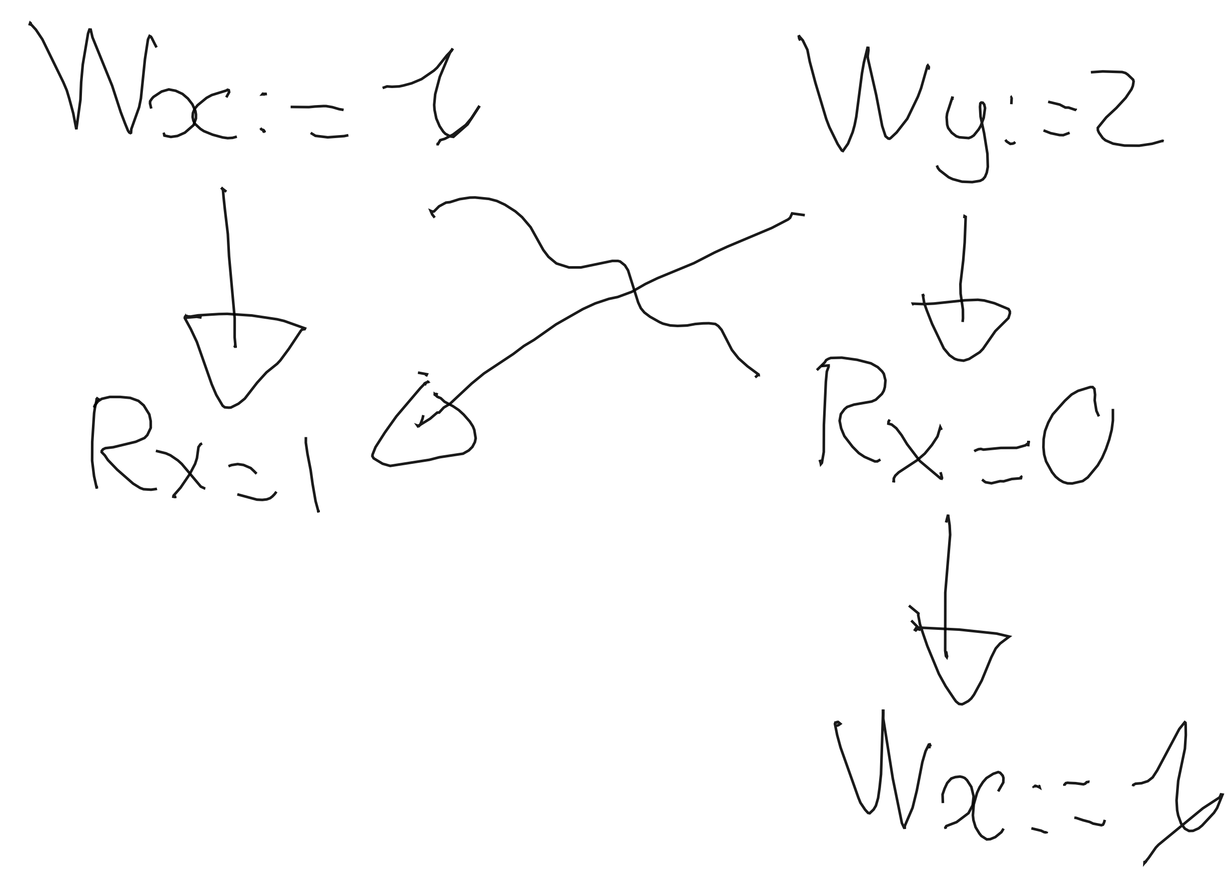
DESCRIBING EVENT STRUCTURES MODELS

Simon Castellan
Inria Rennes

Concurrent games café

From Programs to Event Structures

$$[\![\, x := 1 \parallel \begin{array}{l} y := 2 \\ \text{read } x \end{array}]\!] =$$

