Reconciling nondeterminism and causality
Event structures for weak memory

Simon Castellan\textsuperscript{1}
(Joint work with Jade Alglave and Jean-Marie Madiot.)

\textsuperscript{1}Imperial College London, UK

27 November 2018
Reasoning on concurrent programs

Consider the program mp:

\[
\begin{align*}
data &= flag = 0 \\
data &:= 17; \quad r \leftarrow flag; \\
flag &:= 1 \quad v \leftarrow data
\end{align*}
\]

Does mp $\vDash r = 1 \Rightarrow v = 17$?
Reasoning on concurrent programs

Consider the program \( mp \):

\[
\begin{align*}
data &= flag = 0 \\
data &:= 17; \quad r \leftarrow flag; \\
flag &:= 1 \quad \quad v \leftarrow data
\end{align*}
\]

Does \( mp \models r = 1 \Rightarrow v = 17 \)?

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:

\[ \langle (x := 1; t \parallel p) \circ \mu \rangle \xrightarrow{W_{x:=1}} \langle (t \parallel p) \circ \mu[x := 1] \rangle. \]

Transitions labelled by an action in \( \Sigma ::= W_{x:=k} \mid R_{x=k} \mid \ldots \).

Executions of the program become traces of the LTS:

\[ \langle mp \circ \mu \rangle \xrightarrow{W_{data:=17}} \xrightarrow{W_{flag:=1}} \xrightarrow{R_{flag=1}} \xrightarrow{R_{value=17}}. \]
Operational semantics: machines as LTSs

Formalises an abstract machine running the program:

$$\langle (x := 1; t \parallel p) \circ \mu \rangle \xrightarrow{W_{x:=1}} \langle (t \parallel p) \circ \mu[x := 1] \rangle.$$ 

Transitions labelled by an action in \( \Sigma := W_{x:=k} \mid R_{x=k} \mid \ldots \).

Executions of the program become traces of the LTS:

$$\langle mp \circ \mu \rangle \xrightarrow{W_{data:=17}} \xrightarrow{W_{flag:=1}} \xrightarrow{R_{flag}=1} \xrightarrow{R_{value}=17}.$$ 

- Represents nondeterministic branching points.
  - Liveness properties, whole program optimisations.

- Combinatorial explosion due to interleaving.
  - Hard to simulate, hard to reason on.
Axiomatic semantics

Formalises a program by the set of its valid executions:

\[
\text{program syntax} \rightsquigarrow \text{execution candidates} \rightsquigarrow \text{executions}
\]

set of events + relations

Two candidates for mp:

\[
\begin{align*}
W_{\text{data}} &= 17 & R_{\text{flag}} &= 0 \\
\text{po} \downarrow & & \text{po} \\
W_{\text{flag}} &= 1 & R_{\text{data}} &= 0
\end{align*}
\]

valid on all architectures

\[
\begin{align*}
W_{\text{data}} &= 17 & R_{\text{flag}} &= 1 \\
\text{po} \downarrow & & \text{po} \\
W_{\text{flag}} &= 1 & R_{\text{data}} &= 0
\end{align*}
\]

valid on some (e.g. ARM)
Axiomatic semantics

Formalises a program by the set of its valid executions:

\[
\text{program syntax} \rightsquigarrow \text{execution candidates} \rightsquigarrow \text{executions}
\]

set of events+relations

Two candidates for \( mp \):

\[
\begin{align*}
W_{data} &= 17 & R_{flag} &= 0 \\
W_{flag} &= 1 & R_{data} &= 0
\end{align*}
\]

valid on all architectures

\[
\begin{align*}
W_{data} &= 17 & R_{flag} &= 1 \\
W_{flag} &= 1 & R_{data} &= 0
\end{align*}
\]

valid on some (eg. ARM)

\(\oplus\) **Causal** account of executions.

\(\rightsquigarrow\) Easy to simulate; allows higher-level reasoning.

