Weak memory models using event structures

Simon Castellan\textsuperscript{1}

\textsuperscript{1}LIP, ENS Lyon

November 26th, 2016
Dagstuhl Seminar
A simple weak memory model: TSO

In this talk, we will focus on a simple weak memory model: TSO.

**Store buffering.** (can observe $r = s = 0$ on TSO but not SC):

\[
x, y \text{ initialized to 0} \\
x := 1 \quad y := 1 \\
r \leftarrow y \quad s \leftarrow x
\]

Implementation: thread-local write buffers.

\[
\langle t_1 \parallel \ldots \parallel t_n @ (\mu : \mathcal{V} \rightarrow \mathbb{N}) \rangle
\]

States of a SC machine

becomes

\[
\langle t_1 : \kappa_1 \in (\mathcal{V} \times \mathbb{N})^* \parallel \ldots \parallel t_n : \kappa_n @ \mu \rangle
\]

State of a TSO machine

Some transition rules:

\[
(Write) \quad \langle (x := k; t : b) @ \mu \rangle \rightarrow \langle (t : b++[(x, k)]) @ \mu \rangle
\]

\[
(Commit) \quad \langle (t : [(x, k)]++b) @ \mu \rangle \rightarrow \langle (t : b) @ \mu [x \leftarrow k] \rangle
\]
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. An example: a model for **TSO**.
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:

\[
e, e' ::= \{ \text{Expressions} \}
\]
\[
k \in \mathbb{N} \mid r \in \mathcal{R} \mid e + e'
\]
\[
\iota ::= \{ \text{Instructions} \}
\]
\[
| a := e \quad \text{(Write on a variable)}
\]
\[
| r \leftarrow a \quad \text{(Read on a variable)}
\]
\[
t ::= \{ \text{Threads} \}
\]
\[
| \iota; \ldots; \iota
\]
\[
p ::= \{ \text{Programs} \}
\]
\[
t_1 \parallel \ldots \parallel t_n
\]

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.

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

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

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

Notations: \((\tau : R_{x=k})\), \((\tau : W_{x:=k})\). \((\tau):\) thread-id

Two steps:

1. **Thread 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 \(r\).
Thread semantics

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

(Writes) $\lbrack x := e; t \rbrack(\rho, \tau) = (\tau : W_{x := \rho(e)} \cdot \lbrack t \rbrack \rho$

(Reads) $\lbrack r \gets x; t \rbrack(\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : R_{x = i} \cdot \lbrack t \rbrack(\rho[r \gets i], \tau))$
Thread semantics

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

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

(Reads) $[r \leftarrow x; t](\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : R_{x:=i}) \cdot [t](\rho[r \leftarrow i], \tau))$

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

$[t_1 \parallel \ldots \parallel t_n] = [t_1](\emptyset, 1) \otimes \ldots \otimes [t_n](\emptyset, n)$
Thread semantics

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

\begin{align*}
(\text{Writes}) & \quad [x := e; t](\rho, \tau) = (\tau : W_{x := \rho(e)} \cdot [t])\rho \\
(\text{Reads}) & \quad [r \leftarrow x; t](\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : R_{x = i} \cdot [t](\rho[r \leftarrow i], \tau))
\end{align*}

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

$$[[ t_1 \parallel \ldots \parallel t_n ]] = [[[ t_1 ]](\emptyset, 1) \otimes \ldots \otimes [[[ t_n ]](\emptyset, n)$$

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

- $W_{x := 1} \cdot W_{y := 1} \cdot R_{y = 3} \cdot R_{x = 2} \in [[[ p ]]]$
- but $R_{x = 0} \cdot R_{y = 0} \cdot W_{x := 1} \cdot W_{y := 1} \not\in [[[ p ]]]$. 

Weak memory models using event structures · Simon Castellan
Closed semantics

Obtained by eliminating “inconsistent” traces (eg. \( W_{x:=2} \cdot R_{x=3} \))

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

\[
M(\mu : V \rightarrow N) ::= \epsilon \\
| \tau : R_{x=\mu(x)} \cdot M(\mu) \\
| \tau : W_{x:=k} \cdot M(\mu[x \leftarrow k])
\]

\[
M ::= M(x \mapsto 0)
\]

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: thread semantics should be a set of partial-orders.