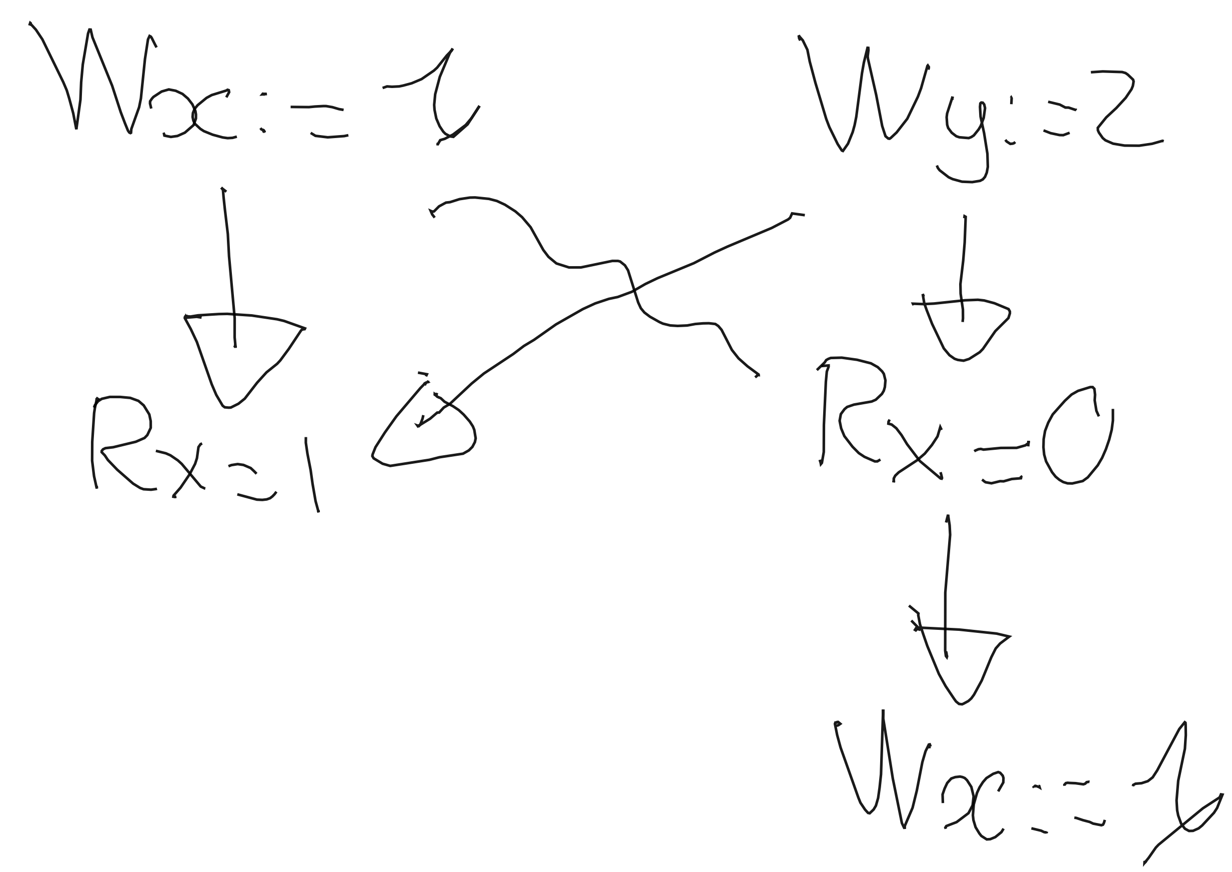


Methodology:

1. Define a suitable space of event structures (labelling, ...)
2. Interpret each language construct by a semantic operation
3. Write a paper

From Programs to Event Structures

$$\left[\!\left[\, x := 1 \;\|\; \begin{array}{l} y := 2 \\ \text{read}\ x \end{array} \right]\!\right] =$$



Methodology:

1. Define a suitable space of event structures (labelling, ...)
2. Interpret each language construct by a semantic operation
3. Write a paper
4. Get rejected: "It's too complicated, also what is a pullback"

From Programs to Event Structures

$$[\![x := 1 \parallel \begin{matrix} y := 2 \\ \text{read } x \end{matrix}]\!] = \begin{tikzcd} Wx := 1 \arrow[d] \arrow[dr, squiggly, dash] & Wy := 2 \arrow[d] \arrow[dl] \\ Rx = 1 & Rx = 0 \arrow[d] \\ & Wx := 1 \end{tikzcd}$$

Methodology:

1. Define a suitable space of event structures (labelling, ...)
2. Interpret each language construct by a semantic operation
3. Write a paper
4. Get rejected: "It's too complicated, also what is a pullback"

<u>Question</u>: do we need to know what a pullback is?

Composition: a priori, a posteriori [DHR'92, GT'12, LS'14, Jab'15]

Composition : a priori , a posteriori [DHR'92, GT'12, LS'14, Jab'15]

$$[\![t \| u]\!] \stackrel{\text{def}}{=} [\![t]\!] \wedge [\![u]\!]$$

Composition : a priori , a posteriori [DHR'92, GT'12, LS'14, Sab'15]

$[\![t \| u]\!] \stackrel{\text{def}}{=} [\![t]\!] \wedge [\![u]\!]$

Lemma. $[\![t \| u]\!] = [\![t]\!] \wedge [\![u]\!]$

Composition: a priori, a posteriori [DHR'92, GT'12, LS'14, Jab'15]

$[\![t \parallel u]\!] \stackrel{\text{def}}{=} [\![t]\!] \wedge [\![u]\!]$

Lemma. $[\![t \parallel u]\!] = [\![t]\!] \wedge [\![u]\!]$

Understanding $\parallel$ is necessary for:

Everything from the interpretation

Soundness of compositional reasoning

Composition : a priori , a posteriori [DHR'92, GT'12, LS'14, Jab'15]

$[\![t \| u]\!] \stackrel{\mathrm{def}}{=} [\![t]\!] \wedge [\![u]\!]$

Lemma. $[\![t \| u]\!] = [\![t]\!] \wedge [\![u]\!]$

Understanding $\|$ is necessary for:

Everything from the interpretation

Soundness of compositional reasoning

What could causal models a posteriori compositional look like?

