Weak memory models using event structures

Simon Castellan[1]

[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):

$$\begin{array}{c} x, y \text{ initialized to } 0 \\ \begin{array}{c} x \ := \ 1 \\ r \leftarrow y \end{array} \left\| \begin{array}{c} y \ := \ 1 \\ s \leftarrow x \end{array} \right. \end{array}$$

Implementation: thread-local write buffers.

$$\underbrace{\langle t_1 \parallel \ldots \parallel t_n @ (\mu : \mathcal{V} \to \mathbb{N}) \rangle}_{\text{States of a SC machine}}$$

$$\text{becomes} \quad \underbrace{\langle t_1 : \kappa_1 \in (\mathcal{V} \times \mathbb{N})^* \parallel \ldots \parallel t_n : \kappa_n @ \mu \rangle}_{\text{State of a TSO machine}}$$

Some transition rules:

$$\begin{array}{rl} (\mathit{Write}) & \langle (x \ := \ k; t : b) @ \mu \rangle \to \langle (t : b {+}{+} [(x, k)]) @ \mu \rangle \\ (\mathit{Commit}) & \langle (t : [(x, k)] {+}{+} b) @ \mu \rangle \to \langle (t : b) @ \mu[x \leftarrow k] \rangle \end{array}$$

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:

$$
\begin{array}{rll}
e, e' ::= & \{\ \textit{Expressions}\ \} & \\
& k \in \mathbb{N} \mid r \in \mathcal{R} \mid e + e' & \\
\iota ::= & \{\ \textit{Instructions}\ \} & \\
& \mid a := e & \text{(Write on a variable)} \\
& \mid r \leftarrow a & \text{(Read on a variable)} \\
t ::= & \{\ \textit{Threads}\ \} & \\
& \mid \iota; \ldots; \iota & \\
p ::= & \{\ \textit{Programs}\ \} & \\
& t_1 \parallel \ldots \parallel t_n &
\end{array}
$$

In real life: conditionals and barriers.

Denotational semantics

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

Our space here: langages of traces.

$$\begin{aligned}
\Sigma_a &= \mathcal{V} \times \{\mathtt{R}, \mathtt{W}\} && \text{(Abstract memory event)} \\
\Sigma_c &= \mathbb{N} \times \Sigma_a \times \mathbb{N} && \text{(Concrete memory event)} \\
E &= \mathscr{P}({\Sigma_c}^*)
\end{aligned}$$

Notations: $(\tau : \mathtt{R}_{x=k})$, $(\tau : \mathtt{W}_{x:=k})$. ($\tau$: thread-id)

Two steps:

1. Thread semantics $[\![t]\!]^O$: shared variables are considered *volatile*: $[\![x \ \mathtt{:=}\ 1; r \leftarrow x]\!]^O$ 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 $[\![x \ \mathtt{:=}\ 1; r \leftarrow x]\!]$ reads 1 in r.

Thread semantics

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

$$\text{(Writes)} \quad [\![x \;\texttt{:=}\; e; t]\!](\rho, \tau) = (\tau : \texttt{W}_{x := \rho(e)}) \cdot [\![t]\!]\rho$$

$$\text{(Reads)} \quad [\![r \leftarrow x; t]\!](\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : \texttt{R}_{x = i} \cdot [\![t]\!](\rho[r \leftarrow i], \tau))$$

Thread semantics

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

$$\text{(Writes)} \quad [\![x \;\texttt{:=}\; e; t]\!](\rho, \tau) = (\tau : \mathtt{W}_{x:=\rho(e)}) \cdot [\![t]\!]\rho$$

$$\text{(Reads)} \quad [\![r \leftarrow x; t]\!](\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : \mathtt{R}_{x=i} \cdot [\![t]\!](\rho[r \leftarrow i], \tau))$$