Term:

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

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

Dependencies (depends on the architecture):

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

\[\text{traces on } \Sigma \text{ becomes partially ordered multisets over } \Sigma \text{ (pomsets)} \]

\[J, t, K, O \text{ becomes a set of such pomsets.} \]

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

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

**Executions** (depends on the architecture):

\[
\begin{align*}
W_x & = 1 \\
R_x & = i \\
W_z & = i + j
\end{align*}
\]

\[
\begin{align*}
W_y & = 1 \\
R_y & = j
\end{align*}
\]

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

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

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

**Executions (depends on the architecture):**

\[
\begin{align*}
W_x &= 1 \\
R_x &= i \\
W_y &= 1 \\
R_y &= j \\
W_z &= i + j
\end{align*}
\]

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

- traces on \( \Sigma_c \) becomes *partially ordered multisets* over \( \Sigma_c \) (pomsets)
- \( \llbracket t \rrbracket^O \) becomes a set of such *pomsets*.
- **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:

\[
\begin{align*}
W_x &= 1 \\
R_x &= 0 \\
W_z &= 0 \\
R_x &= 1 \\
W_z &= 1 \\
\cdots
\end{align*}
\]

\[
\begin{align*}
W_y &= 1 \\
R_y &= 0 \\
W_z &= 1 \\
R_y &= 1 \\
W_z &= 1 \\
\cdots
\end{align*}
\]

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:

\[
\begin{align*}
W_x &= 1 \\
R_x &= 0 \\
\end{align*}
\]

\[
\begin{align*}
W_x &= 1 \\
R_x &= 1 \\
\end{align*}
\]

\[
\begin{align*}
R_x &= 0 \\
W_z &= 0 \\
\end{align*}
\]

\[
\begin{align*}
W_z &= 1 \\
R_x &= 1 \\
\end{align*}
\]

\[
\begin{align*}
R_x &= 0 \\
W_z &= 2 \\
\end{align*}
\]

\[
\begin{align*}
W_z &= 1 \\
R_y &= 0 \\
\end{align*}
\]

\[
\begin{align*}
R_y &= 1 \\
W_y &= 1 \\
\end{align*}
\]

\[
\begin{align*}
W_y &= 1 \\
R_y &= 1 \\
\end{align*}
\]

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

- \( w \) must be downward-closed for \( \rightarrow \)
- and \( \ldots \)? \( \{W_x = 1, R_x = 0, R_x = 1\} \) 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:

\[
\begin{align*}
W_x &= 1 \\
R_x &= 0 \\
W_z &= 0
\end{align*}
\]

\[
\begin{align*}
W_y &= 1 \\
R_y &= 0 \\
W_z &= 1
\end{align*}
\]

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

- \( w \) must be downward-closed for \( \rightarrow \)
- and \( \ldots \)? \( \{W_x = 1, R_x = 0, R_x = 1\} \) 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:

\[
\begin{align*}
W_x &= 1 \\
W_y &= 1 \\
R_x &= 0 \\
R_y &= 0 \\
W_z &= 0 \\
W_z &= 1 \\
W_z &= 2 \\
W_z &= 1 \\
\cdots
\end{align*}
\]
Event structures save the day

On the example:

We have the configuration:

\( W_x := 1 \)
Event structures save the day

On the example:

We have the configuration:
Event structures save the day

On the example:

\[
\begin{align*}
W_x & := 1 \\
R_x & = 0 \quad R_x = 1 \\
W_z & := 0 \quad W_z := 1 \\
W_y & := 1 \\
R_y & = 0 \quad R_y = 1
\end{align*}
\]

We have the configuration:

\[
\begin{align*}
W_x & := 1 \\
R_x & = 1
\end{align*}
\]
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
A model for the TSO architecture

We now repeat the story using event structures for TSO.

Two steps:

- **Open semantics:** \([ t ]^O\) is an event structure
- **Closed semantics:** \([ t ] = [ t ]^O \land M_{TSO}\)

Store buffering:

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

becomes:

\[
\begin{align*}
W_x := 1 \quad \parallel \quad W_y := 1 \\
R_y = 0 & \quad \parallel \quad R_y = 1 \\
R_x = 1 & \quad \parallel \quad R_x = 0
\end{align*}
\]
Thread semantics

By induction as before, generalizing operations to event structures.