\(\ominus\) **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 \( \rightarrow \) on each execution

Event structures
global notion of events (\( \approx \) transitions)
events equipped with \( \sim \) and \( \rightarrow \)

\[ \begin{align*}
\text{mp:} & \quad W_{\text{data}} := 17 \\
& \quad W_{\text{flag}} := 1 \\
& \quad R_{\text{flag}} := 1 \sim \sim R_{\text{flag}} := 0 \\
& \quad R_{\text{data}} := 17 \\
& \quad R_{\text{data}} := 0 \sim \sim R_{\text{data}} := 17
\end{align*} \]
The best of both worlds: event structures

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

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

Event structures
global notion of events ($\approx$ transitions)
events equipped with $\sim$ and $\rightarrow$

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

Reconciling nondeterminism and causality · Simon Castellan
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

\[ [mp] = \]

\[ W_{\text{data}} = 17 \]

\[ W_{\text{flag}} = 1 \]

\[ R_{\text{flag}} = 0 \]

\[ R_{\text{data}} = 0 \]

\[ R_{\text{data}} = 17 \]

\[ W_{\text{flag}} = 1 \]

\[ R_{\text{data}} = 17 \]

\[ W_{\text{data}} = 17 \]

\[ W_{\text{flag}} = 1 \]
Our language

We consider a simple imperative language:

\[
e ::= r \mid e + e \mid \ldots \quad \text{expressions}
\]

\[
t ::= \epsilon \mid x := e; t \mid r \leftarrow x; t
\quad \mid \text{output } e \mid r \leftarrow \text{input}
\quad \mid \text{if } (0 == e) \{ t \} \{ t \}
\]

\[
p ::= t \parallel \ldots \parallel t \quad \text{programs}
\]

Features *global variables* and *thread registers*

Input / Output instructions used as “observation points”

Traditional LTS on states \( \langle p \odot \mu : V \rightarrow \mathbb{N} \rangle \) labeled over:

\[
\Sigma_{SC} ::= R_{x=k} \mid W_{x:=k} \mid O_k \mid I_k
\]
Event structures

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

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

+ axioms of finite causes and conflict inheritance.

$\Rightarrow \Rightarrow$ is derived from $\leq$ and $\sim\sim$ from $\#$.

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

- downclosed and conflict-free

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

$$x \xrightarrow{a} y \iff y = x \cup \{e\} \land \text{lbl}(e) = a.$$

Reconciling nondeterminism and causality · Simon Castellan
Overview of the semantics

Goal: produce \( [[\langle p \circ \mu \rangle]]_{\text{sc}} \) for each state such that:

\[
\mathcal{C}( [[\langle p \circ \mu \rangle]]_{\text{sc}} ) \approx \langle p \circ \mu \rangle \quad \text{as } \Sigma_{\text{sc}}\text{-LTSs}.
\]

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:

\[
[x := k; t]_{sc} = \sum_{n \in \mathbb{N}} R_{x=n} \cdot [t(n)]_{sc}
\]

\[
W_{x:=k} \\
\downarrow \\
[t]_{sc}
\]

\[
R_{x=0} \sim \sim \sim R_{x=1} \sim \ldots \\
\downarrow \\
[t(0)]_{sc} \quad [t(1)]_{sc} \quad \ldots
\]

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

\[
[t_1 \parallel \ldots \parallel t_n]_{sc} = [t_1]_{sc} \parallel \ldots \parallel [t_n]_{sc} \\
[t_1]_{sc} \quad \ldots \quad [t_n]_{sc}
\]
Semantics of the memory

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

\[ m_x := k = \begin{array}{c}
R_x = k \\
\sim \ \\
W_x = 0 \\
\sim \\
W_x = 1 \\
\sim \\
\cdots \\
\downarrow \\
\downarrow \\
\downarrow \\
\downarrow \\
m_x := k \\
m_x := 0 \\
m_x := 1 \\
\cdots \\
\end{array} \]

\[ [[\mu]] = m_x := \mu(x) \parallel m_y := \mu(y) \parallel \cdots \]

\[ [[\mu]] \text{ is } \Sigma_m\text{-labelled } (\Sigma_m ::= R_x = k \mid W_x = k). \]

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

\[ \llangle p \odot \mu \rrangle \] should combine the behaviours of \([p]\) and \([\mu]\):

\[
\begin{align*}
W_{\text{data}} &= 17 \\
[p] &\Downarrow \\
W_{\text{flag}} &= 1 \\
[\mu] &\Downarrow \\
R_{\text{flag}} &= 1 \\
[p] \Downarrow \\
R_{\text{data}} &= 17
\end{align*}
\]

\[ \in \mathcal{C}(\llangle p \odot \mu \rrangle) \]