Compose as early as possible

$\text{Language} \xrightarrow{\text{Interpretation with } \lambda} \text{D.E.S.}$

Compose as early as possible

$$
\begin{tikzcd}
\text{Language} \arrow[rr, "\text{Interpretation with } \wedge"] \arrow[dr, swap, "\text{Simpler interpretation with } \hat{\wedge}"] & & \text{D.E.S.} \\
& \text{Something} \arrow[ur, swap, "\text{Projection}"] &
\end{tikzcd}
$$

Theorem. $\text{project}\,(t \mathbin{\hat{\wedge}} u) = \text{project}\ t \wedge \text{project}\ u$

Compose as early as possible

$$
\begin{tikzcd}
\text{Language} \arrow[rr, "\text{Interpretation with } \wedge"] \arrow[dr, "\text{Simpler interpretation with } \hat{\wedge}"'] & & \text{D.E.S.} \\
& \text{Something} \arrow[ur, "\text{Projection}"'] &
\end{tikzcd}
$$

Theorem. $\text{project}\,(t \mathbin{\hat{\wedge}} u) = \text{project}\ t \wedge \text{project}\ u$

<u>Benefits</u>:
- simplicity
- implementability / performance (avoid computing $\wedge$)

Part I

Closure operators

Source $\dashrightarrow$ C.O. $\longrightarrow$ E.S.

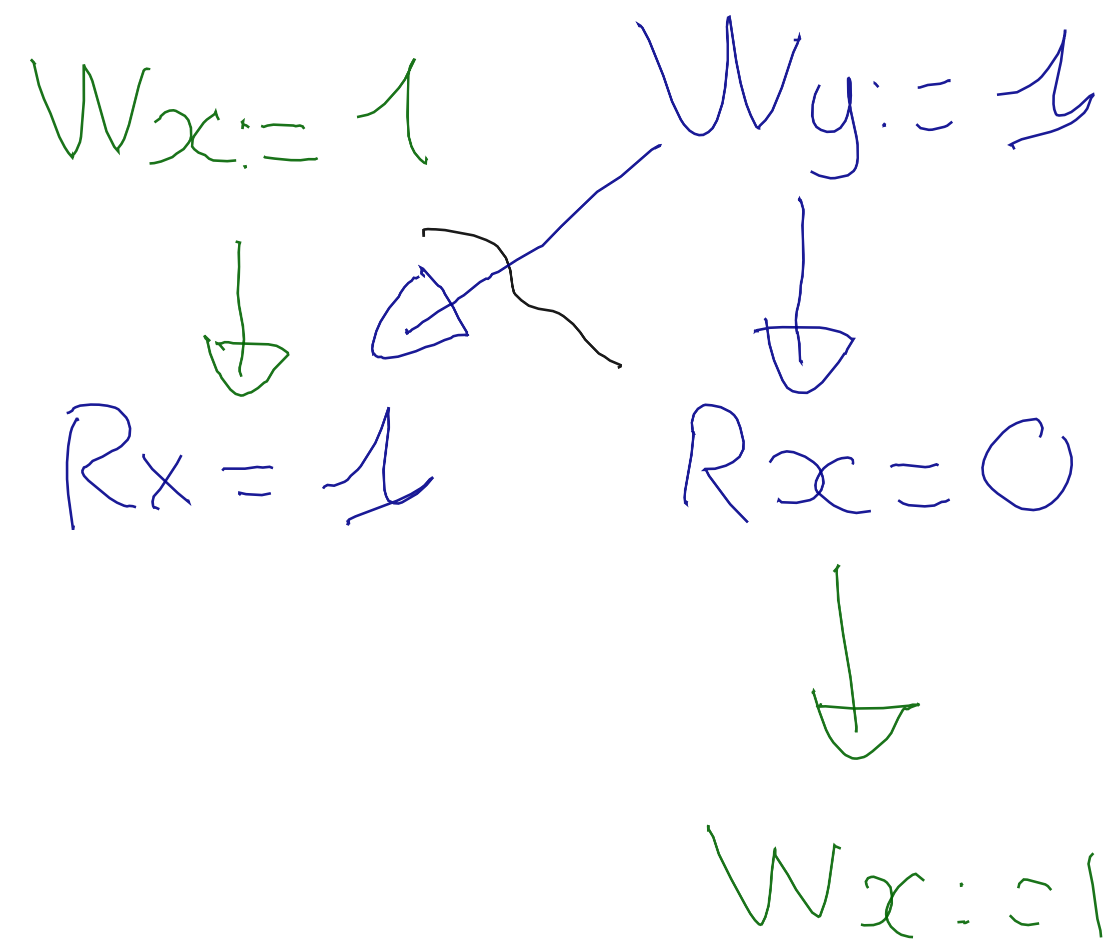
Causal models: why is it so hard?

