Weak memory models using event structures

Simon Castellan\textsuperscript{1}

\textsuperscript{1}LIP, ENS Lyon

March 25, 2016
Gallium Seminar
Unexpected behaviours

A simple concurrent and imperative program:

\[
\begin{align*}
    x, y & \text{ initialized to 0} \\
    x & := 1 \qquad y := 2 \\
    r & \leftarrow y \qquad s \leftarrow x
\end{align*}
\]

shared variable · local register

Expected outcome: \( r \neq 0 \lor s \neq 0 \).
Unexpected behaviours

A simple concurrent and imperative program:

\[
\begin{align*}
  &\text{x, y initialized to 0} \\
  &r \leftarrow y \quad s \leftarrow x \\
  &x := 1 \quad y := 2
\end{align*}
\]

shared variable · local register

Expected outcome: \( r \neq 0 \lor s \neq 0 \).
Wrong on modern architectures (x86, ARM, \ldots).
Unexpected behaviours

Another simple program:

\[
\begin{align*}
  x & := 1 \\
  r_1 & \leftarrow x \\
  r_2 & \leftarrow y \\
  y & := 1 \\
  s_1 & \leftarrow y \\
  s_2 & \leftarrow x
\end{align*}
\]

Expected outcome: \( r_1 = s_1 = 1 \Rightarrow r_2 = s_2 = 1 \)
Unexpected behaviours

Another simple program:

\[
\begin{align*}
    x & := 1 \\
    y & := 1 \\
    r_1 & \leftarrow x \\
    s_1 & \leftarrow y \\
    r_2 & \leftarrow y \\
    s_2 & \leftarrow x
\end{align*}
\]

Expected outcome: \( r_1 = s_1 = 1 \Rightarrow r_2 = s_2 = 1 \)

Wrong even without read exchange (Read Own Write Early).
A need to specify the behaviour

What are the expected behaviour of a concurrent programs?
→ It depends on the architectures.

Architectures need to be specified:
  ▶ what instructions can be reordered?
  ▶ how are writes propagated from one thread to the other?
A need to specify the behaviour

What are the expected behaviour of a concurrent programs?
→ It depends on the architectures.

Architectures need to be specified:
▶ what instructions can be reordered?
▶ how are writes propagated from one thread to the other?

To that end, manufacturers provide prosaic documents, but:
▶ ambiguity: behaviours that are not specified
▶ inconsistent: some observations may not be predicted.

Some architectures:
▶ SC (Sequential consistency): no reordering, sequential memory,
▶ ARM: reordering of instructions targeting different variables, write caches.
▶ x86: ...
Semantics saves the day

**Semantics**: Formalize mathematically the vendors specifications:
- get a (possibly computer-verified) proof of non-ambiguity,
- implement the specifications and mechanically **test it** against real life architectures.

Two main types of semantics among existing models:
- **operational semantics**: executions are described by the runs of an abstract machines,
- **axiomatic semantics**: the notion of valid execution is axiomatized.

Those models are called **weak memory models**.
Semantics and executions

The semantics generates from a program its possible executions:

<table>
<thead>
<tr>
<th>Program</th>
<th>Some executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x := 1)</td>
<td>(y := 2)</td>
</tr>
<tr>
<td>(r \leftarrow y)</td>
<td>(s \leftarrow x)</td>
</tr>
<tr>
<td></td>
<td>(W_x^{(1)} \cdot W_y^{(2)} \cdot R_y^{(2)} \cdot R_x^{(1)})</td>
</tr>
<tr>
<td></td>
<td>(W_y^{(2)} \cdot R_x^{(0)} \cdot W_x^{(2)} \cdot R_y^{(1)})</td>
</tr>
</tbody>
</table>

*Executions* can be formalized in different ways: traces, partial-order, ...
This talk

A semantics that is

- **denotational**: executions computed by induction
  - the semantics is thus *compositional*

- **compact**: based on event structures
  - no combinatorial explosion

- **extensible**: inspired from game semantics
  - it is easy to add loops, control operators, higher-order, ...