Definition

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

1. \(X.\text{thr} \in \mathcal{C}([p])\) and \(X.\text{hist} \in \mathcal{C}([\mu])\).
2. \(\varphi\) is a label-preserving bijection \(X.\text{thr} \cap \Sigma_m \simeq X.\text{hist}\).
Combining them: interaction states

\[ [[p \odot \mu]]\] should combine the behaviours of \([p]\) and \([\mu]\):

\[
\begin{align*}
W_{\text{data}} &= 17 \\
R_{\text{flag}} &= 1 \\
[p] &\Downarrow \\
\downarrow & [\mu] \\
\downarrow [p] &\in \mathcal{C}([[p \odot \mu]]) \\
W_{\text{flag}} &= 1 \\
R_{\text{data}} &= 17
\end{align*}
\]

Definition

A \textit{synchronisation} is a tuple \(X = (X.\text{thr}, X.\text{hist}, \varphi)\) with:

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

There are two partial orders on \(X.\text{thr}:

\[
s \preceq_{\text{thr}(X)} s' := s \preceq_{[p]} s' \quad s \preceq_{\text{mem}(X)} s' := \varphi s \preceq_{[\mu]} \varphi s'.
\]

\(X\) is \textit{acyclic} when \(\preceq_{\text{thr}(X)} \cup \preceq_{\text{mem}(X)}\) is acyclic.
The prime construction

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

Theorem (Prime construction, [Hay14])

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

We let $\llbracket p \rrbracket \ast \llbracket \mu \rrbracket$ to the be primes of acyclic configurations.
Correctness

\([p] * [\mu] \) can be equipped with two orders \( \leq_{\text{thr}} \) and \( \leq_{\text{mem}} \).

Letting \( [\langle p \diamond \mu \rangle] = [p] * [\mu] \) we have: \( [\langle p \diamond \mu \rangle] \approx \langle p \diamond \mu \rangle \).

\( \rightsquigarrow \) Proof of correctness component by component.
II. A strong DRF result for TSO

if $p$ race-free on SC:

$$p \models_{SC} \varphi \iff p \models_{TSO} \varphi$$
Total Store Ordering in one slide \cite{oss09}

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

\[
\begin{align*}
  x &= y = 0. \\
  x := 1 & \parallel y := 1 \\
  r \leftarrow y & \parallel s \leftarrow x \\
  \text{Allowed } r = s = 0.
\end{align*}
\]

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**: \(\Sigma_{TSO} := \Sigma_{SC} \mid \text{fence} \mid \text{BR}_{x:=k} \mid \text{BW}_{x:=k}\).

**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.

\[
\begin{align*}
(t, []) & \\
\downarrow \text{BW}_{x:=1} & \\
(r \leftarrow y, [(x, 1)]) & \\
\end{align*}
\]

\[
\begin{align*}
(\epsilon, [(x, 1)]) & \\
\downarrow \text{R}_{y=k} & \\
\text{BW}_{x:=1} & \\
\end{align*}
\]

\[
\begin{align*}
(r \leftarrow y, []) & \\
\downarrow \text{R}_{y=k} & \\
\text{BW}_{x:=1} & \\
(\epsilon, []) & \\
\end{align*}
\]
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 $W_{x:=1}$ and $R_{y=k}$ should be **concurrent** in $\llbracket (t, []) \rrbracket_{\text{TSO}}$. 
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.

Events \( W_x = 1 \) and \( BR_x = 1 \) should be in conflict in \( \llbracket (t, \emptyset) \rrbracket_{TSO} \).
Threads are not deterministic anymore

For the thread $t = x := 1; r ← x$, 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.

Events $W_x := 1$ and $BR_x := 1$ should be in conflict in $\llbracket (t, []) \rrbracket_{TSO}$. 

