Weak memory models using event structures

Simon Castellan[1]

[1]LIP, ENS Lyon

March 25, 2016
Gallium Seminar

Unexpected behaviours

A simple concurrent and imperative program:

$$\begin{array}{c} x, y \text{ initialized to } 0 \\ \begin{array}{rcl} x & := & 1 \\ r & \leftarrow & y \end{array} \Bigg\| \begin{array}{rcl} y & := & 2 \\ s & \leftarrow & x \end{array} \end{array}$$

shared variable · local register

Expected outcome: $r \neq 0 \vee s \neq 0$.

Unexpected behaviours

A simple concurrent and imperative program:

$$\begin{array}{c} x, y \text{ initialized to } 0 \\ \begin{array}{ccc|c|ccc} r & \leftarrow & y & & s & \leftarrow & x \\ x & := & 1 & & y & := & 2 \end{array} \\ \text{shared variable} \cdot \text{local register} \end{array}$$

Expected outcome: $r \neq 0 \vee s \neq 0$.
Wrong on modern architectures (x86, ARM, ...).

Unexpected behaviours

Another simple program:

$$\begin{array}{ccc||ccc} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$

Expected outcome: $r_1 = s_1 = 1 \Rightarrow r_2 = s_2 = 1$

Unexpected behaviours

Another simple program:

$$\begin{array}{rcl|rcl} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$

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*:

Program	Some executions
$x \ := \ 1 \ \Vert \ y \ := \ 2$ $r \leftarrow y \ \Vert \ s \leftarrow x$	$\mathtt{W}_x^{(1)} \cdot \mathtt{W}_y^{(2)} \cdot \mathtt{R}_y^{(2)} \cdot \mathtt{R}_x^{(1)}$ $\mathtt{W}_y^{(2)} \cdot \mathtt{R}_x^{(0)} \cdot \mathtt{W}_x^{(2)} \cdot \mathtt{R}_y^{(1)}$

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:

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

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.

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

Notations: $\mathtt{R}_x^{(k)}$, $\mathtt{W}_x^{(k)}$.

Two steps:

1. Volatile 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.

Volatile semantics

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

$$\begin{aligned}
&\text{(Writes)} \quad [\![x \ \texttt{:=} \ e; t]\!]\rho = \mathtt{W}_x^{(\rho(e))} \cdot [\![t]\!]\rho \\
&\text{(Reads)} \quad [\![r \ \leftarrow \ x; t]\!]\rho = \bigcup_{i \in \mathbb{N}} \left(\mathtt{R}_x^{(i)} \cdot [\![t]\!](\rho[r \leftarrow i])\right)
\end{aligned}$$

Volatile semantics

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

$$
\begin{array}{ll}
\text{(Writes)} & [\![x \ \texttt{:=} \ e; t]\!]\rho = \mathtt{W}_{x}^{(\rho(e))} \cdot [\![t]\!]\rho \\
\text{(Reads)} & [\![r \ \leftarrow \ x; t]\!]\rho = \bigcup_{i \in \mathbb{N}} \left(\mathtt{R}_{x}^{(i)} \cdot [\![t]\!](\rho[r \leftarrow i]) \right)
\end{array}
$$

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

$$[\![t_1 \parallel \ldots \parallel t_n]\!] = [\![t_1]\!]\emptyset \circledast \ldots \circledast [\![t_n]\!]\emptyset$$

Volatile semantics

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

$$\begin{array}{rl}\text{(Writes)} & [\![x \;\mathtt{:=}\; e; t]\!] \rho = \mathtt{W}_x^{(\rho(e))} \cdot [\![t]\!] \rho \\ \text{(Reads)} & [\![r \leftarrow x; t]\!] \rho = \bigcup_{i \in \mathbb{N}} \left(\mathtt{R}_x^{(i)} \cdot [\![t]\!] (\rho[r \leftarrow i]) \right)\end{array}$$

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

$$[\![t_1 \parallel \ldots \parallel t_n]\!] = [\![t_1]\!] \emptyset \circledast \ldots \circledast [\![t_n]\!] \emptyset$$

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

- $\mathtt{W}_x^{(1)} \cdot \mathtt{W}_y^{(1)} \cdot \mathtt{R}_y^{(3)} \cdot \mathtt{R}_x^{(2)} \in [\![p]\!]$
- 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:

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

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

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

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

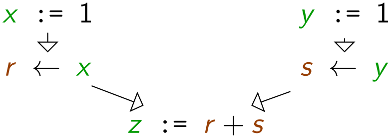
Term:

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

Replacing traces by partial-orders

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

Executions (depends on the architecture):

$$\begin{tikzcd}
\mathtt{W}_x^{(1)} \arrow[d] & & \mathtt{W}_y^{(1)} \arrow[d] \\
\mathtt{R}_x^{(i)} \arrow[dr] & & \mathtt{R}_y^{(j)} \arrow[dl] \\
& \mathtt{W}_z^{(i+j)} &
\end{tikzcd}$$

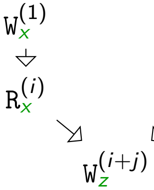
for $i, j \in \mathbb{N}^2$.

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

Executions (depends on the architecture):

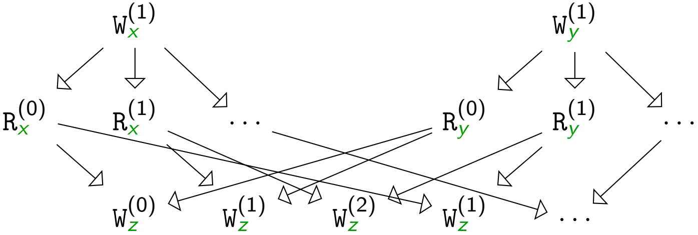


for $i, j \in \mathbb{N}^2$.

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

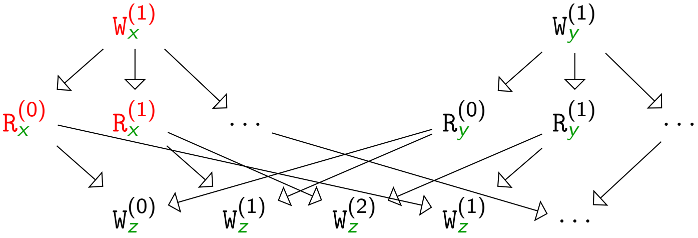


Which sets of events *w* are (partial) executions?

- *w* must be downward-closed for $\rightarrowtriangle$

Can we sum up *all* executions in a single object?

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

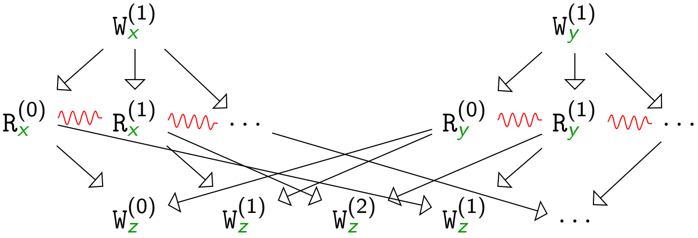


Which sets of events *w* are (partial) executions?

- *w* must be downward-closed for $\rightarrow$
- 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.

⇒ 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 $l : E \to \Sigma$.

(+ axioms)

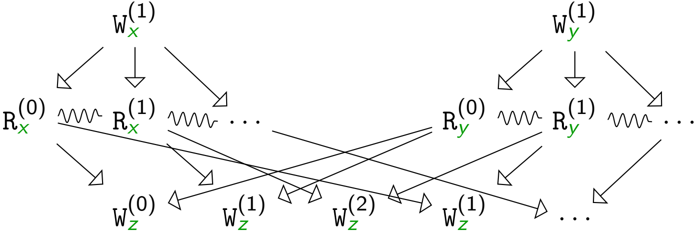
Definition (Configuration or partial execution)

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

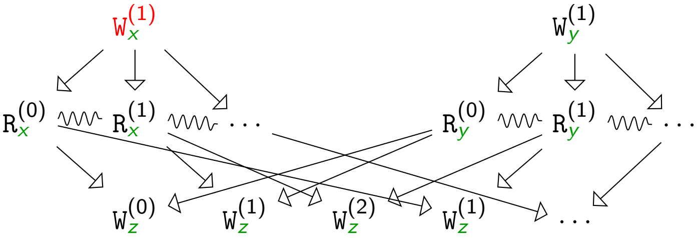
Event structures save the day

On the example:



Event structures save the day

On the example:

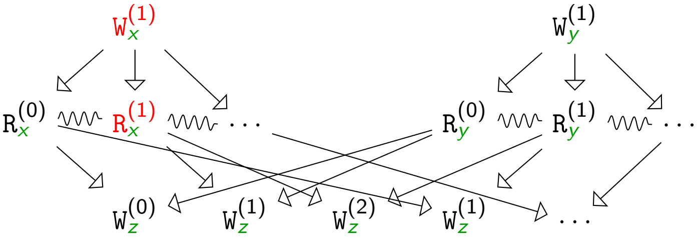


We have the configuration:

$$\mathtt{W}_x^{(1)}$$

Event structures save the day

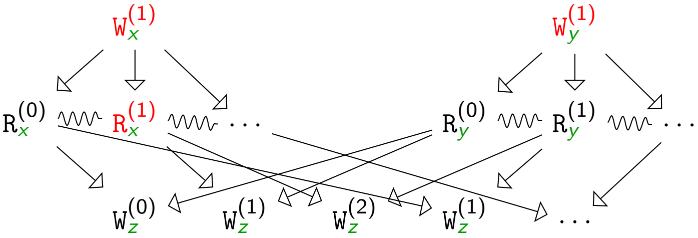
On the example:



We have the configuration:

Event structures save the day

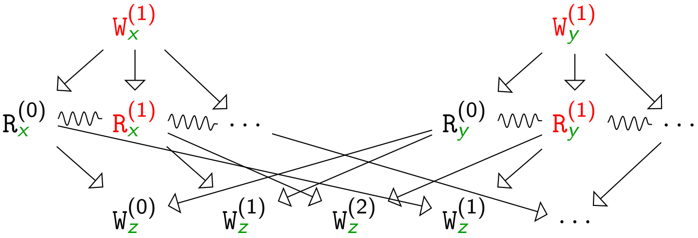
On the example:



We have the configuration:

Event structures save the day

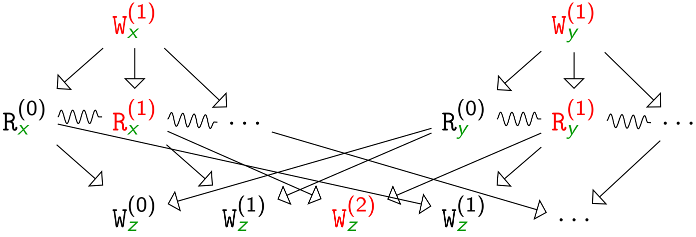
On the example:



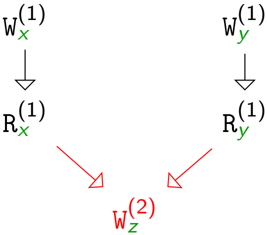
We have the configuration:

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

Defining architectures

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

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

Examples for $\rightarrowtriangle_{\mathscr{A}}$:

- $\rightarrowtriangle_{\mathtt{SC}} = \Sigma_a \times \Sigma_a$
- $\rightarrowtriangle_{\mathtt{ARM}} = \{(e, e') \mid \mathtt{v}(e) = \mathtt{v}(e')\}$ $(\mathtt{v}(x, \_) = x)$.
- $\rightarrowtriangle_{\mathtt{x86}} = \dots$

Examples for $E_{\mathscr{A}}$ include all languages $M \subseteq \Sigma_c^*$ (they can be viewed as event structures).

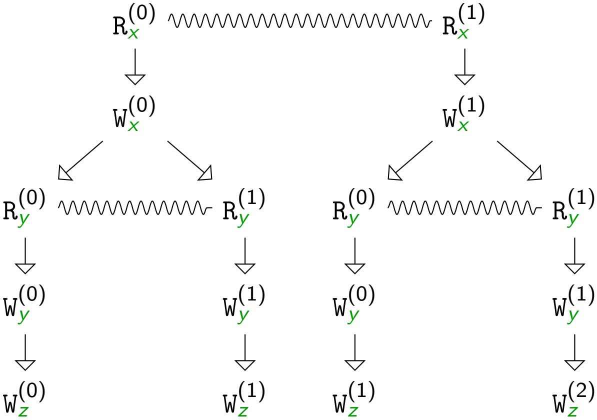
Computing the semantics $[\![p]\!]_{\mathscr{A}}$

As previously, in two steps:

- **Volatile semantics**:
  - *threads*: $[\![t]\!]^{O}_{\mathscr{A}}$ is defined as previously but where the causality outside $\rightharpoonup_{\mathscr{A}}$ are relaxed.
  - *programs*: $[\![t_1 \parallel \ldots \parallel t_n]\!]^{O}_{\mathscr{A}} = [\![t_1]\!]^{O}_{\mathscr{A}} \parallel \ldots \parallel [\![t_n]\!]^{O}_{\mathscr{A}}$ where $\parallel$ is *parallel composition*.
- **Closed semantics**: $[\![p]\!]_{\mathscr{A}} = [\![p]\!]^{O}_{\mathscr{A}} \wedge E_{\mathscr{A}}$ where $\wedge$ is the *synchronized product*: a generalization of intersection of languages to event structures.