Outline of the talk:

1. **A semantics warm-up**: compute the SC semantics using *traces*.
2. Getting back the **causality**.
3. Our contribution: A **parametric** semantics using event structures.
4. A game semantics aparté at the end (if time allows)
I. A denotational semantics for SC

With traces of originality
Syntax precedes semantics

Our very simple programming language:

\[ e, e' ::= \{ \text{Expressions} \} \]

\[ k \in \mathbb{N} \mid r \in \mathcal{R} \mid e + e' \]

\[ \nu ::= \{ \text{Instructions} \} \]

\[ | a := e \quad \text{(Write on a variable)} \]

\[ | r \leftarrow a \quad \text{(Read on a variable)} \]

\[ t ::= \{ \text{Threads} \} \]

\[ | \nu; \ldots ; \nu \]

\[ p ::= \{ \text{Programs} \} \]

\[ t_1 \parallel \ldots \parallel t_n \]

In real life: conditionals and barriers.
Denotational semantics

**Goal:** compute $[t] \in E$ where $E$ is some space of denotations.

Our space here: languages of traces.

\[
\Sigma_a = \mathcal{V} \times \{R, W\} \quad \text{(Abstract memory event)}
\]

\[
\Sigma_c = \Sigma_a \times \mathbb{N} \quad \text{(Concrete memory event)}
\]

\[
E = \mathcal{P}(\Sigma_c^*)
\]

Notations: $R_x^{(k)}, W_x^{(k)}$.

Two steps:

1. **Volatile semantics** $[t]^O$: shared variables are considered *volatile*: $[x := 1; r \leftarrow x]^O$ does not guarantee to read 1 in $r$.

2. **Closed semantics**: once $[t]^O$ is calculated for the whole program, we restrict the scope of the variable $[x := 1; r \leftarrow x]$ reads 1 in $r$. 
Volatile semantics

Semantics of threads. Parametrized over $\rho : \mathcal{R} \rightarrow \mathbb{N}$.

(Writes) $[x := e; t] \rho = W_x(\rho(e)) \cdot [t] \rho$

(Reads) $[r \leftarrow x; t] \rho = \bigcup_{i \in \mathbb{N}} \left( R_x^{(i)} \cdot [t](\rho[r \leftarrow i]) \right)$
Volatile semantics

Semantics of threads. Parametrized over $\rho : \mathcal{R} \rightarrow \mathbb{N}$.

(Writes) \[ [x := e; t] \rho = W^{(\rho(e))}_x \cdot [t] \rho \]

(Reads) \[ [r \leftarrow x; t] \rho = \bigcup_{i \in \mathbb{N}} \left( R^{(i)}_x \cdot [t](\rho[r \leftarrow i]) \right) \]

Semantics of programs. Obtained by interleaving ($\otimes$):

\[ [t_1 \parallel \ldots \parallel t_n] = [t_1] \emptyset \otimes \ldots \otimes [t_n] \emptyset \]
Volatile semantics

Semantics of threads. Parametrized over $\rho : \mathcal{R} \to \mathbb{N}$.

(Writes) $[x := e; t] \rho = W_{x}^{(\rho(e))} \cdot [t] \rho$

(Reads) $[r \leftarrow x; t] \rho = \bigcup_{i \in \mathbb{N}} \left( R_{x}^{(i)} \cdot [t] (\rho[r \leftarrow i]) \right)$

Semantics of programs. Obtained by interleaving ($\otimes$):

$[t_1 || \ldots || t_n] = [t_1] \emptyset \otimes \ldots \otimes [t_n] \emptyset$

Example. Define $p = (x := 1; y \leftarrow r || y := 1; x \leftarrow s)$

- $W_{x}^{(1)} \cdot W_{y}^{(1)} \cdot R_{y}^{(3)} \cdot R_{x}^{(2)} \in [p]$
- but $R_{x}^{(0)} \cdot R_{y}^{(0)} \cdot W_{x}^{(1)} \cdot W_{y}^{(1)} \not\in [p]$. 
Closed semantics

Obtained by eliminating “inconsistent” traces (eg. $W_x^{(2)} \cdot R_x^{(3)}$)

**Linear memory model.** A language of “consistent” traces:

$$M(\mu : \mathcal{V} \to \mathbb{N}) ::= \epsilon$$

$$| \ R_x^{(\mu(x))} \cdot M(\mu)$$

$$| \ W_x^{(k)} \cdot M(\mu[x \leftarrow k])$$

$$M ::= M(x \mapsto 0)$$

Closed semantics: $[p] = [p]^O \cap M$.

**Example.** Write $p = (x := 1; r \leftarrow y) \parallel (y := 2; s \leftarrow x)$

- every trace of $[p]$ ends with $R_x^{(1)}$ or a $R_y^{(2)}$. 
Summary

Advantages.

- Easy to define semantics, by induction on programs.
- By making $M$ more complex, complex cache schemes can be handled.

Drawbacks.