$$\left[\!\!\left[\; x := 1 \;\Big\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right]\!\!\right] =$$

$$\begin{tikzcd}
Wx := 1 \arrow[d] & Wy := 1 \arrow[dl] \arrow[d] \\
Rx = 1 & Rx = 0 \arrow[d] \\
& Wx := 1
\end{tikzcd}$$

Causal models: why is it so hard?

$$\left[\!\left[\; x := 1 \;\Big\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right]\!\right] =$$



Standard interpretation:

pullback / product

$$[\![P \,\|\, Q]\!] = [\![P]\!] \,\|\, [\![Q]\!]$$

account for all possibilities

→ Alternative description?

Incrementality

$$\left[\; x := 1 \;\middle\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right] =$$

Incremtality

$$\left[\; x := 1 \;\middle\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right] = \quad W\, x := 1 \qquad \begin{array}{c} W\, y := 1 \\ \downarrow \\ R\, x = 0 \end{array}$$

Incremtality

$$\left[\!\left[\; x := 1 \;\middle\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right]\!\right] = \quad
\begin{tikzcd}
W x := 1 \arrow[d] & W y := 1 \arrow[d] \arrow[dl] \\
R x = 1 & R x = 0 \arrow[d] \\
 & W x := 1
\end{tikzcd}$$

Incremtality

$$\left[\!\!\left[\; x := 1 \;\middle\|\; \begin{array}{l} y := 1 \\ \mathrm{read}\ x \end{array} \;\right]\!\!\right] = \begin{tikzcd} W\, x := 1 \arrow[d] & W\, y := 1 \arrow[d] \arrow[dl] \\ R\, x = 1 & R\, x = 0 \arrow[d] \\ & W\, x := 1 \end{tikzcd}$$

Model programs by closure operators à la Abramsky-Melliès

$$[\![P]\!] : (ES, \subseteq) \to (ES, \subseteq) \text{ with}$$

$$\mathcal{E} \subseteq [\![P]\!](\mathcal{E}) \qquad [\![P]\!] \circ [\![P]\!] = [\![P]\!]$$

Incrementality

$$\left[\!\left[\; x := 1 \;\middle\|\; \begin{array}{l} y := 1 \\ \text{read } x \end{array} \;\right]\!\right] = \begin{tikzcd} Wx := 1 \arrow[d] & & Wy := 1 \arrow[dll] \arrow[d] \\ Rx = 1 & & Rx = 0 \arrow[d] \\ & & Wx := 1 \end{tikzcd}$$

Model programs by closure operators à la Abramsky-Melliès

$$[\![P]\!] : (ES, \subseteq) \to (ES, \subseteq) \text{ with}$$

$$\mathcal{E} \subseteq [\![P]\!](\mathcal{E}) \qquad [\![P]\!] \circ [\![P]\!] = [\![P]\!]$$

Example:

$$[\![x := 1]\!](\mathcal{E}) := \mathcal{E} \cup \{Wx := 1\} \bigcup_{\substack{o \in \mathcal{E} \\ var(o) = x}} \{Wx := 1\}$$

Alternative description of event Structures

$$(E, \leq_E, \#_E)$$

binary relations
not a local property
on events

Alternative description of event Structures

$$(E, \leq_E, \#_E) \rightsquigarrow \left(\begin{array}{l} E, \mathrm{view} : E \to \mathcal{P}_{fin}(E), \\ \quad \mathrm{world} : E \to W \end{array}\right)$$

binary relations
not a local property
on events

$(W, \leq)$ a partial order
of worlds (abstract configurations)

Alternative description of event Structures

$$(E, \leq_E, \#_E) \rightsquigarrow \left(E, \text{view} : E \to \mathcal{P}_{fin}(E), \; \text{world} : E \to W \right)$$

binary relations
not a local property
on events

$(W, \leq)$ a partial order
of worlds (abstract configurations)

$$(E, \leq_E, \#_E) \longmapsto (E, e \mapsto [e], e \mapsto [e])$$

$$(W = (\mathcal{C}(E), \subseteq))$$

Alternative description of event structures

$(E, \leq_E, \#_E) \rightsquigarrow \left(\begin{array}{l} E, \mathrm{view}: E \to \mathcal{P}_{fin}(E), \\ \mathrm{world}: E \to W \end{array} \right)$

binary relations
not a local property
on events

$(W, \leq)$ a partial order
of worlds (abstract configurations)

