# Reconciling nondeterminism and causality

## Event structures for weak memory

Simon Castellan[1]
(Joint work with Jade Alglave and Jean-Marie Madiot.)

[1]Imperial College London, UK

27 november 2018

# Reasoning on concurrent programs

Consider the program `mp`:

$$\begin{array}{c}
\mathit{data} = \mathit{flag} = 0 \\
\begin{array}{l|l}
\mathit{data} := 17; & r \leftarrow \mathit{flag}; \\
\mathit{flag} := 1 & v \leftarrow \mathit{data}
\end{array}
\end{array}$$

Does $\mathtt{mp} \models r = 1 \Rightarrow v = 17$?

# Reasoning on concurrent programs

Consider the program `mp`:

$$
\begin{array}{c}
data = flag = 0 \\
\begin{array}{l} data := 17; \\ flag := 1 \end{array}
\;\Bigg\|\;
\begin{array}{l} r \leftarrow flag; \\ v \leftarrow data \end{array}
\end{array}
$$

Does $\texttt{mp} \models r = 1 \Rightarrow v = 17$?

Two main solutions to prove this:

- **Operational** semantics formalises the **machine**
- **Axiomatic** semantics formalises the **executions**

# Operational semantics: machines as LTSs

Formalises an **abstract machine** running the program:

$$\langle (x := 1; t \parallel p) \odot \mu \rangle \xrightarrow{\mathtt{W}_{x:=1}} \langle (t \parallel p) \odot \mu[x := 1] \rangle.$$

Transitions labelled by an action in $\Sigma ::= \mathtt{W}_{x:=k} \mid \mathtt{R}_{x=k} \mid \ldots$.

Executions of the program become **traces** of the LTS:

$$\langle \mathtt{mp} \odot \mu \rangle \xrightarrow{\mathtt{W}_{data:=\mathbf{17}}} \xrightarrow{\mathtt{W}_{flag:=\mathbf{1}}} \xrightarrow{\mathtt{R}_{flag=\mathbf{1}}} \xrightarrow{\mathtt{R}_{value=\mathbf{17}}}$$

# Operational semantics: machines as LTSs

Formalises an **abstract machine** running the program:

$$\langle (x := 1; t \parallel p) \odot \mu \rangle \xrightarrow{\mathtt{W}_{x:=1}} \langle (t \parallel p) \odot \mu[x := 1] \rangle.$$

Transitions labelled by an action in $\Sigma ::= \mathtt{W}_{x:=k} \mid \mathtt{R}_{x=k} \mid \ldots$.

Executions of the program become **traces** of the LTS:

$$\langle \mathtt{mp} \odot \mu \rangle \xrightarrow{\mathtt{W}_{data:=\mathbf{17}}} \xrightarrow{\mathtt{W}_{flag:=\mathbf{1}}} \xrightarrow{\mathtt{R}_{flag=\mathbf{1}}} \xrightarrow{\mathtt{R}_{value=\mathbf{17}}}$$

$\oplus$ Represents **nondeterministic branching points**.
$\rightsquigarrow$ Liveness properties, whole program optimisations.

$\ominus$ Combinatorial explosion due to **interleaving**.
$\rightsquigarrow$ Hard to simulate, hard to reason on.

# Axiomatic semantics

Formalises a program by the set of its valid **executions**:

$$\text{program} \overset{\textbf{syntax}}{\rightsquigarrow} \underbrace{\text{execution candidates}}_{\text{set of events+relations}} \overset{\textbf{model}}{\rightsquigarrow} \text{executions}$$

Two candidates for `mp`:

$$\begin{tikzcd}
\mathtt{W}_{\textit{data}:=17} \arrow[d, "\mathsf{po}"'] & \mathtt{R}_{\textit{flag}=0} \arrow[d, "\mathsf{po}"] \\
\mathtt{W}_{\textit{flag}:=1} & \mathtt{R}_{\textit{data}=0}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\mathtt{W}_{\textit{data}:=17} \arrow[d, "\mathsf{po}"'] & \mathtt{R}_{\textit{flag}=1} \arrow[d, "\mathsf{po}"] \\
\mathtt{W}_{\textit{flag}:=1} \arrow[ur, "\mathsf{rf}"] & \mathtt{R}_{\textit{data}=0}
\end{tikzcd}$$

valid on all architectures — valid on some (eg. ARM)

# Axiomatic semantics

Formalises a program by the set of its valid **executions**:

$$\text{program} \overset{\textbf{syntax}}{\rightsquigarrow} \underbrace{\text{execution candidates}}_{\text{set of events+relations}} \overset{\textbf{model}}{\rightsquigarrow} \text{executions}$$

Two candidates for `mp`:

$$\begin{tikzcd} \mathtt{W}_{\mathit{data}:=17} \arrow[d, "\mathsf{po}"'] & \mathtt{R}_{\mathit{flag}=0} \arrow[d, "\mathsf{po}"] \\ \mathtt{W}_{\mathit{flag}:=1} & \mathtt{R}_{\mathit{data}=0} \end{tikzcd} \qquad \begin{tikzcd} \mathtt{W}_{\mathit{data}:=17} \arrow[d, "\mathsf{po}"'] & \mathtt{R}_{\mathit{flag}=1} \arrow[d, "\mathsf{po}"] \\ \mathtt{W}_{\mathit{flag}:=1} \arrow[ur, "\mathsf{rf}"] & \mathtt{R}_{\mathit{data}=0} \end{tikzcd}$$

valid on all architectures    valid on some (eg. ARM)