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

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

Thread semantics

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

$$\text{(Writes)} \quad [\![x \ := \ e; t]\!](\rho, \tau) = (\tau : \mathtt{W}_{x := \rho(e)}) \cdot [\![t]\!]\rho$$

$$\text{(Reads)} \quad [\![r \ \leftarrow \ x; t]\!](\rho, \tau) = \bigcup_{i \in \mathbb{N}} (\tau : \mathtt{R}_{x=i} \cdot [\![t]\!](\rho[r \leftarrow i], \tau))$$

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

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

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

- $\mathtt{W}_{x:=1} \cdot \mathtt{W}_{y:=1} \cdot \mathtt{R}_{y=3} \cdot \mathtt{R}_{x=2} \in [\![p]\!]$
- but $\mathtt{R}_{x=0} \cdot \mathtt{R}_{y=0} \cdot \mathtt{W}_{x:=1} \cdot \mathtt{W}_{y:=1} \notin [\![p]\!]$.

Closed semantics

Obtained by eliminating "inconsistent" traces (eg. $\mathtt{W}_{x:=2} \cdot \mathtt{R}_{x=3}$)

Linear memory model. A language of "consistent" traces:

$$\begin{aligned} M(\mu : \mathcal{V} \to \mathbb{N}) &::= \epsilon \\ &\mid \tau : \mathtt{R}_{x=\mu(x)} \cdot M(\mu) \\ &\mid \tau : \mathtt{W}_{x:=k} \cdot M(\mu[x \leftarrow k]) \\ M &::= M(x \mapsto 0) \end{aligned}$$

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

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

- every trace of $[\![p]\!]$ ends with $\mathtt{R}_{x=1}$ or a $\mathtt{R}_{y=2}$.

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.

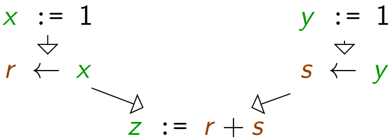
Term:

$$
\begin{aligned}
&x \;:=\; 1; y \;:=\; 1;\\
&r \;\leftarrow\; x; s \;\leftarrow\; y;\\
&z \;:=\; s + t
\end{aligned}
$$

Replacing traces by partial-orders

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

Dependencies (depends on the architecture):



$x := 1$

$r \leftarrow x$

$y := 1$

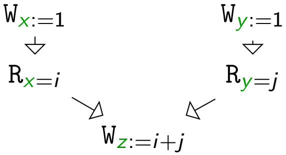
$s \leftarrow y$

$z := r + s$

Replacing traces by partial-orders

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

Executions (depends on the architecture):



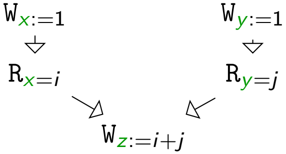
for $i, j \in \mathbb{N}^2$.

- traces on Σ_c becomes *partially ordered multisets* over Σ_c (pomsets)
- $[\![t]\!]^O$ 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):



$\mathtt{W}_{x:=1}$ $\mathtt{W}_{y:=1}$

$\mathtt{R}_{x=i}$ $\mathtt{R}_{y=j}$

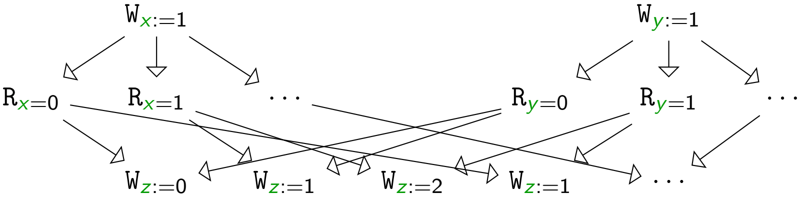
$\mathtt{W}_{z:=i+j}$

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

- traces on Σ_c becomes *partially ordered multisets* over Σ_c (pomsets)
- $[\![t]\!]^O$ 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:

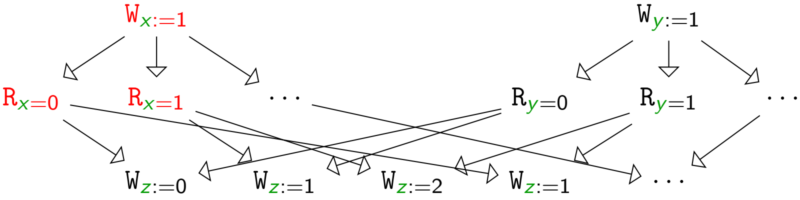


Which sets of events w are (partial) executions?