$(E, \leq_E, \#_E) \longmapsto (E, e \mapsto [e], e \mapsto [e])$

$(W = (\mathcal{C}(E), \subseteq))$

$(E, \mathrm{view}, \mathrm{world}) \longmapsto (E, \leq_E, \#_E)$

$e \leq_E e' \iff e \in \mathrm{view}(e')$

$e \#_E e' \iff \mathrm{world}(e) \vee \mathrm{world}(e')$ undefined

join in the partial order

Writing down closure operators

Consider $(W, \leq)$ a partial order of worlds

Our domain for closure operators

$$\mathrm{Event} ::= \mathrm{Label} \times \wp(\mathrm{Event}) \times W$$

$$D := \wp(\mathrm{Event}) \qquad \text{[there is a partial } D \to ES\text{]}$$

Writing down closure operators

Consider $(W, \leq)$ a partial order of worlds

Our domain for closure operators

$$\text{Event} ::= \text{Label} \times \mathcal{P}(\text{Event}) \times W$$

$$D := \mathcal{P}(\text{Event}) \qquad \text{[there is a partial } D \to ES\text{]}$$

With $W =$ set of memory traces on x ordered by prefix:

$$[\![x := 1]\!](\mathcal{E}) := \mathcal{E} \cup \{(W_x := 1, \emptyset, W_x := 1)\}$$

$$\cup \bigcup_{e = (O_x, v, w) \in \mathcal{E}} \{(W_x := 1, v \cup \{e\}, w \cdot W_x := 1)\}$$

Writing down closure operators

Consider $(W, \leq)$ a partial order of worlds

Our domain for closure operators

$$\text{Event} ::= \text{Label} \times \wp(\text{Event}) \times W$$

$$D ::= \wp(\text{Event})$$

[there is a partial $D \to ES$]

With W = set of memory traces on x ordered by prefix:

action

$$[\![x := 1]\!](\mathcal{E}) := \mathcal{E} \cup \{(W_{x:=1}, \emptyset, W_{x:=1})\}$$

$$\cup \bigcup_{e = (O_x, v, w) \in \mathcal{E}} \{(W_{x:=1}, v \cup \{e\}, w \cdot W_{x:=1})\}$$

reaction

updating the world

Increasing maps & closure operators

$f : \mathcal{P}(E) \to \mathcal{P}(G)$ is <u>increasing</u> when it is monotonic & $X \subseteq f(X)$ $\forall e \in E$

$\rightsquigarrow$ A <u>closure operator</u> f is an increasing map s.t $f \circ f = f$

<u>Lemma</u>. For each increasing $f : \mathcal{P}(E) \to \mathcal{P}(E)$ there exists a least closure operator f^∞ containing f

$$f^\infty(X) = \lim (X \subseteq f(X) \subseteq f^2(X) \subseteq \dots)$$

Following [AM'11], we define for f, g closure operators on $\mathcal{P}(E)$:

$$f \| g := (f \cup g)^\infty = (f \circ g)^\infty = (g \circ f)^\infty$$

Writing down closure operators (2)

In general:

$$[\![P]\!] : \overbrace{\underbrace{\text{View}}_{P(E)} \times \text{World}}^{\text{Local history when } P \text{ is invoked}} \to [D \to D] := \text{Comp}$$

What are the operations we use to describe such objects?

Writing down closure operators (2)

In general:

$$[\![P]\!] : \overbrace{\underbrace{View}_{\mathcal{P}(E)} \times World}^{\text{Local history when } P \text{ is invoked}} \longrightarrow [D \to D] := Comp$$

What are the operations we use to describe such objects?

1. Emitting an event: $emit\,(\ell \in Label) = \lambda(v, w).\ \lambda \mathcal{E}.\ \mathcal{E} \cup \{(\ell, v, w)\}$

Writing down closure operators (2)

In general:

$$[\![P]\!] : \overbrace{\underbrace{\mathit{View}}_{\mathcal{P}_{fin}(E)} \times \mathit{World}}^{\text{Local history when } P \text{ is invoked}} \longrightarrow [D \to D] := \mathit{Comp}$$

What are the operations we use to describe such objects?

1. Emitting an event: $\mathrm{emit}\ (\ell \in \mathit{Label}) = \lambda(v,w).\ \lambda \mathcal{E}.\ \mathcal{E} \cup \{(\ell, v, w)\}$

2. Inspecting existing events

$$\mathrm{inspect}\ (c : \mathit{Label} \to \mathit{Comp}) := \lambda(v,w).\ \Big(\lambda \mathcal{E}.\ \mathcal{E} \cup \bigcup_{\substack{(\ell, v', w') \\ w \vee w' = \overline{w}}} c\,\ell\,(v \cup v', \overline{w})(\mathcal{E})\Big)^{\infty}$$

Writing down closure operators (2)

In general:

$$[\![P]\!] : \underbrace{\overbrace{\text{View} \times \text{World}}^{\text{Local history when } P \text{ is invoked}}}_{\mathcal{P}_{fin}(E)} \longrightarrow [D \to D] := \text{Comp}$$

What are the operations we use to describe such objects?