- ⊕ **Causal** account of executions.
  $\rightsquigarrow$ Easy to simulate; allows higher-level reasoning.
- ⊖ **Per-execution** modelling of the program.
  $\rightsquigarrow$ No grip on the nondeterministic branching point

# The best of both worlds: event structures

**Operational**
Machines & transitions
Conflict $\sim$ between transitions

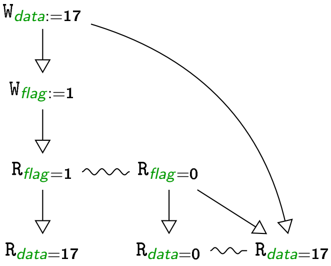
**Axiomatic**
Execution-level events
Causality $\rightarrowtriangle$ on each execution

add causality to transitions

globalise events

**Event structures**
global notion of events ($\approx$ transitions)
events equiped with $\sim$ and $\rightarrowtriangle$

mp:



# The best of both worlds: event structures

**Operational**
Machines & transitions
Conflict $\sim$ between transitions

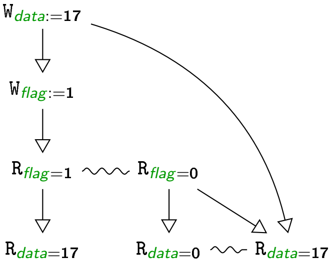
**Axiomatic**
Execution-level events
Causality $\rightarrowtriangle$ on each execution

add causality to transitions

globalise events

**Event structures**
global notion of events ($\approx$ transitions)
events equiped with $\sim$ and $\rightarrowtriangle$

mp:




Maximal conflict-free subsets $\leftrightarrow$ Axiomatic executions.

# Outline of the talk

(1) **From programs to event structures.**
The sequentially consistent case.

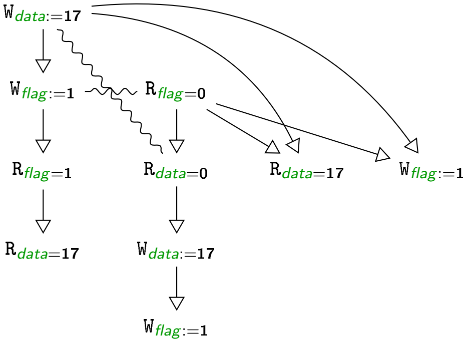
(2) **A strong data-race-free theorem for TSO.**
Which preserves liveness properties.

(3) **Relaxing coherence.**
Improving over the co of axiomatic semantics.

(4) **Beyond assembly: higher-order languages**
When labels become moves.

# I. From programs to event structures: naive SC

$[\![\mathtt{mp}]\!] =$



# Our language

We consider a simple imperative language:

$$\begin{aligned}
e &::= r \mid e + e \mid \ldots && \textbf{expressions}\\
t &::= \epsilon \mid x := e; t \mid r \leftarrow x; t && \textbf{threads}\\
&\quad \mid \texttt{output}\ e \mid r \leftarrow \texttt{input} &&\\
&\quad \mid \texttt{if}\ (0 == e)\ \{t\}\ \{t\} &&\\
p &::= t \parallel \ldots \parallel t && \textbf{programs}
\end{aligned}$$

- Features *global variables* and *thread registers*
- Input / Output instructions used as "observation points"

Traditional LTS on states $\langle p \odot \mu : V \to \mathbb{N}\rangle$ labeled over:

$$\Sigma_{\mathsf{SC}} ::= \mathtt{R}_{x=k} \mid \mathtt{W}_{x:=k} \mid \mathtt{O}_k \mid \mathtt{I}_k$$

# Event structures

## Definition

A $\Sigma$-**event structure** is a tuple $(E, \leq_E, \#_E, \mathrm{lbl}_E : E \to \Sigma)$:

- $(E, \leq_E)$: a partial order representing *causality*
- $\#_E \subseteq E^2$: binary irreflexive relation representing *conflict*

+ axioms of *finite causes* and *conflict inheritance*.

$\rightsquigarrow$ $\rightarrowtriangle$ is derived from $\leq$ and $\sim$ from $\#$.

A **configuration** of $E$ is a subset $x \subseteq E$ which is:

- *downclosed* and *conflict-free*

$\mathscr{C}(E)$, the set of configurations of $E$ is a LTS:

$$x \xrightarrow{a} y \qquad \text{iff} \qquad y = x \uplus \{e\} \wedge \mathrm{lbl}(e) = a.$$

# Overview of the semantics

Goal: produce $[\![\langle p \odot \mu \rangle]\!]_{\mathtt{SC}}$ for each state such that:

$$\mathscr{C}([\![\langle p \odot \mu \rangle]\!]_{\mathtt{SC}}) \approx \langle p \odot \mu \rangle \quad \text{as } \Sigma_{SC}\text{-LTSs.}$$

4 steps:

(1) Semantics of individual threads
(2) Semantics of programs (without memory)
(3) Semantics of memory
(4) Combining the semantics.