Volatile semantics

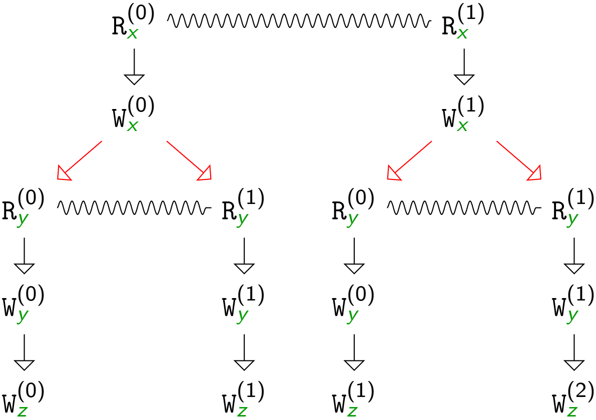
Pour $t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}$, on a:



(SC)

Volatile semantics

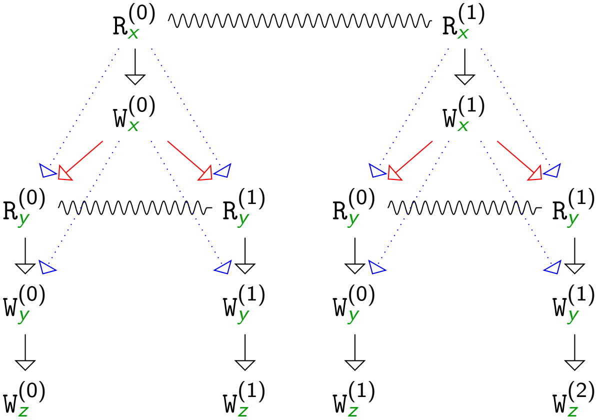
Pour $t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}$, on a:



(SC)

Volatile semantics

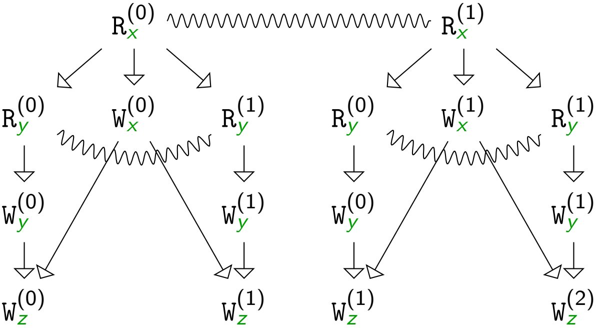
Pour $t = \begin{pmatrix} s & \leftarrow & x; x & := & s; \\ t & \leftarrow & y; y & := & t; \\ z & := & s + t & & \end{pmatrix}$, on a:



(x86)

Volatile semantics

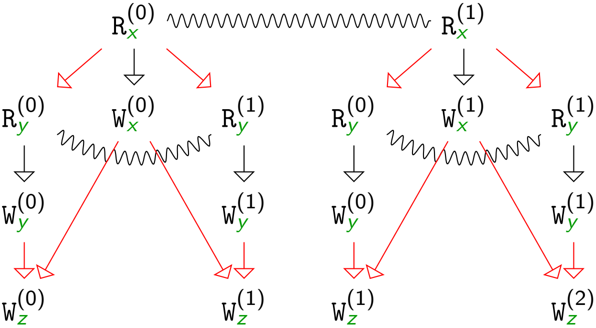
Pour $t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}$, on a:



(x86)

Volatile semantics

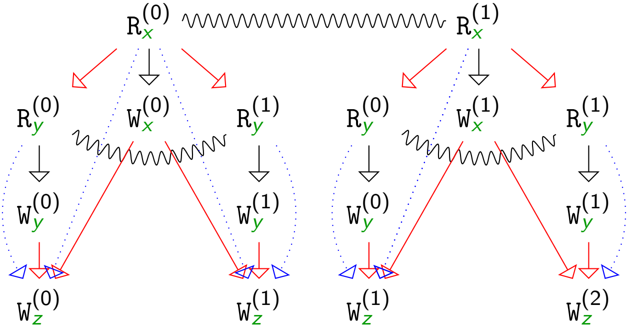
$$\text{Pour } t = \begin{pmatrix} s \leftarrow x; x := s; \\ t \leftarrow y; y := t; \\ z := s + t \end{pmatrix}, \text{ on a:}$$