1 Emitting an event: $\text{emit}\,(\ell \in \text{Label}) = \lambda(v, w).\ \lambda\mathcal{E}.\ \mathcal{E} \cup \{(\ell, v, w)\}$

2 Inspecting existing events:

$$\text{inspect}\,(c : \text{Label} \to \text{Comp}) := \lambda(v, w).\left(\lambda\mathcal{E}.\ \mathcal{E} \cup \bigcup_{\substack{(\ell, v', w') \\ w \vee w' = \overline{w}}} c\ell\,(v \cup v', \overline{w})(\mathcal{E})\right)^{\infty}$$

3 Manipulating the world: $(\text{set } w\,;\, f)\,(v, \_) = f\,(v, w)$

II. A process calculus

$$\text{Source} \dashrightarrow \text{Process Calculus} \xrightarrow{[\![\cdot]\!]} \text{C.O.} \longrightarrow \text{E.S.}$$

A process calculus inspired with the model.

Basic idea : emit $\rightarrow$ sending a message

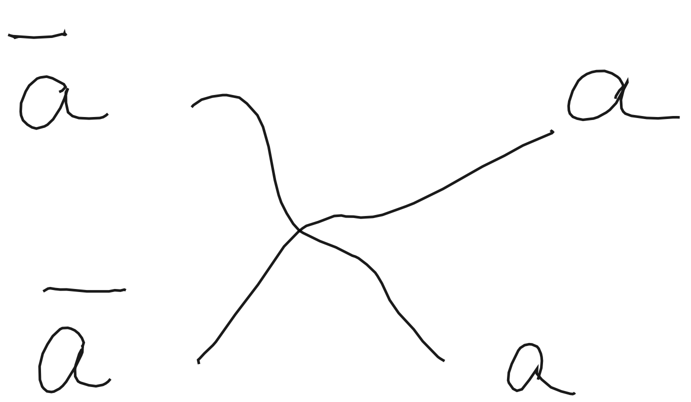
inspect $\rightarrow$ receiving a message

A process calculus inspired with the model.

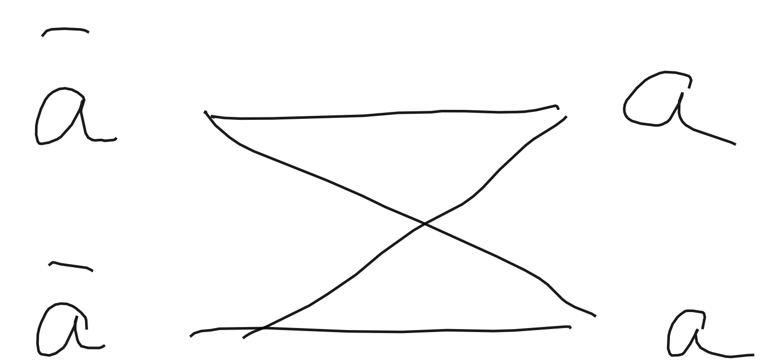
Basic idea: emit → sending a message

inspect → receiving a message

⚠ Unusual semantics: "deterministic channels"



CCS



Our model

Terminology: sending ⇒ put

receiving ⇒ watch

Syntax

Let $\mathbb{A}$ be a set of names, and $\gamma \in Env := \mathbb{A} \to Set$

We define by induction $\gamma \vdash P$

$$\frac{}{\gamma \vdash 0} \qquad \frac{\gamma \vdash P \quad \gamma \vdash Q}{\gamma \vdash P \parallel Q} \qquad \frac{\gamma, a : X \vdash P}{\gamma \vdash (\nu a : X) P}$$

$$\frac{v \in \gamma(a) \quad \gamma \vdash P}{\gamma \vdash put\ a\ v.\ P} \qquad \frac{\forall v \in \gamma(a) : \ \gamma \vdash P_v}{\gamma \vdash watch\ a\ (P_v)_{v \in \gamma(a)}}$$

$$\frac{\forall w \in W : \ \gamma \vdash P_w}{\gamma \vdash get.\ (P_w)_{w \in W}} \qquad \frac{\gamma \vdash P \quad w \in W}{\gamma \vdash set\ w.\ P}$$

Parameter: $(W, \leq)$

$$Proc\ \gamma = \{ P \mid \gamma \vdash P \}$$

Semantics

$$\mathrm{Msg}\ \gamma := \sum_{a \in \mathbb{A}} \gamma(a) \times \mathcal{P}_{fin}(\mathrm{Msg}\ \gamma) \times \mathrm{World}$$

$$\mathrm{State}\ \gamma := \mathcal{P}(\mathrm{Msg}\ \gamma) \qquad \text{[Linear model]}$$

$$[\![\gamma \vdash P]\!] : \mathcal{P}_{fin}(\mathrm{Msg}\ \gamma) \times \mathrm{World} \to \text{Closure operator on } \mathrm{State}\ \gamma$$