# Semantics of individual threads and memory

**Individual threads.** Using sums and prefixes:

$$[\![x := k; t]\!]_{\mathtt{SC}} = \mathtt{W}_{x:=k} \cdot [\![t]\!]_{\mathtt{SC}}$$

$$\begin{tikzcd} \mathtt{W}_{x:=k} \arrow[d] \\ [\![t]\!]_{\mathtt{SC}} \end{tikzcd}$$

$$[\![r \leftarrow x; t]\!]_{\mathtt{SC}} = \sum_{n \in \mathbb{N}} \mathtt{R}_{x=n} \cdot [\![t(n)]\!]_{\mathtt{SC}}$$

$$\begin{tikzcd} \mathtt{R}_{x=0} \arrow[d] \arrow[r, squiggly, dash] \arrow[rr, squiggly, dash, bend left] & \mathtt{R}_{x=1} \arrow[d] \arrow[r, squiggly, dash] & \ldots \\ [\![t(0)]\!]_{\mathtt{SC}} & [\![t(1)]\!]_{\mathtt{SC}} & \ldots \end{tikzcd}$$

**Programs.** Threads are combined using parallel composition

$$[\![t_1 \parallel \ldots \parallel t_n]\!]_{\mathtt{SC}} = [\![t_1]\!]_{\mathtt{SC}} \parallel \ldots \parallel [\![t_n]\!]_{\mathtt{SC}} \qquad [\![t_1]\!]_{\mathtt{SC}} \quad \ldots \quad [\![t_n]\!]_{\mathtt{SC}}$$

# Semantics of the memory

Storage semantics in SC orders accesses *on the same variable*.

$$m_{x:=k} = \begin{tikzcd} \mathtt{R}_{x=k} \arrow[r, no head, squiggly] \arrow[rr, no head, squiggly, bend left] \arrow[d] & \mathtt{W}_{x:=0} \arrow[r, no head, squiggly] \arrow[d] & \mathtt{W}_{x:=1} \arrow[d] & \dots \\ m_{x:=k} & m_{x:=0} & m_{x:=1} & \dots \end{tikzcd}$$

$$[\![\mu]\!] = m_{x:=\mu(x)} \parallel m_{y:=\mu(y)} \parallel \dots$$

$[\![\mu]\!]$ is $\Sigma_m$-labelled ($\Sigma_m ::= \mathtt{R}_{x=k} \mid \mathtt{W}_{x:=k}$).

More concretely:

- **Events** of $[\![\mu]\!]$: consistent history *on one variable*.
- **Configurations** of $[\![\mu]\!]$: consistent *global* history.

$[\![\mu]\!]$ works for all multicopy atomics architectures.

# Combining them: interaction states

$[\![\langle p \odot \mu\rangle]\!]$ should combine the behaviours of $[\![p]\!]$ and $[\![\mu]\!]$:

$$\begin{tikzcd} \mathtt{W}_{data:=17} \arrow[d, "{[\![p]\!]}"'] & \mathtt{R}_{flag=1} \arrow[d, "{[\![p]\!]}"] \\ \mathtt{W}_{flag:=1} \arrow[ur, "{[\![\mu]\!]}"] & \mathtt{R}_{data=17} \end{tikzcd} \quad \in \mathscr{C}([\![\langle p \odot \mu\rangle]\!])$$

## Definition

A **synchronisation** is a tuple $X = (X.\mathtt{thr}, X.\mathtt{hist}, \varphi)$ with:

- $X.\mathtt{thr} \in \mathscr{C}([\![p]\!])$ and $X.\mathtt{hist} \in \mathscr{C}([\![\mu]\!])$.
- $\varphi$ is a label-preserving bijection $X.\mathtt{thr} \cap \Sigma_m \simeq X.\mathtt{hist}$.

# Combining them: interaction states

$[\![\langle p \odot \mu \rangle]\!]$ should combine the behaviours of $[\![p]\!]$ and $[\![\mu]\!]$:

$$\begin{tikzcd}
\mathtt{W}_{\mathit{data}:=17} \arrow[d, "{[\![p]\!]}"'] & \mathtt{R}_{\mathit{flag}=1} \arrow[d, "{[\![p]\!]}"] \\
\mathtt{W}_{\mathit{flag}:=1} \arrow[ur, "{[\![\mu]\!]}"] & \mathtt{R}_{\mathit{data}=17}
\end{tikzcd} \in \mathscr{C}([\![\langle p \odot \mu \rangle]\!])$$

## Definition

A **synchronisation** is a tuple $X = (X.\mathtt{thr}, X.\mathtt{hist}, \varphi)$ with:

- $X.\mathtt{thr} \in \mathscr{C}([\![p]\!])$ and $X.\mathtt{hist} \in \mathscr{C}([\![\mu]\!])$.
- $\varphi$ is a label-preserving bijection $X.\mathtt{thr} \cap \Sigma_m \simeq X.\mathtt{hist}$.

There are two partial orders on $X.\mathtt{thr}$:

$$s \leq_{\mathtt{thr(X)}} s' := s \leq_{[\![p]\!]} s' \qquad s \leq_{\mathtt{mem(X)}} s' := \varphi\, s \leq_{[\![\mu]\!]} \varphi\, s'.$$

$X$ is **acyclic** when $\leq_{\mathtt{thr}(X)} \cup \leq_{\mathtt{mem}(X)}$ is acyclic.