- Combinatorial explosion due to interleavings.
- How to model reordering of instructions?

Towards partial-orders.

- Because of reorderings, threads are not totally ordered.
- Our goal: compute fine precisely dependencies between the instructions, given an architecture.
II. Event structures

*Raiders of the lost causality*
Replacing traces by partial-orders

**Idea:** volatile semantics should be a set of partial-orders.

Term:

\[
\begin{align*}
x & := 1; y := 1; \\
r & \leftarrow x; s \leftarrow y; \\
z & := s + t
\end{align*}
\]
Replacing traces by partial-orders

**Idea:** volatile semantics should be a set of partial-orders.

Dependencies (depends on the architecture):

\[
\begin{align*}
x & := 1 \\
\downarrow & \\
r & \leftarrow x
\end{align*}
\quad
\begin{align*}
y & := 1 \\
\downarrow & \\
s & \leftarrow y
\end{align*}
\quad
\begin{align*}
z & := r + s
\end{align*}
\]

Traces on $\Sigma_c$ becomes partially ordered multisets over $\Sigma_c$ ($\text{pomsets}$).

Problem: lots of redundancies in the pomsets..
Replacing traces by partial-orders

**Idea**: volatile semantics should be a set of partial-orders.

Executions (depends on the architecture):

\[
\begin{align*}
W_x^{(1)} & \Rightarrow W_y^{(1)} \\
R_x^{(i)} & \Rightarrow R_y^{(i)} \\
W_z^{(i+j)} &
\end{align*}
\]

for \(i, j \in \mathbb{N}^2\).

- traces on \(\Sigma_c\) becomes *partially ordered multisets* over \(\Sigma_c\) (pomsets)
- \([t]^O\) becomes a set of such *pomsets*. 
Replacing traces by partial-orders

Idea: volatile semantics should be a set of partial-orders.

Executions (depends on the architecture):

\[ W_x^{(1)} \downarrow \quad W_y^{(1)} \downarrow \]
\[ R_x^{(i)} \quad R_y^{(j)} \]
\[ \triangleright \quad \triangleright \]
\[ W_z^{(i+j)} \]

for \( i, j \in \mathbb{N}^2 \).

- traces on \( \Sigma_c \) becomes \textit{partially ordered multisets} over \( \Sigma_c \) (pomsets)
- \( \llbracket t \rrbracket^D \) becomes a set of such \textit{pomsets}.
- \textbf{Problem}: lots of redundancies in the pomsets.
Can we sum up all executions in a single object?

Can we glue the executions all together in a partial-order? For instance:

Which sets of events $w$ are (partial) executions?

- $w$ must be downward-closed for $\rightarrow$
Can we sum up *all* executions in a single object?

Can we glue the executions all together in a partial-order? For instance:

\[ W(1)_x \rightarrow W(1)_x \rightarrow R(0)_x \rightarrow W(0)_z \rightarrow W(1)_z \rightarrow R(1)_x \rightarrow \ldots \]
\[ R(0)_y \rightarrow R(1)_y \rightarrow W(1)_y \rightarrow W(0)_z \rightarrow W(1)_z \rightarrow R(1)_y \rightarrow \ldots \]

Which sets of events \( w \) are (partial) executions?

- \( w \) must be downward-closed for \( \rightarrow \)
- and \ldots? \( \{ W^{(1)}_x, R^{(0)}_x, R^{(1)}_x \} \) cannot be a valid execution.
Can we sum up all executions in a single object? Can we glue the executions all together in a partial-order? For instance:

\[
W_x^{(1)} \quad W_y^{(1)}
\]

\[
R_x^{(0)} \quad R_x^{(1)} \quad R_y^{(0)} \quad R_y^{(1)}
\]

\[
W_z^{(0)} \quad W_z^{(1)} \quad W_z^{(2)} \quad W_z^{(1)}
\]

Which sets of events \( w \) are (partial) executions?

- \( w \) must be downward-closed for \( \rightarrow \)
- and ...? \( \{W_x^{(1)}, R_x^{(0)}, R_x^{(1)}\} \) cannot be a valid execution.

\[ \Rightarrow \text{Need more structure than a partial-order: conflicts.} \]
Event structures save the day

Definition (Event structures)
A set of event $E$ with:

- A notion of causality represented by a partial order $\leq_E$
- A notion of conflict represented by a relation $\sim_E$
- A labelling $l : E \rightarrow \Sigma$.