Semantics

$$\mathrm{Msg}\ \gamma := \sum_{a \in \mathbb{A}} \gamma(a) \times \mathcal{P}_{fin}(\mathrm{Msg}\ \gamma) \times \mathrm{World}$$

$$\mathrm{State}\ \gamma := \mathcal{P}(\mathrm{Msg}\ \gamma)$$ [Linear model]

$$[\![\gamma \vdash P]\!] : \mathcal{P}_{fin}(\mathrm{Msg}\ \gamma) \times \mathrm{World} \to \text{Closure operator on } \mathrm{State}\ \gamma$$

Examples:

$$[\![\gamma \vdash \mathrm{put}\ a\ v.\ P]\!](h, w) := \mathrm{let}\ e = (a, v, h, w)\ \mathrm{in}$$
$$\lambda \mathcal{E}.\ \mathcal{E} \cup \{e\} \cup [\![\gamma \vdash P]\!](h \cup \{e\}, w)(\mathcal{E})$$

$$[\![\gamma \vdash \mathrm{watch}\ a\ (\lambda v.\ P_v)]\!](h, w)(\mathcal{E}) :=$$

$$\mathcal{E} \cup \bigcup_{\substack{(a, v', h, w_0) \in \mathcal{E} \\ w \vee w_0 = \overline{w}}} [\![\gamma \vdash P]\!](v \cup v', \overline{w})(\mathcal{E})$$

Equivalence

$$P \approx Q \quad := \quad [\![P]\!] = [\![Q]\!]$$

$$(\nu b)\Big(\text{put } b\ 1 \parallel \text{put } b\ 2 \parallel \text{watch } b\ (\lambda v.\ \text{put } a\ v) \parallel \text{watch } b\ (\lambda v.\ \text{put } a\ v)\Big)$$

$$(\nu c)(\nu b)\Big(\text{put } b\ \text{tt} \parallel \text{put } b\ \text{ff} \parallel \text{watch } b\ (\lambda v.\ \text{if } v \text{ then put } a\ 1 \text{ else watch } a\ (\lambda \_ .\ \text{put } c\ 1))\Big)$$

Final state : an event structure

Given $\gamma \vdash P$, we define

$$\mathcal{E}(P) = [\![P]\!](\emptyset,\emptyset)(\emptyset)$$ an event structure if P is correct

$$\mathcal{E}(\text{put } a\ 1 \parallel \text{watch } a\ (\lambda n.\ \text{put } a\ (n+1)))$$

Expressivity of the language

Able to encode:

- linearisable datatypes (eg. shared reference)

  World: memory trace

- CCS/π channels

  World: partial bijections between output and input prefixes

Clean separation between communication / nondeterminism

Implementation

Goal: compute $\mathcal{E}(P)$

$\rightarrow$ Model induces a lot of redundant computation at each iteration

However, we can look at normal forms upto $\simeq$:

$$N ::= (\nu \vec{a})\big(\underbrace{\text{put } \vec{a}\ \vec{v}}_{\text{initial state}} \parallel \underbrace{\text{watch } \vec{a}\ \vec{N}}_{\text{handlers}}\big)$$

$\mathcal{E}(P)$ can be computed via a transition system on normal forms avoiding repetition

III. A monad for truly concurrent computation

Source $\xrightarrow[\text{Translation}]{\text{Monadic}}$ Process Calculus $\longrightarrow$ C.O. $\longrightarrow$ E.S

From programming languages to process calculi

$$\Gamma \vdash M : \sigma \longrightarrow [\![\Gamma \vdash M : \sigma]\!] : (o \notin \mathrm{dom}\, \gamma) \longrightarrow \mathrm{Proc}([\![\Gamma]\!], o : [\![\sigma]\!])$$

$$[\![e_1 + e_2]\!]\, o = (\nu a b)\big(\, [\![e_1]\!]_a \mid \mathrm{watch}\ a\ (\lambda n_1.\ [\![e_2]\!]_b \mid \mathrm{watch}\ b\ (\lambda n_2.\ \mathrm{put}\ o\, (n_1 + n_2)))\big)$$

Syntactic counterpart of composition in denotational semantics

$$[\![e_1 + e_2]\!] = \mathrm{add} \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$$

From programming languages to process calculi

$$\Gamma \vdash M : \sigma \longrightarrow [\![\Gamma \vdash M : \sigma]\!] : (o \notin \operatorname{dom} \gamma) \to \mathrm{Proc}([\![\Gamma]\!], o : [\![\sigma]\!])$$

$$[\![e_1 + e_2]\!]\, o = (\nu a b) \left([\![e_1]\!]_a \mid \text{watch } a\, (\lambda n_1.\ [\![e_2]\!]_b \mid \text{watch } b\, (\lambda n_2.\ \text{put } o\, (n_1 + n_2))) \right)$$

Syntactic counterpart of composition in denotational semantics

$$[\![e_1 + e_2]\!] = \text{add} \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$$

Unsatisfactory:

Bad performance (each communication takes one iteration in the closure)