# The prime construction

Acyclic synchro. should be the configurations of $[\![\langle p \odot \mu \rangle]\!]$.
$\rightsquigarrow$ In any $E$, $|E| \simeq \{x \in \mathscr{C}(E) \mid x$ has a greatest element$\}$.

## Theorem (Prime construction, [Hay14])

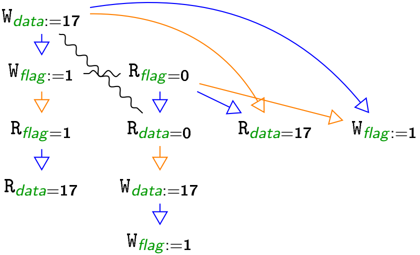
*For a collection of partial orders $\mathscr{Q}$ (closed under prefix), there exists an event structure $Pr(\mathscr{Q})$ such that $\mathscr{C}(Pr(\mathscr{Q})) \cong Q$.*
$\rightsquigarrow$ Its events are elements of $\mathscr{Q}$ with a greatest element.

We let $[\![p]\!] * [\![\mu]\!]$ to the be primes of acyclic configurations.

# Correctness

$[\![p]\!] * [\![\mu]\!]$ can be equipped with two orders $\leq_{\texttt{thr}}$ and $\leq_{\texttt{mem}}$.

$[\![\texttt{mp}]\!] =$




Letting $[\![\langle p \odot \mu \rangle]\!] = [\![p]\!] * [\![\mu]\!]$ we have: $[\![\langle p \odot \mu \rangle]\!] \approx \langle p \odot \mu \rangle$.
$\rightsquigarrow$ Proof of correctness component by component.

## II. A strong DRF result for TSO

if $p$ race-free on SC:

$$p \models_{SC} \varphi \Leftrightarrow p \models_{TSO} \varphi$$

# Total Store Ordering in one slide [OSS09]

TSO is a memory specification allowing for **store buffers**.

$$x = y = 0.$$
$$\begin{array}{r||l} x := 1 & y := 1 \\ r \leftarrow y & s \leftarrow x \end{array}$$
$$\text{Allowed } r = s = 0.$$

Usual LTS for TSO equips threads with a buffer in $(V \times \mathbb{N})^*$.

- New instruction, `fence`: flushes the current thread's buffer.
- New labels: $\Sigma_{\mathsf{TSO}} := \Sigma_{\mathsf{SC}} \mid \mathtt{fence} \mid \mathtt{BR}_{x:=k} \mid \mathtt{BW}_{x:=k}$.

Our variations:

- Atomic accesses require empty buffers (as fences do)
- Input/Outputs do *not* require empty buffers.

# Threads are not sequential anymore

For the thread $t = x := 1; r \leftarrow y$, a TSO processor may do:

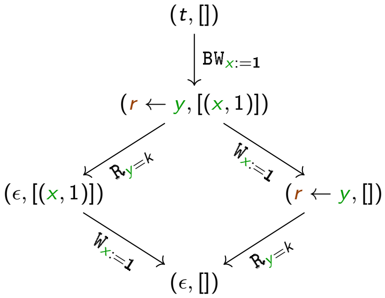
- Store the write, perform the read, commit the write.
- Commit directly the write and perform the read.

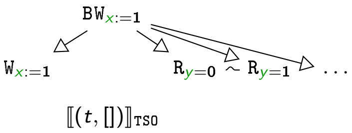
$$
\begin{tikzcd}
 & (t, []) \arrow[d, "\mathtt{BW}_{x:=1}"] & \\
 & (r \leftarrow y, [(x, 1)]) \arrow[dl, "\mathtt{R}_{y=k}"'] \arrow[dr, "\mathtt{W}_{x:=1}"] & \\
(\epsilon, [(x, 1)]) \arrow[dr, "\mathtt{W}_{x:=1}"'] & & (r \leftarrow y, []) \arrow[dl, "\mathtt{R}_{y=k}"] \\
 & (\epsilon, []) &
\end{tikzcd}
$$

# Threads are not sequential anymore

For the thread $t = x := 1; r \leftarrow y$, a TSO processor may do:

- Store the write, perform the read, commit the write.
- Commit directly the write and perform the read.




Events $\mathtt{W}_{x:=1}$ and $\mathtt{R}_{y=k}$ should be **concurrent** in $[\![(t, [])]\!]_{\mathtt{TSO}}$.

# Threads are not deterministic anymore

For the thread $t = x := 1; r \leftarrow x$, a TSO processor may do:

- commit the write, and satisfy the read from memory
- store the write, read from the buffer and only then commit.

Those transitions are **not concurrent**.

$$
\begin{tikzcd}
 & (t, []) \arrow[d, "\mathtt{BW}_{x:=1}"] & \\
 & (r \leftarrow x, [(x,1)]) \arrow[dl, "\mathtt{BR}_{x:=1}"'] \arrow[dr, "\mathtt{W}_{x:=1}"] & \\
(\epsilon, [(x,1)]) \arrow[d, "\mathtt{W}_{x:=1}"] & & (r \leftarrow x, []) \arrow[d, "\mathtt{R}_{x=k}"] \\
(\epsilon, []) & & (\epsilon, [])
\end{tikzcd}
$$

Events $\mathtt{W}_{x:=1}$ and $\mathtt{BR}_{x:=1}$ should be **in conflict** in $[\![(t, [])]\!]_{\mathtt{TSO}}$.