**Threads**: (omitting thread-ids)

\[ \mathcal{T}[x := e; t] \rho = \mathcal{W}_{x:=\rho(e)} \cdot [t] \rho \]

\[ \mathcal{W}_{x:=\rho(e)} \]

\[ [t] \rho \]

\[ \mathcal{T}[r \leftarrow x; t] \rho = \sum_{i \in \mathbb{N}} R_{x=i} \cdot [t] (\rho[r \leftarrow i]) \]

\[ R_{x=0} \]

\[ R_{x=1} \]

\[ \ldots \]

\[ [t] (\rho[r \leftarrow 0]) \]

\[ [t] (\rho[r \leftarrow 1]) \]

\[ \ldots \]

**Programs**:

\[ \mathcal{T}[t_1 \parallel \ldots \parallel t_n] = [t_1](\emptyset, 1) \parallel \ldots \parallel [t_n](\emptyset, n) \]

\[ [t_1] \emptyset \]

\[ \ldots \]

\[ [t_n] \emptyset \]
Consistent memory behaviours

A Σ-labeled partial order is TSO-consistent when it satisfies:

1. **Write serialization.** Writes on a variable are totally ordered.
   
   \[ W_x := 1 \rightarrow W_x := 3 \rightarrow W_x := 4 \]
   
   \[ W_y := 2 \rightarrow W_y := 0 \]

2. **Coherent reading.** For \( e = R_{x=k} \in q \), \( W_{x=k} \) is the maximal event of \( \{ W_{x=n} \in q \mid W_{x=n} \leq e \} \)
   
   \[ W_y := 2 \]
   
   \[ W_x := 2 \rightarrow W_x := 3 \rightarrow R_y = 0 \rightarrow R_x = 3 \]

3. **Writes propagation.** For all writes \( w \in q \), and for all incomparable reads \( r, r' \in q \) in a different thread than \( w \), \( (w \leq r) \text{ iff } (w \leq r') \)

4. **Thread sequentialization** Two events from the same thread are comparable [unless it is an independent read & write pair].
\( M_{TSO} \) and the synchronized product

**Theorem**

There exists an event structure \( M_{TSO} \) whose configurations are exactly consistent TSO-execution.

(Relies on TSO execution being closed under “prefix”)

How to combine \( \mathcal{T}[t] \) and \( M_{TSO} \)? Using the **synchronized product**:

\[
[t] = \mathcal{T}[t] \land M_{TSO}.
\]
Example

\[
p = \begin{align*}
x & := 1 \quad | \quad y := 1 \\
r & \leftarrow y \quad | \quad s \leftarrow x
\end{align*}
\]

(\text{Thread semantics})
Example

\[ p = x := 1 \parallel y := 1 \]
\[ r \leftarrow y \parallel s \leftarrow x \]

\[
\begin{array}{c}
W_{x:=1} \\
\downarrow \\
R_{y=0} \quad R_{y=1}
\end{array}
\quad \quad
\begin{array}{c}
W_{y:=1} \\
\downarrow \\
R_{x=1} \quad R_{x=0}
\end{array}
\]

(Computing \( \mathcal{T}[p] \land \mathcal{M}_{\text{TSO}} \))
Example

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

\[ W_x := 1 \quad \begin{align*}
  R_{y=0} & \quad R_{y=1}
\end{align*} \quad W_y := 1 \quad \begin{align*}
  R_{x=1} & \quad R_{x=0}
\end{align*} \]

(Computing \( \mathcal{T}[p] \land \mathcal{M}_{TSO} \))
Example

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

\[ W_x := 1 \quad W_y := 1 \]

\[ R_y = 0 \quad R_y = 1 \quad R_x = 1 \quad R_x = 0 \]

(Computing \( T[p] \land M_{\text{TSO}} \))
Example

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

\[ W_x := 1 \quad \begin{array}{c}
  R_y = 0 \\
  R_y = 1
\end{array} \quad \begin{array}{c}
  W_y := 1 \\
  R_x = 1 \\
  R_x = 0
\end{array} \]

(Computing \( \mathcal{T}[p] \land \mathcal{M}_{TSO} \))
Example

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

\[ (\text{Computing } \mathcal{T}[p] \land M_{TSO}) \]
Example

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

\( W_x := 1 \)

\( W_y := 1 \)

\( R_y = 0 \)

\( R_y = 1 \)

\( R_x = 1 \)

\( R_x = 0 \)