(+ axioms)

Definition (Configuration or partial execution)
A configuration of $E$ is a subset $w$ of $E$:

- downward-closed: $e \leq e' \in w \Rightarrow e \in w$.
- that does not contain two conflicting events
Event structures save the day

On the example:

We have the configuration:
Event structures save the day

On the example:

We have the configuration:

\[ W_x^{(1)} \]
Event structures save the day

On the example:

Weak memory models using event structures · Simon Castellan
Event structures save the day

On the example:

We have the configuration:

\[
\begin{align*}
  W^{(1)}_x & \quad R^{(1)}_x \\
  R^{(0)}_x & \quad W^{(0)}_z \\
  W^{(1)}_z & \quad W^{(2)}_z \\
  R^{(0)}_y & \quad R^{(1)}_y \\
  W^{(1)}_y & \quad R^{(1)}_x
\end{align*}
\]
Event structures save the day

On the example:

We have the configuration:
Event structures save the day

On the example:

We have the configuration:
III. Designing a semantics with event structures

Dessine-moi une structure d’événements
Defining architectures

Now we define an architecture $\mathcal{A}$ as a pair $(\to_{\mathcal{A}}, E)$:

- $\to_{\mathcal{A}} \subseteq \Sigma_a \times \Sigma_a$ indicates which causality cannot be erased.
- $E_{\mathcal{A}}$ is an event structure representing the memory model.

Examples for $\to_{\mathcal{A}}$:
- $\to_{\text{SC}} = \Sigma_a \times \Sigma_a$
- $\to_{\text{ARM}} = \{(e, e') \mid v(e) = v(e')\} \ (v(x, _) = x)$.
- $\to_{\text{x86}} = ...$

Examples for $E_{\mathcal{A}}$ include all languages $M \subseteq \Sigma_c^*$ (they can be viewed as event structures).
Computing the semantics $[p]_\mathcal{A}$

As previously, in two steps:

- **Volatile semantics:**
  - *threads:* $[t]_\mathcal{A}^O$ is defined as previously but where the causality outside $\rightarrow_\mathcal{A}$ are relaxed.
  - *programs:* $[t_1 \parallel \ldots \parallel t_n]_\mathcal{A}^O = [t_1]_\mathcal{A}^O \parallel \ldots \parallel [t_n]_\mathcal{A}^O$
    where $\parallel$ is *parallel composition*.

- **Closed semantics:** $[p]_\mathcal{A} = [p]_\mathcal{A}^O \wedge E_\mathcal{A}$
  where $\wedge$ is the *synchronized product*: a generalization of intersection of languages to event structures.
Volatile semantics

Pour \( t = \left( \begin{array}{l}
\text{\texttt{s} } \leftarrow \text{\texttt{x}; x} \texttt{ := s;} \\
\text{\texttt{t} } \leftarrow \text{\texttt{y}; y} \texttt{ := t;} \\
\text{\texttt{z} } := \text{\texttt{s} + t}
\end{array} \right) \), on a:

(SC)
Volatile semantics

Pour \( t = (s \leftarrow x; x := s; \) \)
\( t \leftarrow y; y := t; \) \), on a:
\( z := s + t \)

\[
\begin{align*}
R_x^{(0)} & \xrightarrow{y} W_x^{(0)} \\
R_y^{(0)} & \xrightarrow{y} W_y^{(0)} \\
W_z^{(0)} & \xrightarrow{y} W_z^{(0)} \\
W^{(0)} & \xrightarrow{y} W^{(0)} \\
W^{(1)} & \xrightarrow{y} W^{(1)} \\
W^{(2)} & \xrightarrow{y} W^{(2)} \\
(\text{SC}) & \\
\end{align*}
\]
Volatile semantics

Pour $t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}$, on a:

(x86)
Volatile semantics

Pour \( t = \left( s \leftarrow x; x := s; \right) \), on a:
\[
\begin{align*}
  t & \leftarrow y; y := t; \\
  z & := s + t
\end{align*}
\]
Volatile semantics

Pour $t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}$, on a:
Volatile semantics

Pour $t = \left( s \leftarrow x; x := s; \right) \left( t \leftarrow y; y := t; \right)$, on $a$:

$$z := s + t$$
Volatile semantics

Pour $t = \begin{cases} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{cases}$, on a:

$$W^{(0)}_x \rightsquigarrow W^{(1)}_x \rightsquigarrow R^{(0)}_x \rightsquigarrow R^{(1)}_x \rightsquigarrow W^{(2)}_z \rightsquigarrow W^{(1)}_z \rightsquigarrow W^{(1)}_z \rightsquigarrow R^{(0)}_y \rightsquigarrow R^{(1)}_y \rightsquigarrow W^{(0)}_y \rightsquigarrow W^{(1)}_y$$
The memory model $\mathcal{E}$