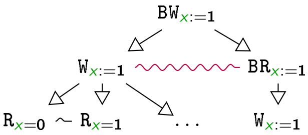
# Threads are not deterministic anymore

For the thread $t = x := 1; r \leftarrow x$, a TSO processor may do:

- commit the write, and satisfy the read from memory
- store the write, read from the buffer and only then commit.

Those transitions are **not concurrent**.

$$\begin{tikzcd}
 & (t, []) \arrow[d, "\mathtt{BW}_{x:=1}"] & \\
 & (r \leftarrow x, [(x, 1)]) \arrow[dl, "\mathtt{BR}_{x:=1}"'] \arrow[dr, "\mathtt{W}_{x:=1}"] & \\
(\epsilon, [(x, 1)]) \arrow[d, "\mathtt{W}_{x:=1}"] & & (r \leftarrow x, []) \arrow[d, "\mathtt{R}_{x=k}"] \\
(\epsilon, []) & & (\epsilon, [])
\end{tikzcd}$$




$[\![(t, [])]\!]_{\mathtt{TSO}}$

Events $\mathtt{W}_{x:=1}$ and $\mathtt{BR}_{x:=1}$ should be **in conflict** in $[\![(t, [])]\!]_{\mathtt{TSO}}$.

# Generalised prefix and TSO thread semantics

To represent thread concurrency, we relax the usual prefix:

$$\ell \cdot_R E = \begin{tikzcd} \ell \arrow[d, bend right] \arrow[d, bend left] \\ E \end{tikzcd} : \ell \leq e \text{ when } (\ell, \mathsf{lbl}(e')) \notin R \text{ for some } e' \leq e.$$

where $R \subseteq \Sigma \times \Sigma$ is the **concurrency relation**. For TSO:

$$R = \{(\mathtt{W}_{x:=k}, e) \mid e \text{ I/O, read on nonatomic } y \neq x\}$$

# Generalised prefix and TSO thread semantics

To represent thread concurrency, we relax the usual prefix:

$$\ell \cdot_R E = \begin{array}{c} \ell \\ \swarrow \quad \searrow \\ E \end{array} : \ell \leq e \text{ when } (\ell, \mathsf{lbl}(e')) \notin R \text{ for some } e' \leq e.$$

where $R \subseteq \Sigma \times \Sigma$ is the **concurrency relation**. For TSO:

$$R = \{(\mathtt{W}_{x:=k}, e) \mid e \text{ I/O, read on nonatomic } y \neq x\}$$

A few interesting rules:

$$[\![x := k; t, \mathfrak{b}]\!] = \mathtt{BW}_{x:=k} \cdot_R [\![t, \mathfrak{b}{+}{+}(x,k)]\!]$$

$$[\![\mathtt{fence}; t, \mathfrak{b}]\!] = \mathtt{W}_{x_1:=k_1} \cdot_R \ldots \cdot_R \mathtt{W}_{x_n:=k_n} \cdot_R \mathtt{fence} \cdot_R [\![t, \epsilon]\!]$$
$$\text{when } \mathfrak{b} = [(x_1, k_1), \ldots, (x_n, k_n)]$$

$$[\![r \leftarrow x; t, \mathfrak{b}]\!] = (\mathtt{BR}_{x:=k} \cdot_R [\![t[r := k], \mathfrak{b}]\!]) + (\mathtt{W}_{y:=m} \cdot_R [\![r \leftarrow x; t, \mathfrak{b}']\!])$$
$$\text{when } x \text{ occurs in } \mathfrak{b} \text{ with value } k \text{ and } \mathfrak{b} = (y, m){+}{+}\mathfrak{b}'.$$

# Results about the TSO semantics.

The semantics extends to machines the same way as for SC:

$$[\![\langle \mathfrak{t}_1 \parallel \ldots \parallel \mathfrak{t}_n \odot \mu\rangle]\!]_{\mathsf{TSO}} = ([\![\mathfrak{t}_1]\!]_{\mathsf{TSO}} \parallel \ldots \parallel [\![\mathfrak{t}_n]\!]_{\mathsf{TSO}}) * [\![\mu]\!]$$

where $\mathfrak{t}_i$ of the form $(t_i, \mathfrak{b}_i)$

## Theorem

*For any TSO machine state $\mathfrak{m}$, we have*

$$[\![\mathfrak{m}]\!]_{TSO} \approx \mathfrak{m}.$$

# Let us talk about races

Races are concurrent accesses on **nonatomic** variables.

## Definition

A program $p$ is **race-free** when for all $\langle p \odot \mu \rangle$ reducing to $\langle p' \odot \mu' \rangle$ (on SC), then $p'$ does not have two initial actions on the same nonatomic variable one of which being a write.