Reconciling nondeterminism and causality · Simon Castellan
Generalised prefix and TSO thread semantics

To represent thread concurrency, we relax the usual prefix:

\[ \ell \cdot_R E = \left\{ \ell \right\} : \ell \leq e \text{ when } (\ell, \text{lbl}(e')) \notin R \text{ for some } e' \leq e. \]

where \( R \subseteq \Sigma \times \Sigma \) is the concurrency relation. For TSO:

\[ R = \{ (\text{w}_{x:=k}, e) \mid e \text{ I/O, read on nonatomic } y \neq x \} \]
Generalised prefix and TSO thread semantics

To represent thread concurrency, we relax the usual prefix:

\[
\ell \cdot_R E = \begin{cases} \ell, & \ell \leq e \text{ when } (\ell, \text{lbl}(e')) \not\in R \text{ for some } e' \leq e. \\
E & \text{otherwise}
\end{cases}
\]

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:

\[
\begin{align*}
[x := k; t, b] &= \mathtt{bw}_{x := k} \cdot_R [t, b++(x, k)] \\
[fence; t, b] &= \mathtt{w}_{x_1 := k_1} \cdot_R \cdots \cdot_R \mathtt{w}_{x_n := k_n} \cdot_R \mathtt{fence} \cdot_R [t, \epsilon] \\
&\text{when } b = [(x_1, k_1), \ldots, (x_n, k_n)] \\
[r \leftarrow x; t, b] &= (\mathtt{br}_{x := k} \cdot_R [t[r := k], b]) + (\mathtt{w}_{y := m} \cdot_R [r \leftarrow x; t, b']) \\
&\text{when } x \text{ occurs in } b \text{ with value } k \text{ and } b = (y, m)++b'.
\end{align*}
\]
Results about the TSO semantics.

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

\[ \langle t_1 \parallel \ldots \parallel t_n \odot \mu \rangle_{\text{TSO}} = (\langle t_1 \rangle_{\text{TSO}} \parallel \ldots \parallel \langle t_n \rangle_{\text{TSO}}) \ast \langle \mu \rangle \]

where \( t_i \) of the form \((t_i, b_i)\)

**Theorem**

*For any TSO machine state \( m \), we have*

\[ [m]_{\text{TSO}} \approx m. \]
Let us talk about races

Races are concurrent accesses on nonatomic variables.

**Definition**
A program $p$ is **race-free** when for all $\langle p \circ \mu \rangle$ reducing to $\langle p' \circ \mu' \rangle$ (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 $e, e' \in \llbracket \langle p \circ \mu \rangle \rrbracket_{SC}$ such that:

- $e$ and $e'$ are not in conflict and not comparable for $\leq_{\text{thr}}$,  
- $e <_{\text{mem}} e'$ 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 $\mu$:

$$C([\langle p \odot \mu \rangle]_{TSO}) \approx_{io} C([\langle p \odot \mu \rangle]_{SC}),$$

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

$\Rightarrow$ 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 $\psi : \llbracket p \rrbracket_{\text{TSO}} \rightarrow \llbracket p \rrbracket_{\text{SC}}$:

This function induces $\overline{\psi} : \mathcal{C}(\llbracket p \rrbracket_{\text{TSO}}) \rightarrow \mathcal{C}(\llbracket p \rrbracket_{\text{SC}})$.

Lemma

If $p$ is race-free, $\overline{\psi}$ lifts to $\mathcal{C}(\llbracket \langle p \odot \mu \rangle \rrbracket_{\text{TSO}}) \rightarrow \mathcal{C}(\llbracket \langle p \odot \mu \rangle \rrbracket_{\text{SC}})$.

$\Rightarrow$ The bisimulation is built using this map.
III. Relaxing coherence

\[ W_x := 1 \sim W_x := 2 \]
\[ \Downarrow \quad \Downarrow \]
\[ W_x := 2 \quad W_x := 1 \]

vs.

\[ W_x := 1 \quad W_x := 2 \]
Coherence is too strict

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

⇝ Introduces undesired redundancy, eg. in mp:

Semantics of (1)

Optimised version

⇝ Same outcomes but fewer configurations on the right.
Coherence is too strict

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

$\leadsto$ Introduces undesired redundancy, eg. in mp:

Semantics of (1)

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

Goal: Given $E$, build $E_\mu$, a more compact version of $E \ast [\mu]$?
Our take on candidates