(ARM)

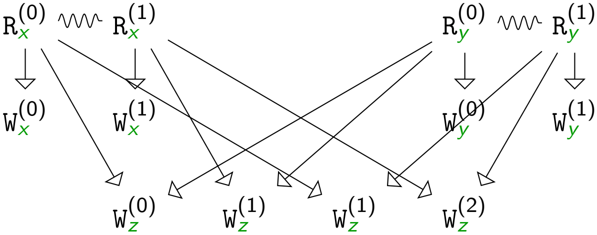
Volatile semantics

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



Volatile semantics

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



The memory model $\mathscr{E}$

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

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

$$\begin{tikzcd}
\mathtt{W}_x^{(1)} \arrow[r] & \mathtt{W}_x^{(3)} \arrow[r] & \mathtt{W}_x^{(4)} \\
\mathtt{W}_y^{(2)} \arrow[r] & \mathtt{W}_y^{(0)} \arrow[ur] &
\end{tikzcd}$$

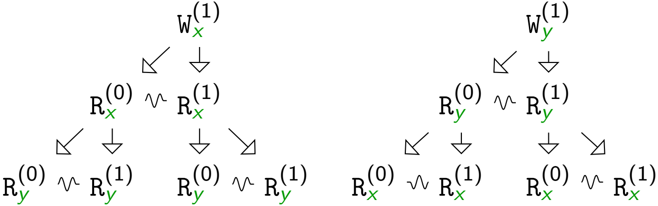
2. **Coherent reading.** For $e = \mathtt{R}_x^{(k)} \in q$, $\mathtt{W}_x^{(k)}$ is the maximal event of $\{\mathtt{W}_x^{(n)} \in q \mid \mathtt{W}_x^{(n)} \leq e\}$

$$\begin{tikzcd}
& \mathtt{W}_y^{(2)} \arrow[drr] & & \\
\mathtt{W}_x^{(2)} \arrow[r] & \mathtt{W}_x^{(3)} \arrow[r] & \mathtt{R}_y^{(0)} \arrow[r] & \mathtt{R}_x^{(3)}
\end{tikzcd}$$

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

Example

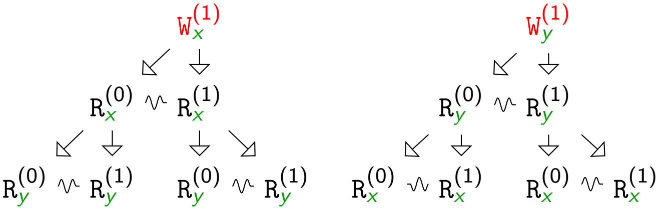
$$p = \begin{array}{rcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{rcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$



(**Volatile semantics** for SC)

Example

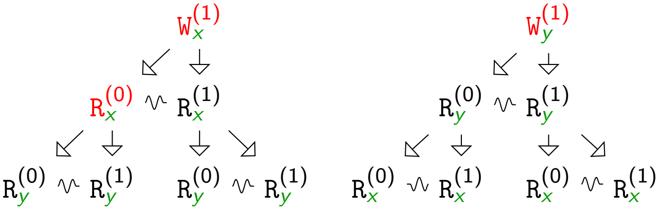
$$p = \begin{array}{rcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \Bigg\| \begin{array}{rcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

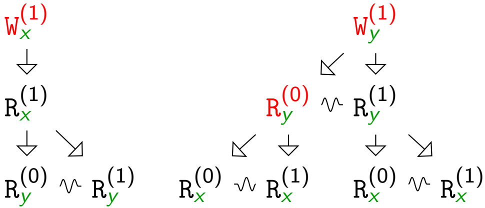
$$p = \begin{array}{rcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{rcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathrm{SC}} \wedge \mathscr{E}$)

Example

$$p = \begin{array}{rcl||rcl} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

$$p = \begin{array}{ccc|ccc} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$

$$\begin{tikzcd}
\mathtt{W}_x^{(1)} \arrow[d] & & & \mathtt{W}_y^{(1)} \arrow[d] & \\
\mathtt{R}_x^{(1)} \arrow[d] \arrow[dr] & & & \mathtt{R}_y^{(1)} \arrow[d] \arrow[dr] & \\
\mathtt{R}_y^{(0)} \arrow[r, no head, squiggly] & \mathtt{R}_y^{(1)} & & \mathtt{R}_x^{(0)} \arrow[r, no head, squiggly] & \mathtt{R}_x^{(1)}
\end{tikzcd}$$