- 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:

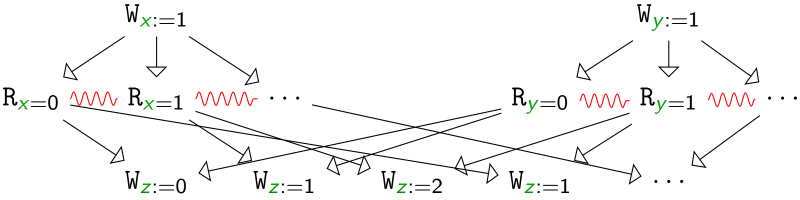


Which sets of events *w* are (partial) executions?

- *w* must be downward-closed for $\rightarrowtriangle$
- and ...? $\{\mathtt{W}_{x:=1}, \mathtt{R}_{x=0}, \mathtt{R}_{x=1}\}$ 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:



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

- *w* must be downward-closed for $\rightarrowtriangle$
- and …? $\{\mathtt{W}_{x:=1}, \mathtt{R}_{x=0}, \mathtt{R}_{x=1}\}$ cannot be a valid execution.

$\Rightarrow$ 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* $\sim\!\!\sim_E$
- A labelling $l : E \to \Sigma$.

(+ axioms)

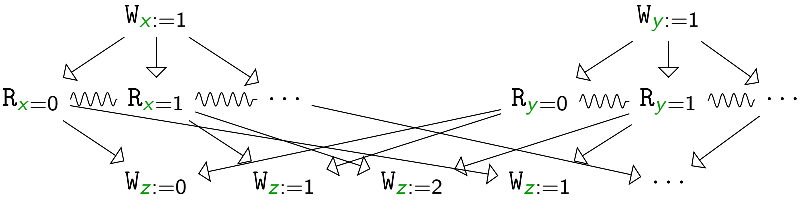
Definition (Configuration or partial execution)

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

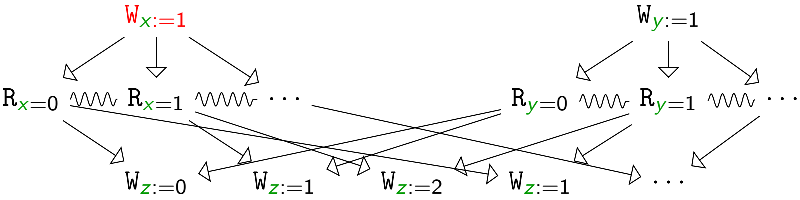
Event structures save the day

On the example:



Event structures save the day

On the example:

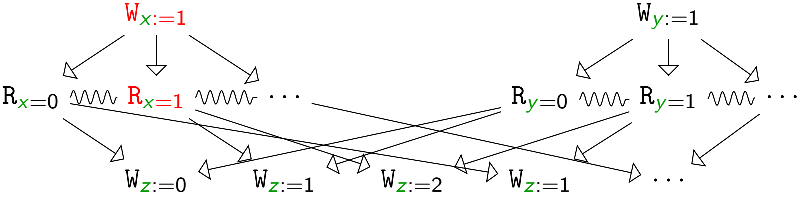


We have the configuration:

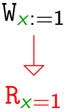
$\mathtt{W}_{x:=1}$

Event structures save the day

On the example:

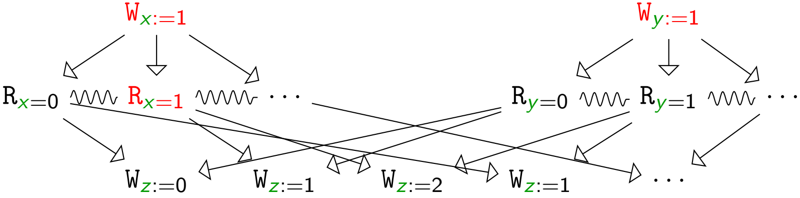


We have the configuration:



Event structures save the day

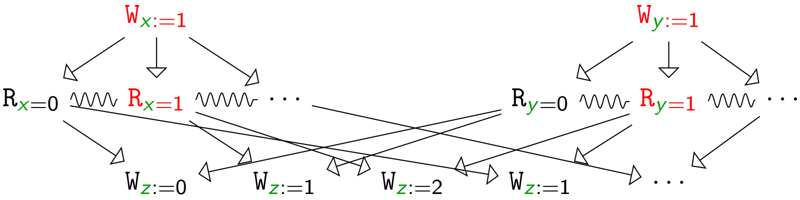
On the example:



We have the configuration:

Event structures save the day

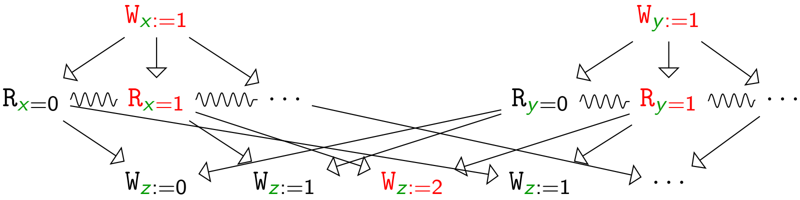
On the example:



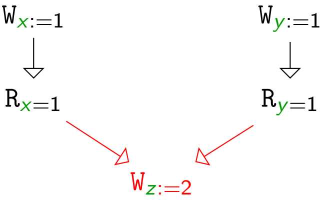
We have the configuration:

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.

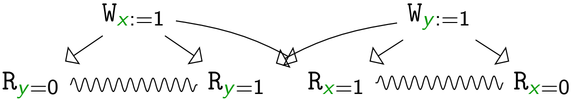
Two steps:

- *Open semantics:* $[\![t]\!]^O$ is an event structure
- *Closed semantics:* $[\![t]\!] = [\![t]\!]^O \wedge \mathscr{M}_{\mathtt{TSO}}$

Store buffering:

$$\begin{array}{c} x, y \text{ initialized to } 0 \\ \begin{array}{ccc||ccc} x & := & 1 & y & := & 1 \\ r & \leftarrow & y & s & \leftarrow & x \end{array} \end{array}$$

becomes:

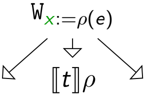


Thread semantics

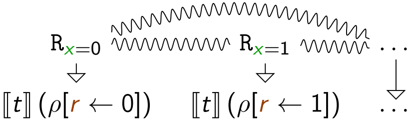
By induction as before, generalizing operations to event structures.

Threads: (omitting thread-ids)

$$\mathscr{T}[\![x := e; t]\!]\rho = \mathtt{W}_{x:=\rho(e)} \cdot [\![t]\!]\rho$$



$$\mathscr{T}[\![r \leftarrow x; t]\!]\rho = \sum_{i\in\mathbb{N}} \mathtt{R}_{x=i} \cdot [\![t]\!]\,(\rho[r \leftarrow i])$$



Programs:

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

$$[\![t_1]\!]\emptyset \qquad \ldots \qquad [\![t_n]\!]\emptyset$$

Consistent memory behaviours

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

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

$$\begin{tikzcd}
\mathtt{W}_{x:=1} \arrow[r] & \mathtt{W}_{x:=3} \arrow[r] & \mathtt{W}_{x:=4} \\
\mathtt{W}_{y:=2} \arrow[r] & \mathtt{W}_{y:=0} \arrow[ur] &
\end{tikzcd}$$