Define a **consistent execution** to be a $\Sigma_c$-labelled partial-order $(q, \leq_q)$ satisfying:

1. **Write serialization.** Writes on a variable are totally ordered.

   $$W_x^{(1)} \rightarrow W_x^{(3)} \rightarrow W_x^{(4)} \rightarrow W_x^{(2)} \rightarrow W_y^{(0)}$$

2. **Coherent reading.** For $e = R_x^{(k)} \in q$, $W_x^{(k)}$ is the maximal event of $\{W_x^{(n)} \in q \mid W_x^{(n)} \leq e\}$

   $$W_x^{(2)} \rightarrow W_x^{(3)} \rightarrow R_y^{(0)} \rightarrow R_x^{(3)}$$

**Theorem.** There is an event structure $\mathcal{E}$ whose configurations are exactly consistent partial-orders.
Example

\[ p = \begin{align*}
  &x := 1 \quad | \quad y := 1 \\
  &r_1 \leftarrow x \quad | \quad s_1 \leftarrow y \\
  &r_2 \leftarrow y \quad | \quad s_2 \leftarrow x
\end{align*} \]

(Volatile semantics for SC)
Example

\[ p = \]
\[ \begin{align*}
  x & := 1 \\
  y & := 1 \\
  r_1 & \leftarrow x \\
  s_1 & \leftarrow y \\
  r_2 & \leftarrow y \\
  s_2 & \leftarrow x \\
\end{align*} \]

\[
\begin{array}{ll}
W_x^{(1)} & W_y^{(1)} \\
R_x^{(0)} \wedge R_x^{(1)} & R_y^{(0)} \wedge R_y^{(1)} \\
R_y^{(0)} \wedge R_y^{(1)} & R_x^{(0)} \wedge R_x^{(1)} \\
R_y^{(0)} \wedge R_y^{(1)} & R_x^{(0)} \wedge R_x^{(1)} \\
\end{array}
\]

(Computing \( \llbracket p \rrbracket_{SC}^O \wedge E \))
Example

\[ p = \]
\[
\begin{align*}
x & := 1 \\
r_1 & \leftarrow x \\
r_2 & \leftarrow y
\end{align*}
\]

\[
\begin{align*}
y & := 1 \\
s_1 & \leftarrow y \\
s_2 & \leftarrow x
\end{align*}
\]

\[
\begin{array}{c}
W_x^{(1)} \\
\swarrow \\
\searrow \quad \swarrow \quad \searrow \\
R_x^{(0)} \wedge R_x^{(1)} \\
\swarrow \\
\searrow \\
R_y^{(0)} \wedge R_y^{(1)} \\
\swarrow \\
\searrow \\
R_y^{(0)} \wedge R_y^{(1)}
\end{array}
\quad
\begin{array}{c}
W_y^{(1)} \\
\swarrow \\
\searrow \quad \swarrow \quad \searrow \\
R_y^{(0)} \wedge R_y^{(1)} \\
\swarrow \\
\searrow \\
R_y^{(0)} \wedge R_y^{(1)} \\
\swarrow \\
\searrow \\
R_x^{(0)} \wedge R_x^{(1)} \quad R_x^{(0)} \wedge R_x^{(1)} \quad R_x^{(0)} \wedge R_x^{(1)}
\end{array}
\]

(Computing \([p]_{sc}^O \land \mathcal{E}\))
Example

\[
p = \begin{align*}
x & := 1 \\
r_1 & \leftarrow x \\
s_1 & \leftarrow y \\
r_2 & \leftarrow y \\
s_2 & \leftarrow x
\end{align*}
\]

We can observe \( r_1 = s_1 = 1 \) \( \land r_2 = s_2 = 0 \).