(Computing \( T[p] \land M_{TS0} \))
Example

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

\[ W_x := 1 \quad W_y := 1 \]

\[ R_y = 0 \quad R_y = 1 \quad R_x = 1 \quad R_x = 0 \]

(Computing \( \mathcal{T}[p] \land M_{TSO} \))
Example

\[
p = x := 1 \parallel y := 1 \\
    r \leftarrow y \parallel s \leftarrow x
\]

(Computing \( T[p] \land M_{TSO} \))
Example

\[ p = \begin{array}{c}
  x := 1 \\
  r \leftarrow y
\end{array} \parallel \begin{array}{c}
  y := 1 \\
  s \leftarrow x
\end{array} \]

\[ W_x := 1 \quad \downarrow \quad W_y := 1 \]

\[ R_y = 0 \quad \cdots \cdots \quad R_y = 1 \quad \downarrow \quad R_x = 1 \quad \cdots \cdots \quad R_x = 0 \]

(Computing \( T[p] \land M_{TSO} \))
Example

\[ p = x := 1 \parallel y := 1 \]
\[ r \leftarrow y \parallel s \leftarrow x \]

(Computing \( \mathcal{T}[p] \land M_{TSO} \))

We can observe \( r = 0 \land s = 0 \).
A trace of $[t]$ is a linearization of a configuration of $[t]$

**Theorem**
The traces of $[t]$ are in one-to-one correspondance between the usual operational semantics for TSO.

However, there are no *explicit* buffers in our semantics.

*Implicitly* represented by *concurrency*: If $R_{x=k}$ is concurrent to $W_{x:=k'}$, the read does not see the write.
Implicit vs. explicit

Our model is *implicit*: no internal events

*Implicit semantics of SB.*

\[
W_x := 1 \\
W_y := 1 \\
R_y = 0 \\
R_y = 1 \\
R_x = 1 \\
R_x = 0
\]

*Explicit semantics of SB.*

\[
W_x := 1 \\
W_y := 1 \\
C_x := 1 \\
C_y := 1 \\
R_y = 0 \\
R_y = 1 \\
R_x = 1 \\
R_x = 0
\]
Reordering

In TSO, it is sound to reorder a write followed by an independent read.

This changes the *thread semantics*.

\( \mathcal{T}^{[SB]} \) becomes:

\[
W_x := 1 \quad R_y = 0 \land R_y = 1 \quad W_y := 1 \quad R_x = 0 \land R_x = 1
\]
IV. THE GAME SEMANTICS BEHIND THAT

*Finding nails for a hammer*
A quick overview of game semantics

Game semantics: *interactive* semantics for higher-order computation.

- Types $\rightarrow$ Games (set of moves + rules)
- Programs $\rightarrow$ Rule-preserving strategies (set of "valid plays")

Objective: use game semantics to reformulate thread semantics.

Instead of

$$[[r \leftarrow x; t]]\rho = \text{complicated surgery on } [[t]]\rho$$

replace it by:

$$[[r \leftarrow x; t]]\rho = \text{let } \odot (x, [[\lambda r.t]])$$

where:

- $\odot$: strategy composition
- let is a carefully-written strategy.
A strategy for read

Usually, reads are interpreted by a strategy \( \text{read} : \text{var} \rightarrow \text{unit} : \)

\[
\text{read} : \quad (x : \text{var}) \rightarrow \text{int}
\]
A strategy for read

Usually, reads are interpreted by a strategy read : var → unit:

$$\text{read} : (x : \text{var}) \rightarrow \text{int}$$

ask
A strategy for read

Usually, reads are interpreted by a strategy $\text{read} : \text{var} \rightarrow \text{unit}$:

$$\text{read} : (x : \text{var}) \rightarrow \text{int}$$
A strategy for read

Usually, reads are interpreted by a strategy \( \text{read} : \text{var} \rightarrow \text{unit} \):

\[
\text{read} : (x : \text{var}) \rightarrow \text{int}
\]
A strategy for read

Usually, reads are interpreted by a strategy \( \text{read} : \text{var} \rightarrow \text{unit} \):

\[
\text{read} : \quad (x : \text{var}) \rightarrow \text{int}
\]
A strategy for read

Usually, reads are interpreted by a strategy \( \text{read} : \text{var} \rightarrow \text{unit} \):