Translation hard to read and write

A simple remark

Defining $T_\gamma : \mathrm{Set}_\omega \to \mathrm{Set}_\omega$

$$X \mapsto (o \notin \mathrm{dom}\,\gamma) \to \mathrm{Proc}(\gamma, o : X)$$

This is a monad (up to $\approx$):

- $\mathrm{return}\ x = \lambda o.\ \mathrm{put}\ o\,x$
- $\mathrm{bind}\ m\ f = \lambda o.\ (\nu a)\,(m\,a \mid \mathrm{watch}\ a\ (\lambda v.\ f\,v\,o))$

We can write now simple monadic interpreters:

$$[\![\Gamma \vdash M : \sigma]\!] : T_{[\![\Gamma]\!]}\,([\![\sigma]\!])$$

A simple remark

Defining $T_\gamma : \mathrm{Set}_\omega \to \mathrm{Set}_\omega$

$$X \mapsto (o \notin \mathrm{dom}\,\gamma) \to \mathrm{Proc}(\gamma, o : X)$$

This is a monad (up to $\approx$):

- $\mathrm{return}\ x = \lambda o.\ \mathrm{put}\ o\, x$
- $\mathrm{bind}\ m\ f = \lambda o.\ (\nu a)\,(m\, a \mid \mathrm{watch}\ a\ (\lambda v.\ f\, v\, o))$

We can write now simple monadic interpreters:

$$[\![\Gamma \vdash M : \sigma]\!] : T_{[\![\Gamma]\!]}\,([\![\sigma]\!])$$

$$\begin{aligned}[\![e_1 + e_2]\!] = {} & [\![e_1]\!] \gg\!= \lambda n_1.\\ & [\![e_2]\!] \gg\!= \lambda n_2.\\ & \mathrm{return}\,(n_1 + n_2)\end{aligned}$$

A simple remark

Defining $T_\gamma : \mathrm{Set}_\omega \to \mathrm{Set}_\omega$

$$X \mapsto (o \notin \mathrm{dom}\,\gamma) \to \mathrm{Proc}(\gamma, o : X)$$

This is a monad (up to $\approx$):

- $\mathrm{return}\ x = \lambda o.\ \mathrm{put}\ o\, x$
- $\mathrm{bind}\ m\ f = \lambda o.\ (\nu a)\,(m\, a \mid \mathrm{watch}\ a\ (\lambda v.\ f\, v\, o))$

We can write now simple monadic interpreters:

$$[\![\Gamma \vdash M : \tau]\!] : T_{[\![\Gamma]\!]}\,([\![\tau]\!])$$

$$\begin{aligned}[\![e_1 + e_2]\!] = [\![e_1]\!] \gg\!= \lambda n_1.\\ [\![e_2]\!] \gg\!= \lambda n_2.\\ \mathrm{return}\,(n_1 + n_2)\end{aligned}$$

$\leadsto$ Trick works with most process calculi

But this is not completely trivial

We can actually define bind by induction on m:

$$\text{bind}\ (\lambda o.\ \text{put}\ o\ x.\ P)\ f = f\,x \parallel \text{bind}\ (\lambda o.\ P)\ f$$

$$\text{bind}\ (\lambda o.\ P \parallel Q)\ f = \text{bind}\ (\lambda o.\ P)\ f \parallel \text{bind}\ (\lambda o.\ Q)\ f$$

$$\vdots$$

⇝ The information flow is now occurring at the meta level (i.e. the language in which the interpreter runs)

⇝ Interpreting a sequential language incurs very little overhead (w.r.t. to a sequential interpreter)

Operations of the monad

$$\mathrm{watch} : \mathbb{A} \to T_\gamma(\gamma(a))$$
$$\mathrm{put} : (a : \mathbb{A}) \to \gamma(a) \to T_\gamma(\mathbb{1})$$

$$\mathrm{get} : T_\gamma(W)$$
$$\mathrm{set} : W \to T_\gamma(\mathbb{1})$$

$$\parallel : T_\gamma(X) \times T_\gamma(X) \to T_\gamma(X)$$
$$\mathrm{join} : T_\gamma(X) \times T_\gamma(Y) \to T_\gamma(X \times Y) \qquad \text{[derived]}$$

The reference implementation: Causality

→ Monad implemented in OCaml

→ On top of it: references, channels

With monadic syntax, causal interpreters look like regular interpreters

The reference implementation : Causality

→ Monad implemented in OCaml

→ On top of it : references, channels

With monadic syntax, causal interpreters look like regular interpreters

Causal OCaml : an implementation of concurrent games of OCaml

Architecture :

- Monadic translation à la Moggi
- Game semantics used for sending/receiving functions on a channel.

  ⇝ depends only on values, not the whole AST

Conclusion

* A monadic framework to write causal interpreters
  Aim: explore interactively the causal behaviour of programs

* No proofs written yet: formalisation of certain tricks used by the implementation is difficult

* In the future: model complex language, eg. weak memory specs
  approximate the model
  formal link with the game semantics as we know it