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

\[
W_x:=1 \quad W_x:=2 \\
\downarrow
\quad \uparrow
\quad \downarrow
\quad \downarrow
R_x:=2
\]

\[
C_1 \quad C_2
\]

Validity

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

Definition

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

1. $|x| = |C|$ and $s \leq \mathcal{E} s' \Rightarrow s \leq C s'$ for $s, s' \in x(\mathcal{E})$
2. In $C$, I/O actions are all comparable.
3. It is minimal: there are no $C'$ satisfying (1) and (2) with $\leq C \subset \leq C'$.
Our take on candidates

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

$W_x := 1$  $W_x := 2$

$R_x := 2$

$C_1$ valid

$C_2$ invalid

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

Definition

An execution of $x \in C(E)$ 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 $\leq C \subsetneq C'$. 
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**

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

$E \ast [\mu]$ does not simulate $E_\mu$: choices are made later in $E_\mu$. 
Approximating the executions

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

1. Compute the possible justifications for reads in $x$. $\leadsto$ 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:

$W_x := k$

$W_y := k'$

$W_x := k''$

(Heuristic independently developed by Luc Maramet)

This heuristic can be implemented in Herd.

$\leadsto$ Ok for simple cases, but not for complicated programs...
Approximating the executions

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

1. Compute the possible justifications for reads in $x$. $\leadsto$ 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.

$\leadsto$ Ok for simple cases, but not for complicated programs...
IV. Beyond assembly: higher-order languages
Functions and LTS

What about code calling foreign functions?

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

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

```
call(amIPresident())
    ret(true)
    ret(false)
call(launchMissiles)
    ret()
    ret()
    ret()
```

Reconciling nondeterminism and causality · Simon Castellan
Functions and LTS

What about code calling foreign functions?

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

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

```
call(amIPresident)
```

```
ret(true) ┌──┬──┐
    │   │   └──┐
    ▼   ▼      │
call(launchMissiles)
```

```
ret(true)
```

```
ret()
```

```
...or as an event structure.
```
Labels organise themselves as games

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

\[
R_x = k \overset{\rightsquigarrow}{\Rightarrow} \text{ReadReq}_x \downarrow \text{ReadAns}_k
\]

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

\[\rightsquigarrow\] Labels organise themselves in **games**: polarised forests.

- A rule-preserving trace of a game is called a **play**.
Labels organise themselves as games

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

\[
R_x = k \rightsquigarrow \text{ReadReq}_x \downarrow \text{ReadAns}_k
\]

▶ Labels have **rules**: “Do not return before you are called.”

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

\[
\text{call}(\text{amIPresident}) \quad \text{call}(\text{launchMissiles})
\]

\[
\text{ret}(\text{true}) \quad \text{ret}(\text{false}) \quad \text{ret}()
\]

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:

```c
int sum(void) {
    return f(0) + f(1);
}
```

⇝ 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?
⇝ To call a function twice (as in the previous slide)

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

game A ⇝ game !A where moves are duplicated \( \omega \) times.

The previous example becomes:

```
int sum(void) {
    return f(0) + f(1);
}
```

\[
\begin{align*}
\text{call}(f,0)_0 & \downarrow \text{ret}(i)_p \\
\text{call}(f,1)_1 & \downarrow \text{ret}(j)_q \\
\text{ret}(i + j)_{\langle p, q \rangle}
\end{align*}
\]
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:

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

\[ q_0 \quad \cdots \]
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);
}
```

\[
\begin{array}{ccc}
q_0 & q_1 & \cdots \\
\downarrow & \downarrow & \downarrow \\
0 & \sim & 0
\end{array}
\]
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:

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

\[
\begin{array}{c}
q_0 \quad q_1 \quad q_2 \quad \ldots \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
0 \sim 0 \sim 0 \\
\end{array}
\]

\[
\begin{array}{c}
0 \sim 0 \sim 0 \\
\end{array}
\]
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.

Jonathan Hayman.
Interaction and causality in digital signature exchange protocols.

On full abstraction for PCF.

Scott Owens, Susmit Sarkar, and Peter Sewell.
A better x86 memory model: x86-tso.

Scott Owens.
Reasoning about the implementation of concurrency abstractions on x86-tso.

Silvain Rideau and Glynn Winskel.
Concurrent strategies.