2. **Coherent reading.** For $e = \mathtt{R}_{x=k} \in q$, $\mathtt{W}_{x:=k}$ is the maximal event of $\{\mathtt{W}_{x:=n} \in q \mid \mathtt{W}_{x:=n} \leq e\}$

$$\begin{tikzcd}
 & \mathtt{W}_{y:=2} \arrow[rr] & & \\
\mathtt{W}_{x:=2} \arrow[r] & \mathtt{W}_{x:=3} \arrow[r] & \mathtt{R}_{y=0} \arrow[r] & \mathtt{R}_{x=3}
\end{tikzcd}$$

3. **Writes propagation.** For all writes $w \in q$, and for all *incomparable* reads $r, r' \in q$ in a different thread than w, $(w \leq r)$ iff $(w \leq r')$
4. **Thread sequentialization** Two events from the same thread are comparable [unless it is an independent read & write pair].

$\mathscr{M}_{\texttt{TSO}}$ and the synchronized product

Theorem

There exists an event structure $\mathscr{M}_{\texttt{TSO}}$ whose configurations are exactly consistent TSO-execution.

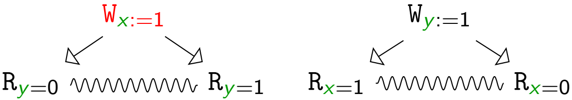
(Relies on TSO execution being closed under “prefix”)

How to combine $\mathscr{T}[\![t]\!]$ and $\mathscr{M}_{\texttt{TSO}}$? Using the **synchronized product**:

$$[\![t]\!] = \mathscr{T}[\![t]\!] \wedge \mathscr{M}_{\texttt{TSO}}.$$

Example

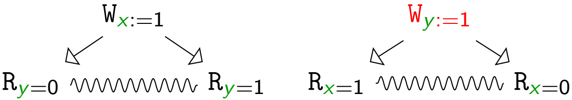
$$p = \begin{array}{lcl} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{lcl} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Thread semantics)

Example

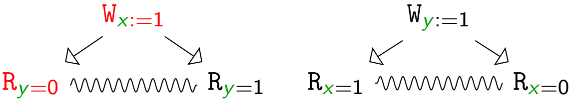
$$p = \begin{array}{lll} x & := & 1 \\ r & \leftarrow & y \end{array} \,\Big\|\, \begin{array}{lll} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

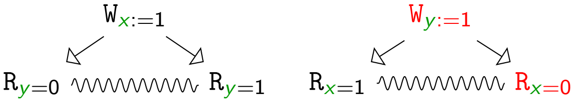
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \Bigg\| \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

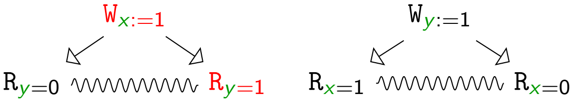
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

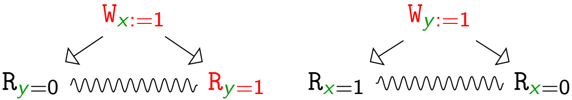
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \Bigg\| \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



$\mathtt{W}_{x:=1}$

$\mathtt{R}_{y=0}$

$\mathtt{R}_{y=1}$

$\mathtt{W}_{y:=1}$

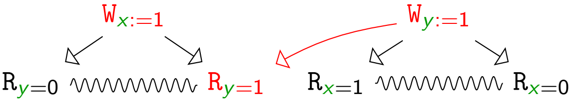
$\mathtt{R}_{x=1}$

$\mathtt{R}_{x=0}$

(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

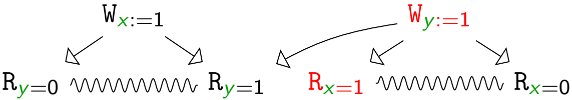
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

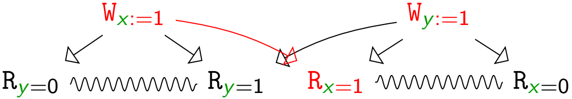
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \Bigg\| \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

