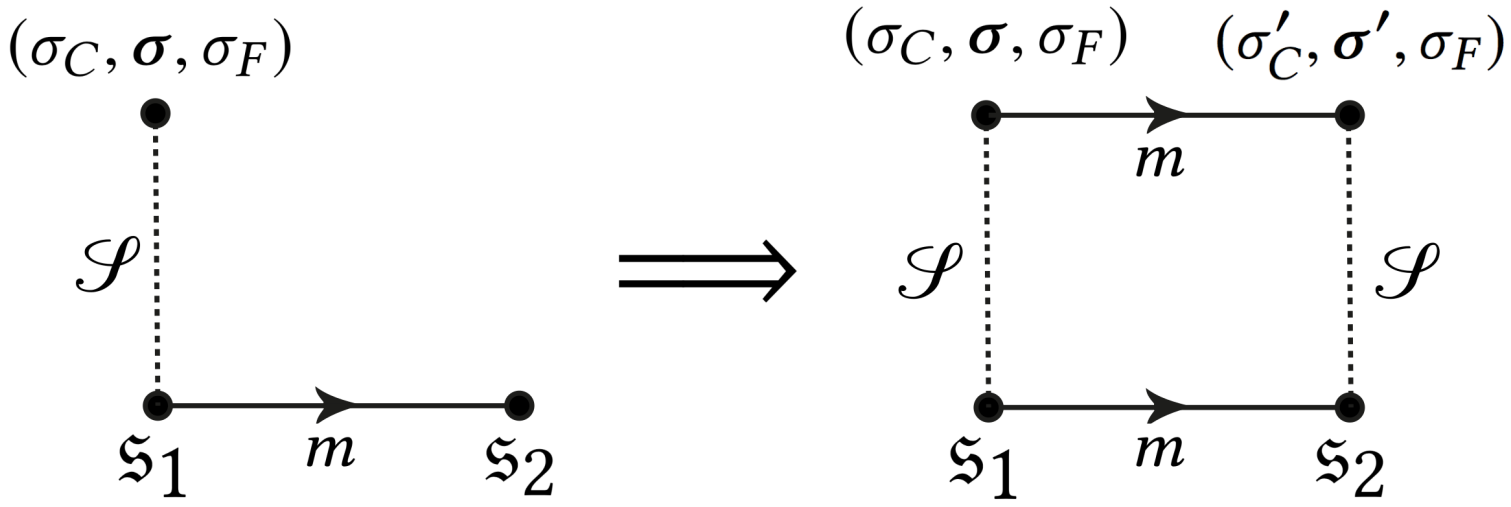
An asynchronous soundness theorem

Theorem 1

The map of asynchronous graphs

$$\left[\!\!\left[\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array} \right]\!\!\right]_{Sep} \xrightarrow{\quad \mathcal{S} \quad} [\![C]\!]_S$$

is a **1-dimensional fibration on Code transitions**.



An asynchronous soundness theorem

Theorem 2

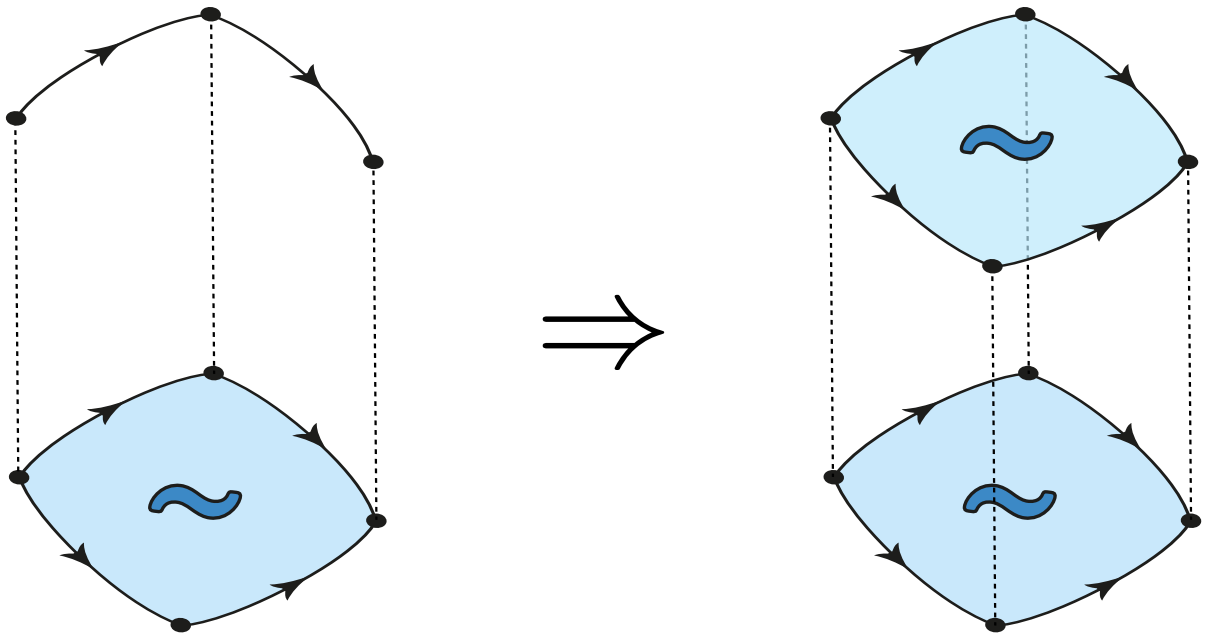
The map of asynchronous graphs

$$\left[\!\!\left[\begin{array}{c} \vdots\,\pi \\ \Gamma \vdash \{P\}\ C\ \{Q\} \end{array} \right]\!\!\right]_{Sep} \xrightarrow{\quad \mathscr{S} \quad} [\![C]\!]_S \xrightarrow{\quad \mathscr{L} \quad} [\![C]\!]_L$$

is a **2-dimensional fibration**.

$[\![\pi]\!]_{Sep}$

$[\![C]\!]_L$



Conclusion & Future work

A topological account of **data races**

A **truly concurrent** semantics for Concurrent Separation Logic
derivation trees gives a **strategy** for **memory management**

Extension to more sophisticated logics and programming languages
eg. higher order, more general concurrency primitives

Understand our semantics in a general categorical framework

Conclusion & Future work

A topological account of **data races**

A **truly concurrent** semantics for Concurrent Separation Logic
derivation trees gives a **strategy** for **memory management**

Extension to more sophisticated logics and programming languages
eg. higher order, more general concurrency primitives

Understand our semantics in a general categorical framework

Thank you!