This only allows thread communication on atomic variables:

## Lemma

*Let $p$ be race-free and $e, e' \in [\![\langle p \odot \mu \rangle]\!]_{SC}$ such that:*

- *$e$ and $e'$ are not in conflict and not comparable for $\leq_{\mathtt{thr}}$,*
- *$e <_{\mathtt{mem}} e'$ with no events in between.*

*Then $e$ and $e'$ are actions on an atomic variable.*

# Data-Race-Free theorem

We can generalise the result of [Owe10]:

## Theorem

*Let $p$ be a race-free program. For any $\mu$:*

$$\mathscr{C}(\llbracket \langle p \odot \mu \rangle \rrbracket_{TSO}) \approx_{io} \mathscr{C}(\llbracket \langle p \odot \mu \rangle \rrbracket_{SC}),$$

$\approx_{io}$: *weak bisimulation where visible events are IO events.*

$\rightsquigarrow$ satisfaction of Hennessy-Milner formulas is transferred.

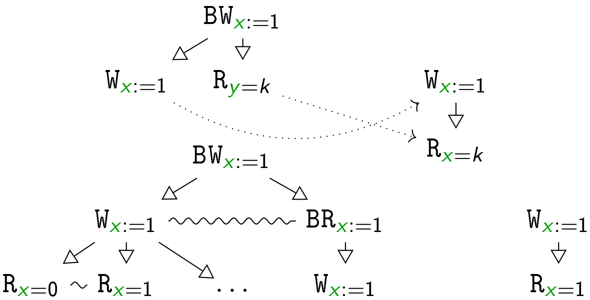
Among HML formulas, there are liveness properties, eg.

Program $p$ inputs a natural number, outputs its double and then stops.

(NB: Trace based equivalences would allow $p$ to stop after the input due to a deadlock.)

# Outline of the proof

We first build a partial function $\psi : [\![p]\!]_{\mathsf{TSO}} \rightharpoonup [\![p]\!]_{\mathsf{SC}}$:




This function induces $\bar{\psi} : \mathscr{C}([\![p]\!]_{\mathsf{TSO}}) \to \mathscr{C}([\![p]\!]_{\mathsf{SC}})$.

## Lemma

*If $p$ is race-free, $\bar{\psi}$ lifts to $\mathscr{C}([\![\langle p \odot \mu \rangle]\!]_{TSO}) \to \mathscr{C}([\![\langle p \odot \mu \rangle]\!]_{SC})$.*

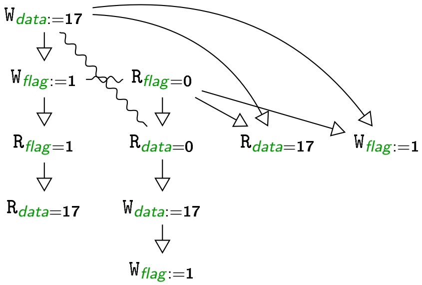
$\rightsquigarrow$ The bisimulation is built using this map.

# III. Relaxing coherence

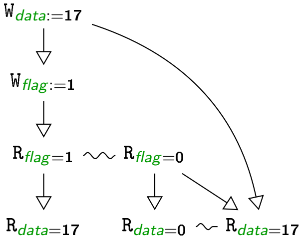
$$\begin{tikzcd} \mathtt{W}_{x:=1} \arrow[r, phantom, "\sim"] \arrow[d] & \mathtt{W}_{x:=2} \arrow[d] \\ \mathtt{W}_{x:=2} & \mathtt{W}_{x:=1} \end{tikzcd} \qquad \text{vs.} \qquad \mathtt{W}_{x:=1} \quad \mathtt{W}_{x:=2}$$

# Coherence is too strict

Our memory cell $[\![\mu]\!]$ orders every access to the same variable.
$\rightsquigarrow$ Introduces undesired redundancy, eg. in mp:




Semantics of (1)



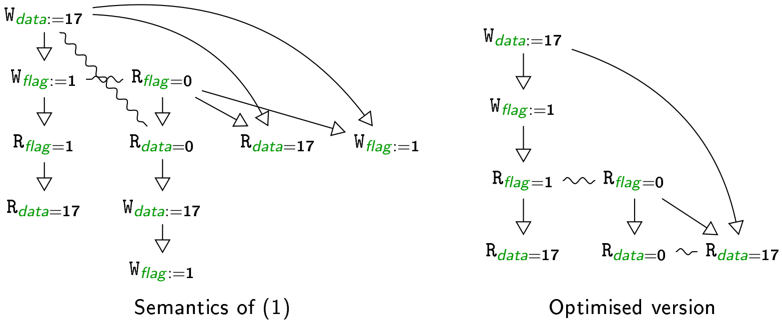

Optimised version

$\rightsquigarrow$ Same outcomes but fewer configurations on the right.

# Coherence is too strict

Our memory cell $[\![\mu]\!]$ orders every access to the same variable.

$\rightsquigarrow$ Introduces undesired redundancy, eg. in `mp`:

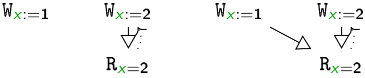



$\rightsquigarrow$ Same outcomes but fewer configurations on the right.

**Goal:** Given $E$, build $E_\mu$, a more compact version of $E * [\![\mu]\!]$?

# Our take on candidates

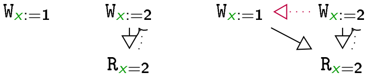
A **candidate** is a Σ-partial order where reads are justified:




$C_1$ $C_2$

# Our take on candidates

A **candidate** is a $\Sigma$-partial order where reads are justified:




$C_1$ valid　　　　$C_2$ invalid

$C$ is **valid** when all linearisations of writes are SC-executable.