(Computing \( \llbracket p \rrbracket^0_{\text{sc}} \land \mathcal{E} \))
Example

\[ p = \begin{align*}
  x &: 1 \\
  r_1 &: x \\
  r_2 &: y \\
  y &: 1 \\
  s_1 &: y \\
  s_2 &: x
\end{align*} \]

\[
\begin{array}{c}
W_{x}^{(1)} \\
\downarrow \\
R_{x}^{(1)} \\
\downarrow \\
R_{y}^{(0)} \lor R_{y}^{(1)}
\end{array}
\quad
\begin{array}{c}
W_{y}^{(1)} \\
\downarrow \\
R_{y}^{(1)} \\
\downarrow \\
R_{x}^{(0)} \lor R_{x}^{(1)}
\end{array}
\]

(Computing \( \llbracket p \rrbracket_{SC}^{O} \land \mathcal{E} \))
Example

\[ p = \begin{align*}
x & := 1 \\
r_1 & \leftarrow x \\
r_2 & \leftarrow y \\
y & := 1 \\
s_1 & \leftarrow y \\
s_2 & \leftarrow x
\end{align*} \]

\( W_x^{(1)} \rightarrow \downarrow \)
\( R_x^{(1)} \rightarrow \downarrow \)
\( R_y^{(0)} \wedge R_y^{(1)} \)

\( W_y^{(1)} \rightarrow \downarrow \)
\( R_y^{(1)} \rightarrow \downarrow \)
\( R_x^{(0)} \wedge R_x^{(1)} \)

(Computing \( \llbracket p \rrbracket_{SC}^O \wedge \mathcal{E} \))
Example

\[ p = \begin{align*}
  x &:= 1 & y &:= 1 \\
  r_1 &\leftarrow x & s_1 &\leftarrow y \\
  r_2 &\leftarrow y & s_2 &\leftarrow x
\end{align*} \]

\[ \begin{array}{c}
  W_x^{(1)} \\
  \downarrow \\
  R_x^{(1)} \\
  \downarrow \quad \triangleleft \\
  R_y^{(0)} \wedge R_y^{(1)}
\end{array} \quad \begin{array}{c}
  W_y^{(1)} \\
  \downarrow \\
  R_y^{(1)} \\
  \downarrow \quad \triangleleft \\
  R_x^{(0)} \wedge R_x^{(1)}
\end{array} \]

\[(\text{Computing } [p]_{SC}^O \wedge \mathcal{E})\]
Example

\[
p = \begin{array}{c|c}
  x := 1 & y := 1 \\
  r_1 \leftarrow x & s_1 \leftarrow y \\
  r_2 \leftarrow y & s_2 \leftarrow x \\
\end{array}
\]

\[
\begin{aligned}
  W^{(1)}_x & \downarrow \\
  R^{(1)}_x & \downarrow \\
  R^{(0)}_y \wedge R^{(1)}_y & \\

  W^{(1)}_y & \downarrow \\
  R^{(1)}_y & \downarrow \\
  R^{(0)}_x \wedge R^{(1)}_x & \\
\end{aligned}
\]

\[
(\text{Computing } [p]^{O}_{\text{sc}} \wedge E)
\]
Example

\[ p = \begin{align*}
    x & := 1 & y & := 1 \\
    r_1 & \leftarrow x & s_1 & \leftarrow y \\
    r_2 & \leftarrow y & s_2 & \leftarrow x
\end{align*} \]

\[(\text{Computing } \llbracket p \rrbracket_{sc}^O \land \mathcal{E})\]
Example

\[
p = x := 1 \quad \Vert \quad y := 1
\]

\[
p = r_1 \leftarrow x \quad s_1 \leftarrow y
\]

\[
p = r_2 \leftarrow y \quad s_2 \leftarrow x
\]

\[
\begin{align*}
W_x^{(1)} & \quad W_y^{(1)} \\
R_x^{(1)} & \quad R_y^{(1)} \\
R_x^{(0)} \land R_y^{(1)} & \quad R_x^{(0)} \land R_y^{(1)}
\end{align*}
\]

(Computing $\llbracket p \rrbracket_{sc} \land E$)
Example

\[
p = \begin{align*}
  x & := 1 & y & := 1 \\
  r_1 & \leftarrow x & s_1 & \leftarrow y \\
  r_2 & \leftarrow y & s_2 & \leftarrow x
\end{align*}
\]

\[
\begin{align*}
  W_x^{(1)} & \downarrow & W_y^{(1)} & \downarrow \\
  R_x^{(1)} & \downarrow & R_y^{(1)} & \downarrow \\
  R_x^{(0)} & \wedge R_y^{(1)} & & \wedge R_y^{(0)} \\
  & & & \wedge R_x^{(1)}
\end{align*}
\]

(Computing \( [p]^{O}_{sc} \land \mathcal{E} \))

We can observe \( r_1 = s_1 = 1 \land r_2 = s_2 = 0 \).
is too relaxed

Consider $p =$
\[
\begin{pmatrix}
x := 1 \\
r \leftarrow x \\
s \leftarrow y \\
y := 1 \\
t \leftarrow x
\end{pmatrix}
\]

The denotation $\llbracket p \rrbracket_{SC} \land \mathcal{E}$ contains the configuration:

\[
\begin{array}{c}
W_x^{(1)} \rightarrow R_x^{(1)} \\
\downarrow \\
R_y^{(0)} \rightarrow R_x^{(0)}
\end{array}
\]

This allows the observation: $r = 1 \land s = t = 0$ which is not possible with TSO (x86’s memory model).