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

Problem. No access to the continuation to break causalities.
Interpreting let

Here let has type \( \text{var} \to (\text{int} \to \text{int}) \to \text{unit} \). For instance:

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

This gives the following strategy:

\[
x : \text{var} \to f : (\text{int} \to \text{unit}) \to \text{unit}
\]
Interpreting let

Here let has type \( \text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit} \). For instance:

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

This gives the following strategy:

\[
\begin{align*}
x : \text{var} \rightarrow f : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\end{align*}
\]
Interpreting let

Here let has type \( \text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit} \). For instance:

```haskell
let read x f =
  let z = !x in f z
```

This gives the following strategy:

\[
\begin{align*}
x & : \text{var} \\
f & : (\text{int} \rightarrow \text{unit}) \rightarrow \text{unit}
\end{align*}
\]
Interpreting let

Here let has type \( \text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit} \). For instance:

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

This gives the following strategy:

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

Here let has type var → (int → int) → unit. For instance:

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

This gives the following strategy:

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

Here let has type \( \text{var} \to (\text{int} \to \text{int}) \to \text{unit} \). For instance:

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

This gives the following strategy:

\[
\begin{align*}
x : \text{var} \to f : (\text{int} \to \text{unit}) \to \text{unit}
\end{align*}
\]
Interpreting let

Here let has type $\text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit}$. For instance:

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

This gives the following strategy:

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

Here let has type \( \text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit} \). For instance:

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

This gives the following strategy:

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

Here let has type \( \text{var} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{unit} \). For instance:

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

This gives the following strategy:

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

This type support more interesting definitions of `let`:

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

This gives the following strategy:

```
x : 'a -> 'b -> unit
```

```
f : ('a -> 'b) -> unit
```
Adding concurrency in the mix

This type support more interesting definitions of let:

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

This gives the following strategy:

```plaintext
x : var -> f : (int -> unit) -> unit
```

run
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

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

This type support more interesting definitions of `let`:

```ml
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

This type support more interesting definitions of `let`:

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

This gives the following strategy:

```
x : var → f : (int → unit) → unit
```

```
run

rd
k
```

run
Adding concurrency in the mix

This type support more interesting definitions of let:

```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

This type support more interesting definitions of \texttt{let}:

\begin{verbatim}
  let read \mathit{x} \mathit{f} =
  let \mathit{thr} = spawn (fun () \rightarrow !\mathit{x}) in
  \mathit{f} (lazy (wait \mathit{thr}))
\end{verbatim}

This gives the following strategy:

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

This type support more interesting definitions of let:

```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

This type support more interesting definitions of let:

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

This gives the following strategy:

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

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

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

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

ask
Example

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

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

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

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

\[\begin{array}{c}
\text{rd} & \text{write}_1 \\
\text{ask} & \\
\end{array}\]
Example

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

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

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

\[
n
\]
Example

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

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

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

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

\[
\begin{array}{ccc}
\text{rd} & \text{write}_1 & \text{ask} \\
\downarrow & \downarrow & \downarrow \\
\text{n} & \text{ok} & \text{n + 1}
\end{array}
\]
A new model

Thread semantics. We use strategies let and write:

\[ T[\text{let } x \ f] = \text{let } \odot \langle x, f \rangle \]
\[ T[x := k] = \text{write } \odot \langle x, k \rangle \]

No more ad-hoc inductive constructions: all is contained in the strategies let and write.
From \( \Gamma \vdash t : A \), we get \( T[t] : [\Gamma] \Rightarrow [A] \).

Storage semantics. \( M_{TSO} \) induces a strategy on \( m_{TSO} : [\text{var}]^n \).

Semantics. For \( x_1 : \text{var}, \ldots, x_n : \text{var} \vdash t : \text{unit} \):

\[ [t] = T[t] \odot m_{TSO} \]
Conclusion

Summary.

▶ We defined an *denotational* and *extensible* interpretation of concurrent programs in terms of *event structures*.
▶ By using the higher-order power of strategies, the behaviour of reads, writes, and the memory are all specified by one event structure.
▶ Because of the game semantics, it scales to function calls, control features, etc.

To go further.

▶ Look at explicit models for weaker architectures (eg. POWER/ARM)
▶ Implicit models for those architecture will need *read-from justifications* (introduced by Jeffrey & Riley)
▶ Software models? (the model is very expressive)