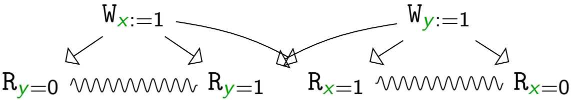
$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Big\|\; \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

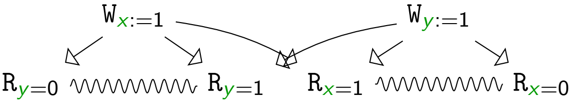
$$p = \begin{array}{lll|lll} x & := & 1 & y & := & 1 \\ r & \leftarrow & y & s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

Example

$$p = \begin{array}{ccc} x & := & 1 \\ r & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{ccc} y & := & 1 \\ s & \leftarrow & x \end{array}$$



(Computing $\mathscr{T}[\![p]\!] \wedge \mathscr{M}_{\mathtt{TSO}}$)

We can observe $r = 0 \wedge s = 0$.

Link with operational semantics

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.

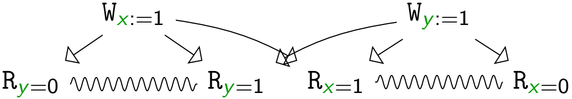
However, there are no *explicit* buffers in our semantics.

Implicitly represented by *concurrency*: If $\mathtt{R}_{x=k}$ is concurrent to $\mathtt{W}_{x:=k'}$, the read does not see the write.

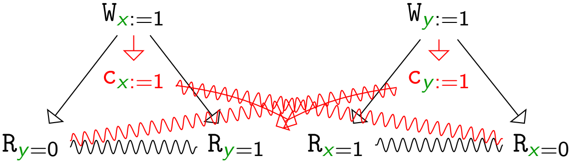
Implicit vs. explicit

Our model is *implicit*: no internal events

*Implicit semantics of **SB**.*



*Explicit semantics of **SB**.*



Reordering

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

This changes the *thread semantics*.

$\mathscr{T}[\![\mathsf{SB}]\!]^{O}$ becomes:

$$\mathtt{W}_{x:=1} \qquad \mathtt{R}_{y=0} \leadsto \mathtt{R}_{y=1} \qquad\qquad \mathtt{W}_{y:=1} \qquad \mathtt{R}_{x=0} \leadsto \mathtt{R}_{x=1}$$

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:

$$[\![r \leftarrow x; t]\!]\rho = \texttt{let} \odot \langle x, [\![\lambda r.t]\!] \rangle$$

where:

- $\odot$: strategy composition
- `let` is a carefully-written strategy.

A strategy for `read`

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

$$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$$

A strategy for `read`

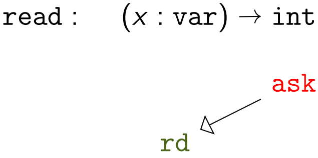
Usually, reads are interpreted by a strategy $\texttt{read} : \texttt{var} \to \texttt{unit}$:

$$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$$

$$\texttt{ask}$$

A strategy for `read`

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

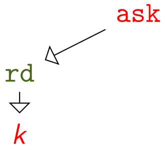


$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$

A strategy for `read`

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

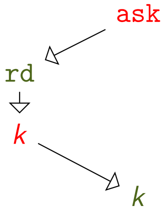
$$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$$



A strategy for `read`

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

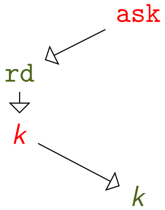
$$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$$



A strategy for `read`

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

$$\texttt{read} : \quad (x : \texttt{var}) \to \texttt{int}$$



Problem. No access to the continuation to break causalities.

Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$

Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$

`run`

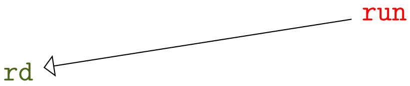
Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$



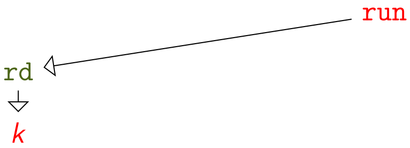
Interpreting let

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

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



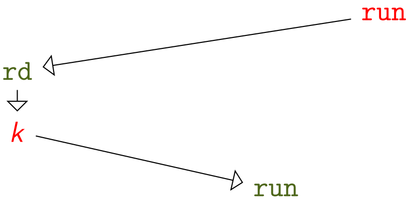
Interpreting let

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

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



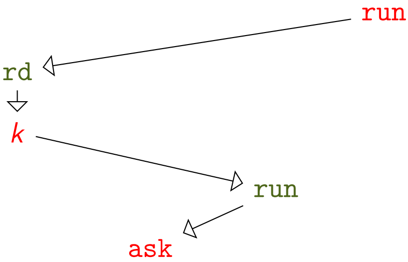
Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$



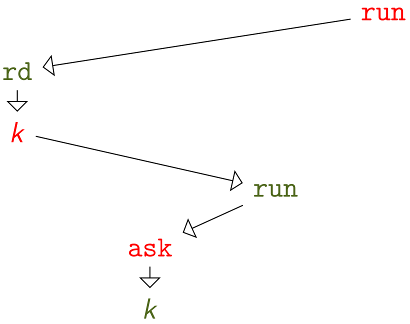
Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$



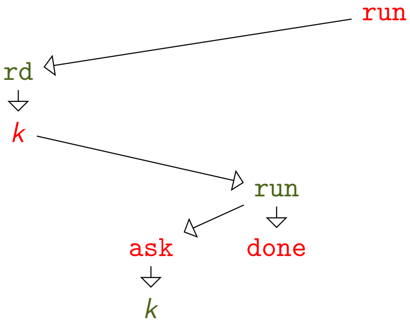
Interpreting let

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

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



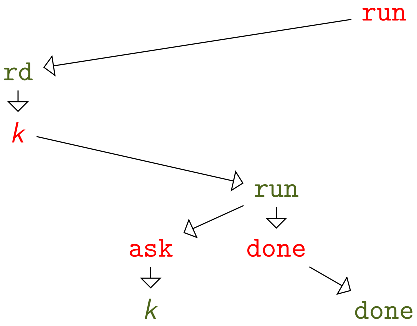
Interpreting let

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

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

This gives the following strategy:

$$x : \texttt{var} \to f : (\texttt{int} \to \texttt{unit}) \to \texttt{unit}$$



run

rd

k

run

ask

done

k

done

Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$

Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$

run

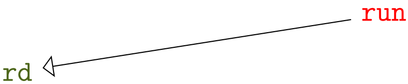
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



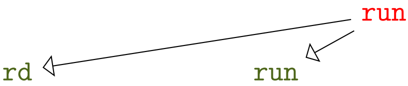
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \rightarrow f : (\mathtt{int} \rightarrow \mathtt{unit}) \rightarrow \mathtt{unit}$$



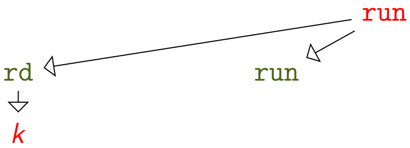
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



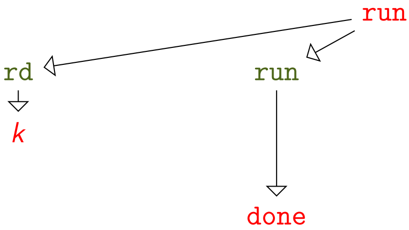
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



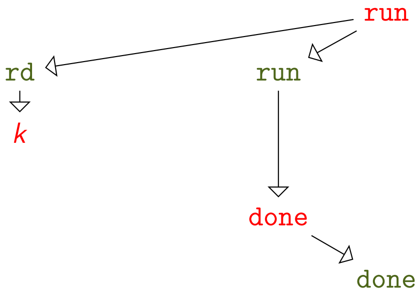
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