**Problem.** With TSO, writes becomes visible to all others threads at the same time.
Defining $E_{TSO}$

1. We need our model to be “thread-aware”:

$$
W_{x}^{(1,1)} \Rightarrow R_{x}^{(2,1)} \quad W_{y}^{(3,1)} \\
\downarrow \quad \downarrow \\
R_{y}^{(2,0)} \quad R_{x}^{(3,0)}
$$

2. Say a consistent execution satisfies the TSO criterion, when:

   for all writes $w \in q$,
   
   for all *incomparable* reads $r, r' \in q$ in a different thread than $w$
   
   $(w \leq r)$ iff $(w \leq r')$

3. Define $E_{TSO}$ to be the set of consistent execution satisfying this criterion.
IV. The game semantics behind all that

La sémantique des jeux vue du ciel
Idealized Parallel Algol

Throwing in simply-typed λ-calculus to our language we get IPA:

\[
A, B := \text{int} | \text{var} | \text{unit} | A \Rightarrow B
\]

\[
t, u := x | \lambda x. \, t | t \, u
\]

\[
| \text{read}^{\text{var} \rightarrow \text{unit}} | \text{write}^{\text{var} \rightarrow \text{int} \rightarrow \text{unit}}
\]

\[
| \text{new} \, x^{\text{var}} \, \text{in} \, t \quad (t \text{ has type int or unit})
\]

\[
| (t; \, u) | (t \, || \, u)
\]

- Comes with an SC and call-by-name operational semantics.
- Giving semantics: a semantics for λ-calculus plus operators for read, write, ...
- Games semantics: types → games, programs → strategies.
- We have good trace-based games model for that.
The usual strategy for read

An example.

\[ x : \text{var} \rightarrow \text{int} \]

**Problem.** No access to the continuation to break causalities.
The usual strategy for read

An example.

\[ x : \text{var} \rightarrow \text{int} \]

\[ \text{ask} \]

Problem. No access to the continuation to break causalities.
The usual strategy for read

An example.

\[ x : \text{var} \rightarrow \text{int} \]

\[ \text{ask} \]

\[ \text{rd} \]

**Problem.** No access to the continuation to break causalities.
The usual strategy for read

An example.

\[ x : \text{var} \rightarrow \text{int} \]

\[ \text{ask} \]
\[ \text{rd} \]
\[ \downarrow \]
\[ k \]

**Problem.** No access to the continuation to break causalities.
The usual strategy for read

An example.

\[ x : \text{var} \rightarrow \text{int} \]

Problem. No access to the continuation to break causalities.
Changing the type of read

The read operation becomes let : var → (int → unit) → unit:

\[
\text{let } \text{read } x f = \\
\text{let } z = !x \ \text{in } f z
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \( \text{let } : \text{var} \rightarrow (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit} : \)

\[
\text{let } \text{read } x f = \\
\text{let } z = !x \text{ in } f z
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]

\[\text{run} \]
Changing the type of read

The read operation becomes \( \text{let}: \var \rightarrow (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}: \)

\[
\text{let \ read \ x \ f =} \\
\text{let \ z = !x \ in \ f \ z}
\]

This gives the following strategy:

\[
x: \var \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \texttt{let \: var \rightarrow (int \rightarrow unit) \rightarrow unit}:

\[
\texttt{let \ read \ x \ f = let \ z = !x \ in \ f \ z}
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \texttt{let : var \to (int \to unit) \to unit:}

\begin{verbatim}
let read x f =
  let z = !x in f z
\end{verbatim}

This gives the following strategy:

\[ x : \text{var} \to f : (\text{int} \to \text{unit}) \to \text{unit} \]
Changing the type of read

The read operation becomes \( \text{let : var} \rightarrow (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit} \):

\[
\text{let } \text{read } x \ f = \ \\
\text{let } z = !x \ \text{in } f \ z
\]