## Definition

An execution of $x \in \mathscr{C}(E)$ is a valid candidate $C$ such that:

(1) $|x| = |C|$ and $s \leq_E s' \Rightarrow s \leq_C s'$ for $s, s' \in x$

(2) In $C$, I/O actions are all comparable.

(3) It is minimal: there are no $C'$ satisfying (1) and (2) with $\leq_C \subsetneq \leq_{C'}$.

# The event structure $E_\mu$

We can construct an event structure based on executions:

## Theorem

*There exists an event structure $E_\mu$ whose **maximal** configurations correspond to pairs $(x, C)$ of a maximal configuration of $E$ and $C$ an execution of $x$.*

**Non-incremental**: need the maximal configurations of $E$.

## Theorem

- $tr_{io}(E_\mu) = tr_{io}(E * [\![\mu]\!])$
- *$E_\mu$ simulates $E * [\![\mu]\!]$.*

$E * [\![\mu]\!]$ does not simulate $E_\mu$: choices are made later in $E_\mu$.

# Approximating the executions

How to compute the executions of $x \in \mathscr{C}(E)$ ?

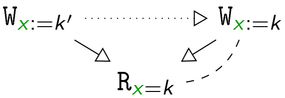
1. Compute the possible justifications for reads in $x$. $\rightsquigarrow$ A set of candidates $C$
2. For each $C$, add causal links to compute the possible executions augmenting $C$.

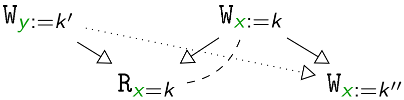
# Approximating the executions

How to compute the executions of $x \in \mathscr{C}(E)$ ?

1. Compute the possible justifications for reads in $x$. $\rightsquigarrow$ A set of candidates $C$
2. For each $C$, add causal links to compute the possible executions augmenting $C$.

A simple heuristic, add links in the following cases:




(Heuristic independently developed by Luc Maranget)
This heuristic can be implemented in **Herd**.

$\rightsquigarrow$ Ok for simple cases, but not for complicated programs...

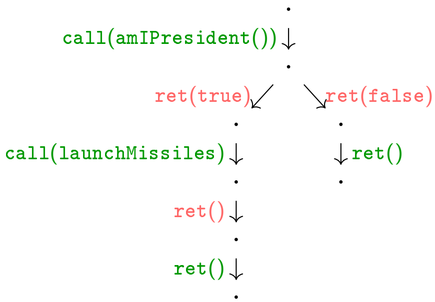
## IV. Beyond assembly: higher-order languages

# Functions and LTS

What about code calling foreign functions?

```
void redButton (void) {
  if (amIPresident())
    launchMissiles();
}
```

This can be described by a LTS using call/return events:

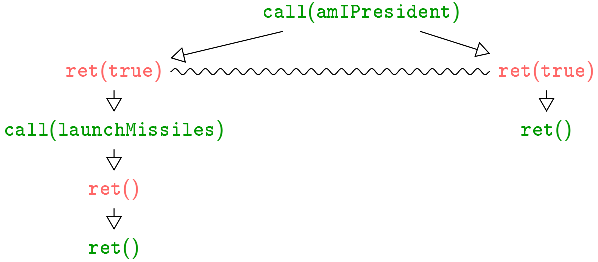


# Functions and LTS

What about code calling foreign functions?

```
void redButton (void) {
  if (amIPresident())
    launchMissiles();
}
```

This can be described by a LTS using call/return events:




…or as an event structure.

# Labels organise themselves as games

- Labels are now **polarised** *Context*/*Program*:

$$\mathtt{R}_{x=k} \rightsquigarrow \begin{array}{c} \mathtt{ReadReq}_x \\ \downarrow \\ \mathtt{ReadAns}_\mathtt{k} \end{array}$$

- Labels have **rules**: "Do not return before you are called."

⇝ Labels organise themselves in **games**: polarised forests.

$$\begin{tikzcd} & \mathtt{call(amIPresident)} \arrow[dl] \arrow[dr] & & \mathtt{call(launchMissiles)} \arrow[d] \\ \mathtt{ret(true)} & & \mathtt{ret(false)} & \mathtt{ret()} \end{tikzcd}$$

A rule-preserving trace of a game is called a **play**.

# Labels organise themselves as games

- Labels are now **polarised** *Context*/*Program*:

$$\mathtt{R}_{x=k} \rightsquigarrow \begin{array}{c}\mathtt{ReadReq}_x \\ \downarrow \\ \mathtt{ReadAns}_{\mathtt{k}}\end{array}$$

- Labels have **rules**: "Do not return before you are called."

⇝ Labels organise themselves in **games**: polarised forests.

$$\begin{array}{ccc} & \mathtt{call(amIPresident)} & & \mathtt{call(launchMissiles)} \\ \swarrow & & \searrow & \downarrow \\ \mathtt{ret(true)} & & \mathtt{ret(false)} & \mathtt{ret()} \end{array}$$

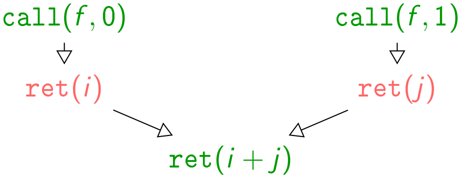
A rule-preserving trace of a game is called a **play**.