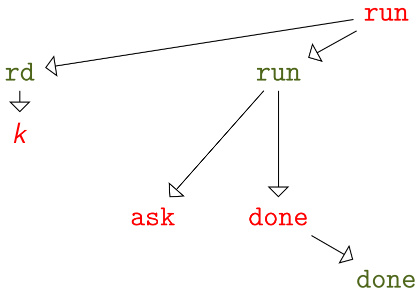
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



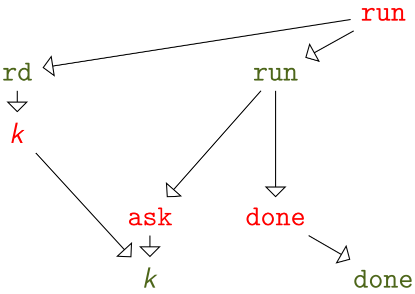
Adding concurrency in the mix

This type support more interesting definitions of `let`:

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

This gives the following strategy:

$$x : \mathtt{var} \to f : (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$$



Example

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

$$x : \texttt{var} \to y : \texttt{var} \longrightarrow \texttt{int}$$

Example

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

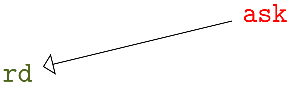
$$x : \texttt{var} \to y : \texttt{var} \longrightarrow \texttt{int}$$

$$\qquad\qquad\qquad\qquad\texttt{ask}$$

Example

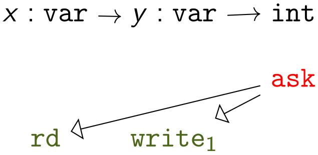
Consider $t(x, y) = \mathtt{let}\ x\ (\lambda n.\ \mathtt{write}\ y\ 1; n + 1)$:

$$x : \mathtt{var} \to y : \mathtt{var} \longrightarrow \mathtt{int}$$



Example

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



$x : \texttt{var} \rightarrow y : \texttt{var} \longrightarrow \texttt{int}$

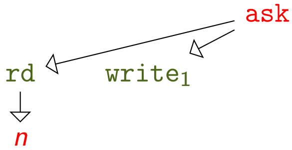
ask

rd write$_1$

Example

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

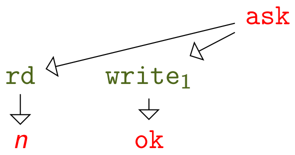
$$x : \texttt{var} \rightarrow y : \texttt{var} \longrightarrow \texttt{int}$$



Example

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

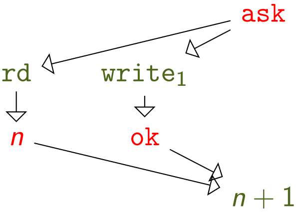
$$x : \texttt{var} \to y : \texttt{var} \longrightarrow \texttt{int}$$



Example

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

$$x : \texttt{var} \to y : \texttt{var} \longrightarrow \texttt{int}$$



A new model

Thread semantics. We use strategies `let` and `write`:

$$\mathscr{T}[\![\mathtt{let}\,x\,f]\!] = \mathtt{let} \odot \langle x, f\rangle$$
$$\mathscr{T}[\![x := k]\!] = \mathtt{write} \odot \langle x, k\rangle$$

No more ad-hoc inductive constructions: all is contained in the strategies `let` and `write`.
From $\Gamma \vdash t : A$, we get $\mathscr{T}[\![t]\!] : [\![\Gamma]\!] \Rightarrow [\![A]\!]$.

Storage semantics. $\mathscr{M}_{\mathtt{TSO}}$ induces a strategy on $\mathfrak{m}_{\mathtt{TSO}} : [\![\mathtt{var}]\!]^n$.

Semantics. For $x_1 : \mathtt{var}, \ldots, x_n : \mathtt{var} \vdash t : \mathtt{unit}$:

$$[\![t]\!] = \mathscr{T}[\![t]\!] \odot \mathfrak{m}_{\mathtt{TSO}}$$

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