(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

$$p = \begin{array}{rcl|rcl} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$

$$\begin{tikzcd}
\mathtt{W}_x^{(1)} \arrow[d] & & & \mathtt{W}_y^{(1)} \arrow[d] & \\
\mathtt{R}_x^{(1)} \arrow[d] \arrow[dr] & & & \mathtt{R}_y^{(1)} \arrow[d] \arrow[dr] & \\
\mathtt{R}_y^{(0)} \arrow[r, squiggly, no head] & \mathtt{R}_y^{(1)} & & \mathtt{R}_x^{(0)} \arrow[r, squiggly, no head] & \mathtt{R}_x^{(1)}
\end{tikzcd}$$

(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

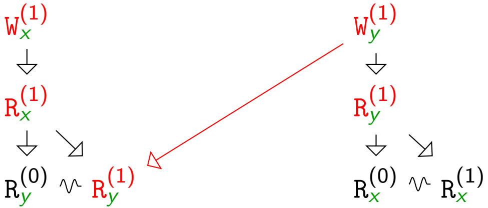
Example

$$p = \begin{array}{rcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{rcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$

(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

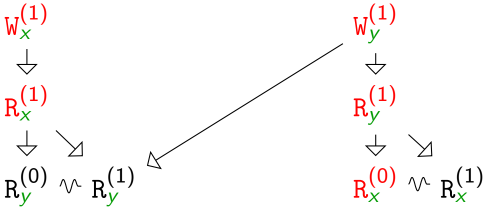
$$p = \begin{array}{rcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{rcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

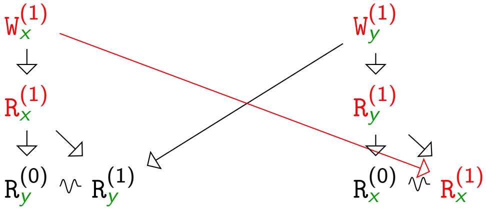
$$p = \begin{array}{lll||lll} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

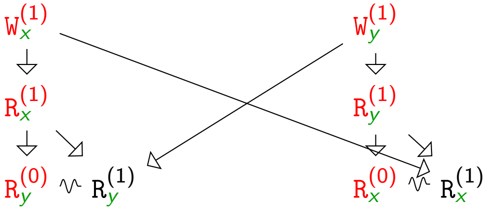
$$p = \begin{array}{ccc|ccc} x & := & 1 & y & := & 1 \\ r_1 & \leftarrow & x & s_1 & \leftarrow & y \\ r_2 & \leftarrow & y & s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

Example

$$p = \begin{array}{lcl} x & := & 1 \\ r_1 & \leftarrow & x \\ r_2 & \leftarrow & y \end{array} \;\Bigg\|\; \begin{array}{lcl} y & := & 1 \\ s_1 & \leftarrow & y \\ s_2 & \leftarrow & x \end{array}$$



(Computing $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$)

We can observe $r_1 = s_1 = 1 \wedge r_2 = s_2 = 0$.

$\mathscr{E}$ is too relaxed

Consider $p = \left(\begin{array}{ccc|ccc|ccc} x & := & 1 & r & \leftarrow & x & y & := & 1 \\ & & & s & \leftarrow & y & t & \leftarrow & x \end{array} \right)$

The denotation $[\![p]\!]^{O}_{\mathtt{SC}} \wedge \mathscr{E}$ contains the configuration:

$$\begin{tikzcd}
\mathtt{W}^{(1)}_{x} \arrow[r] & \mathtt{R}^{(1)}_{x} \arrow[d] & \mathtt{W}^{(1)}_{y} \arrow[d] \\
 & \mathtt{R}^{(0)}_{y} & \mathtt{R}^{(0)}_{x}
\end{tikzcd}$$

This allows the observation: $r = 1 \wedge s = t = 0$ 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 $\mathscr{E}_{TSO}$

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

$$\begin{tikzcd}
\mathtt{W}_{x}^{(1,1)} \arrow[r] & \mathtt{R}_{x}^{(2,1)} \arrow[d] & \mathtt{W}_{y}^{(3,1)} \arrow[d] \\
 & \mathtt{R}_{y}^{(2,0)} & \mathtt{R}_{x}^{(3,0)}
\end{tikzcd}$$