This gives the following strategy:

\[
x: \text{var} \rightarrow f: (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \( \text{let} \ var \rightarrow (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit} \):

\[
\text{let} \ \text{read} \ x \ f = \\
\text{let} \ z = !x \ \text{in} \ f \ z
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \( \text{let} : \text{var} \rightarrow (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit} : \)

\[
\begin{align*}
\text{let} & \quad \text{read} \ x \ f = \\
\text{let} & \quad z = !x \ \text{in} \ f \ z
\end{align*}
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Changing the type of read

The read operation becomes \( \text{let } \text{read} \; x \; f = \) \( \text{let } z = !x \text{ in } f \; z \) \:

This gives the following strategy:

\( x : \text{var} \to f : (\text{int} \to \text{unit}) \to \text{unit} \)
Adding concurrency in the mix

But we have space to make it more concurrent!

\[
\text{let read } x \ f = \\
\quad \text{let thr = spawn (fun () \rightarrow !x) in} \\
\quad f \ (\text{lazy (wait thr)})
\]

This gives the following strategy:

\[
x: \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

```ocaml
let read x f =
  let thr = spawn (fun () -> !x) in
  f (lazy (wait thr))
```

This gives the following strategy:

```latex
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
```

run
Adding concurrency in the mix

But we have space to make it more concurrent!

```ocaml
let read x f =
  let thr = spawn (fun () -> !x) in
  f (lazy (wait thr))
```

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

```ocaml
let read x f =
  let thr = spawn (fun () -> !x) in
  f (lazy (wait thr))
```

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

\[
\text{let } \text{read } x \ f = \\
\text{let } \text{thr } = \text{spawn } (\text{fun } () \rightarrow !x) \ \text{in} \\
f (\text{lazy } (\text{wait } \text{thr}))
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

```haskell
let read x f =
  let thr = spawn (fun () -> !x) in
  f (lazy (wait thr))
```

This gives the following strategy:

\[
\begin{align*}
  x &: \text{var} \\
  f &: (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\end{align*}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

\[
\text{let read } x \ f = \text{let thr = spawn (fun () -> !x) in f (lazy (wait thr))}
\]

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Adding concurrency in the mix

But we have space to make it more concurrent!

```ocaml
let read x f =
  let thr = spawn (
    fun () -> !x
  ) in
  f (lazy (wait thr))
```

This gives the following strategy:

\[ x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit} \]
Adding concurrency in the mix

But we have space to make it more concurrent!

```ocaml
let read x f =
  let thr = spawn (fun () -> !x) in
  f (lazy (wait thr))
```

This gives the following strategy:

\[
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\]
Example

Consider \( t = \text{let } x (\lambda n.\text{write } y 1; n + 1): \)

\[ x : \text{var} \rightarrow y : \text{var} \rightarrow \text{int} \]
Example

Consider \( t = \text{let } x (\lambda n. \text{write } y 1; n + 1): \)

\[
\begin{align*}
x &: \text{var} & y &: \text{var} & \rightarrow & \text{int} \\
\end{align*}
\]

ask
Consider $t = \text{let } x (\lambda n. \text{write } y 1; n + 1)$:

$x : \text{var} \rightarrow y : \text{var} \rightarrow \text{int}$
Consider $t = \text{let } x (\lambda n. \text{write } y 1; n + 1)$:

$x : \text{var} \rightarrow y : \text{var} \rightarrow \text{int}$

weak_memory_models_using_event_structures_simon_castellan_32_33
Example

Consider $t = \text{let } x (\lambda n.\text{write } y 1; n + 1)$:

$$x : \text{var} \rightarrow y : \text{var} \rightarrow \text{int}$$

\[\text{rd} \quad \text{write}_1 \quad \text{ask} \quad n\]
Example

Consider \( t = \text{let } x (\lambda n. \text{write } y 1; n + 1) : \)

\[ x : \text{var} \rightarrow y : \text{var} \rightarrow \text{int} \]

\( \text{rd} \leftarrow \text{write}_1 \leftarrow \text{ask} \)

\( n \leftarrow \text{ok} \)
Consider $t = \text{let } x \ (\lambda n.\text{write } y \ 1; \ n + 1) \colon$

$x : \text{var} \to y : \text{var} \to \text{int}$
Conclusion

Summary.

- We defined an *denotational* and *extensible* interpretation of concurrent programs in terms of *event structures*.
- The interpretation is parametric over the architecture.

Extensions.

- We can define sub-models of $E$ corresponding to actual architectures.
- The model is inspired from a game semantics model and simplified in this first-order setting.

To go further.

- Look at barriers
- Compare that with axiomatic semantics (executions)
- Theorems?