⇝ **Game semantics** pioneered the study of programs as sets of plays on games (strategies) [HO00, AJM00].

# Parallel functions as event structures

[RW11] used event structures to represent strategies:

$$\left[\!\left[\begin{array}{l}\texttt{int sum(void) \{}\\ \quad\texttt{return f(0) + f(1);}\\ \texttt{\}}\end{array}\right]\!\right] =$$




$\rightsquigarrow$ Opens the possibility to model open higher-order concurrent programs with event structures.

However, major restriction, **linearity**: in each configuration, each move must be played once!

# Nonlinearity

What if Player wants to be nonlinear?
$\rightsquigarrow$ To call a function twice (as in the previous slide)

Following [AJM00], we add copy indices to moves:

game $A \rightsquigarrow$ game $!A$ where moves are duplicated $\omega$ times.

The previous example becomes:

$$\left[\!\!\left[\begin{array}{l}\mathtt{int\ sum(void)\ \{} \\ \quad \mathtt{return\ f(0)\ +\ f(1);} \\ \mathtt{\}}\end{array}\right]\!\!\right] = \begin{tikzcd} \mathtt{call}(f,0)_0 \arrow[d] & & \mathtt{call}(f,1)_1 \arrow[d] \\ \mathtt{ret}(i)_p \arrow[dr] & & \mathtt{ret}(j)_q \arrow[dl] \\ & \mathtt{ret}(i+j)_{\langle p,q\rangle} & \end{tikzcd}$$

# A model of IPA

These considerations lead to:

## Theorem (C., Clairambault, Winskel)

*These expanded games and strategies form a model of higher-order concurrent and nondeterministic computation.*

Model highlights the complicated causal patterns of such programs:

$$\left[\!\!\left[\begin{array}{l}\texttt{int shy(void)\{}\\ \quad\texttt{static int timesCalled = 0;}\\ \quad\texttt{timesCalled ++;}\\ \quad\texttt{if (timesCalled == 2) return 0;}\\ \quad\texttt{else while(true);}\\ \texttt{\}}\end{array}\right]\!\!\right] = \quad q_0 \quad \cdots$$

# A model of IPA

These considerations lead to:

## Theorem (C., Clairambault, Winskel)

*These expanded games and strategies form a model of higher-order concurrent and nondeterministic computation.*

Model highlights the complicated causal patterns of such programs:

```
⟦int shy(void){
   static int timesCalled = 0;
   timesCalled ++;
   if (timesCalled == 2) return 0;
   else while(true);
 }⟧
```

$$= \begin{tikzcd} q_0 \arrow[d] \arrow[dr] & q_1 \arrow[d] \arrow[dl] & \cdots \\ 0 \arrow[r, phantom, "\sim\sim"] & 0 & \end{tikzcd}$$

# A model of IPA

These considerations lead to:

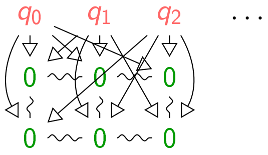
## Theorem (C., Clairambault, Winskel)

*These expanded games and strategies form a model of higher-order concurrent and nondeterministic computation.*

Model highlights the complicated causal patterns of such programs:

$$\left[\!\left[ \text{ (code below) } \right]\!\right] =$$

```
int shy(void){
  static int timesCalled = 0;
  timesCalled ++;
  if (timesCalled == 2) return 0;
  else while(true);
}
```



# Related work

**Weak memory and event structures.**

- Brookes & Kavanagh's model of TSO with pomsets.
- Pichon & Sewell's operational semantics on event structures
- Jeffrey & Riely's axiomatic model using event structures

**Game Semantics for concurrency.**

- Laird, and Ghica & Murawski's models using interleaving.
- Tsukada & Sakayori's model of concurrency using set of pomsets.
- Hirschowitz's model using presheaves over spans.

# A rich semantic universe based on event structures

**Extensions.** Model is extensible and has been extended to:

- continuous probabilities (Paquet, Winskel)
- quantum computation (Clairambault, de Visme, Winskel)

**Ongoing work.** In many different contexts:

- **probabilistic programming**
- **dependences of logical rules**
- **message-passing concurrency**

**Research agenda.**

- Investigate more applied models (ARM, C11), ...
- How to have a finite representation of these two issues:
  - recursion (depth)
  - unbounded contexts (breadth)
- Implement such models in a flexible way (à la Herd), ...

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria.
Full abstraction for PCF.
*Information and Computation*, 163(2):409–470, 2000.

Jonathan Hayman.
Interaction and causality in digital signature exchange protocols.
In Matteo Maffei and Emilio Tuosto, editors, *Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers*, volume 8902 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2014.

Martin Hyland and Luke Ong.
On full abstraction for PCF.
*Information and Computation*, 163:285–408, 2000.

Scott Owens, Susmit Sarkar, and Peter Sewell.

A better x86 memory model: x86-tso.
In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

Scott Owens.
Reasoning about the implementation of concurrency abstractions on x86-tso.
In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 478–503, 2010.

Silvain Rideau and Glynn Winskel.
Concurrent strategies.
In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 409–418, 2011.