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 $\mathscr{E}_{TSO}$ 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**:

$$\begin{aligned} A, B &:= \mathtt{int} \mid \mathtt{var} \mid \mathtt{unit} \mid A \Rightarrow B \\ t, u &:= x \mid \lambda x.\ t \mid t\ u \\ &\quad\ \mid \mathtt{read}^{\mathtt{var} \to \mathtt{unit}} \mid \mathtt{write}^{\mathtt{var} \to \mathtt{int} \to \mathtt{unit}} \\ &\quad\ \mid \mathtt{new}\ \ x^{\mathtt{var}}\ \ \mathtt{in}\ \ t \qquad (t \text{ has type } \mathtt{int} \text{ or } \mathtt{unit}) \\ &\quad\ \mid (t;\ u) \mid (t \parallel u) \end{aligned}$$

- 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.

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

Problem. No access to the continuation to break causalities.

The usual strategy for read

An example.

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

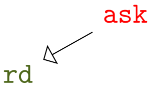
$$\texttt{ask}$$

Problem. No access to the continuation to break causalities.

The usual strategy for read

An example.

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



ask

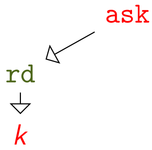
rd

Problem. No access to the continuation to break causalities.

The usual strategy for read

An example.

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

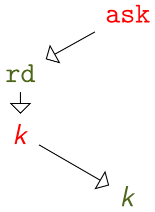


Problem. No access to the continuation to break causalities.

The usual strategy for read

An example.

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



Problem. No access to the continuation to break causalities.

Changing the type of read

The read operation becomes `let` : $\mathtt{var} \to (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$:

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

This gives the following strategy:

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

Changing the type of read

The read operation becomes `let` : $\mathtt{var} \to (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$:

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

This gives the following strategy:

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

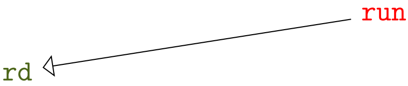
Changing the type of read

The read operation becomes let : var $\rightarrow$ (int $\rightarrow$ unit) $\rightarrow$ unit:

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

This gives the following strategy:

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



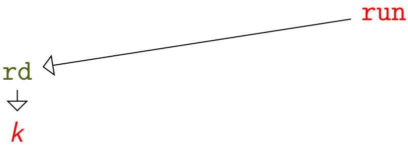
Changing the type of read

The read operation becomes `let` : var $\to$ (int $\to$ unit) $\to$ unit:

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

This gives the following strategy:

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



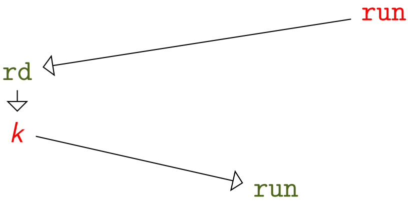
Changing the type of read

The read operation becomes `let` : $\mathtt{var} \to (\mathtt{int} \to \mathtt{unit}) \to \mathtt{unit}$:

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

This gives the following strategy:

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



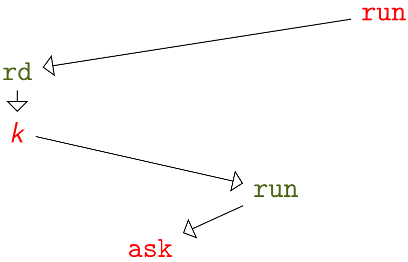
Changing the type of read

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

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

This gives the following strategy:

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



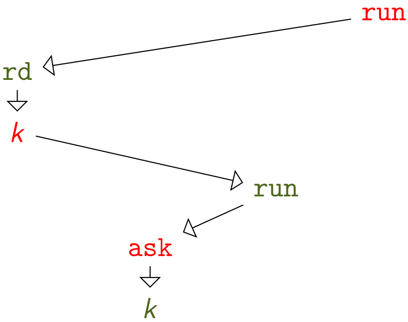
Changing the type of read

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

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

This gives the following strategy:

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



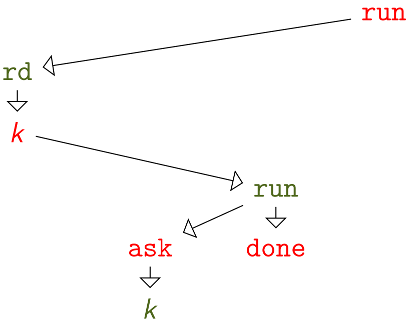
Changing the type of read

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

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

This gives the following strategy:

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



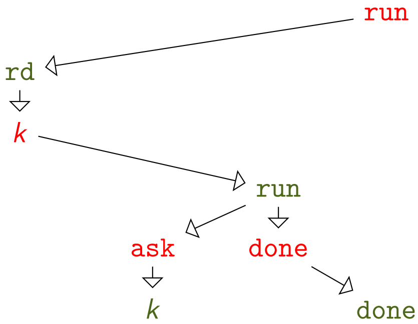
Changing the type of read

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

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

This gives the following strategy:

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



Adding concurrency in the mix

But we have space to make it more concurrent!

```
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

But we have space to make it more concurrent!

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

This gives the following strategy:

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

$$\texttt{run}$$

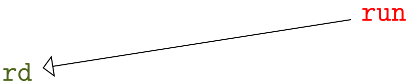
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



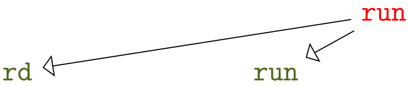
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



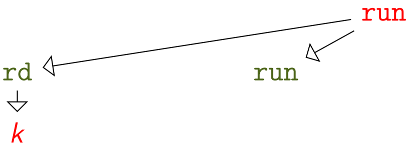
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



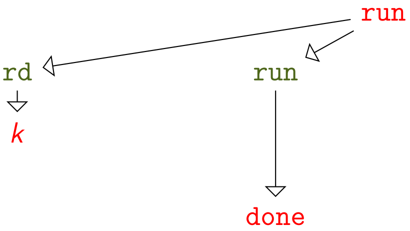
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



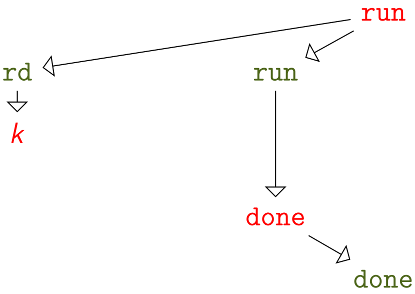
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



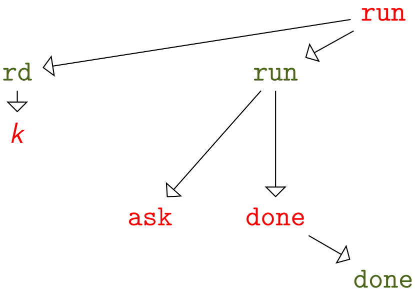
Adding concurrency in the mix

But we have space to make it more concurrent!

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

This gives the following strategy:

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



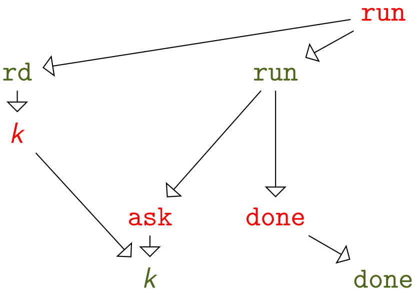
Adding concurrency in the mix

But we have space to make it more concurrent!

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

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

Example

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

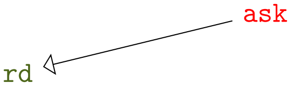
$$x : \mathtt{var} \to y : \mathtt{var} \longrightarrow \mathtt{int}$$

$$\phantom{x : \mathtt{var} \to y : \mathtt{var} \longrightarrow}\ \mathtt{ask}$$

Example

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

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



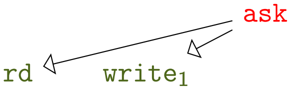
ask

rd

Example

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

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



ask

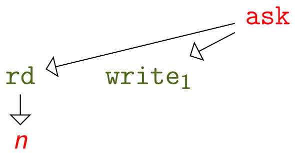
rd

write$_1$

Example

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

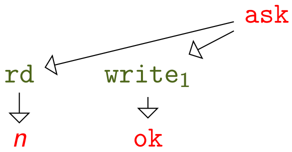
$$x : \texttt{var} \rightarrow y : \texttt{var} \longrightarrow \texttt{int}$$



Example

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

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



ask

rd

$\texttt{write}_1$

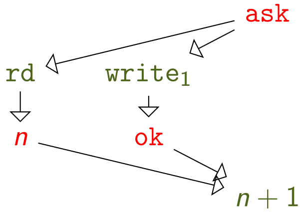
n

ok

Example

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

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



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 $\mathscr{E}$ 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?