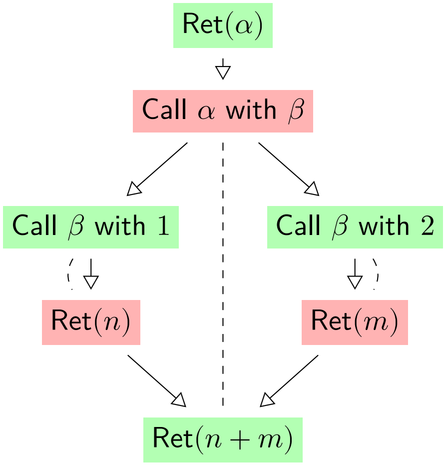
The Causality Monad

A Monad for Truly-Concurrent Computations

Simon Castellan

Causal and Interactive Semantics

$[\![$`fun f -> f 1 + f 2`$]\!] =$



Programs become:

- A **set of interaction points** with Context (events)
- **Causal constraints** between events.

Interest of such semantics

- Models strictly more informative than interleaving models
- In weak memory/distributed systems: focus on causality
- Reasoning on dependencies (static analysis)

State-of-the art: Concurrent games (initiated by Rideau and Winskel'11). Now:

- Functional languages
- Probability
- Quantum
- Shared memory
- Message-passing

Our goal: make it "accessible" to define such models.

How to model a lanaguage with concurrent games?

1. Model each construct by a strategy.
   E.g. conditional gives rise to a strategy if.
2. Use **strategy composition** ($\odot$) to define the interpretation:

$$[\![\mathtt{if}\ M\ N_1\ N_2]\!] = \mathsf{if} \odot \langle [\![M]\!], [\![N_1]\!], [\![N_2]\!] \rangle.$$

How to model a lanaguage with concurrent games?

1. Model each construct by a strategy.
   E.g. conditional gives rise to a strategy if.
2. Use **strategy composition** ($\odot$) to define the interpretation:

$$[\![\texttt{if}\ M\ N_1\ N_2]\!] = \mathsf{if} \odot \langle [\![M]\!], [\![N_1]\!], [\![N_2]\!] \rangle.$$

Standard denotational approach.

- ✓ Little work involved, heavy lifting done by composition.
- ✓ Reasoning by induction on programs.
- ✓ Compositionality **by design**

How to model a lanaguage with concurrent games?

1. Model each construct by a strategy.
   E.g. conditional gives rise to a strategy if.
2. Use **strategy composition** ($\odot$) to define the interpretation:

$$[\![\texttt{if}\ M\ N_1\ N_2]\!] = \mathsf{if} \odot \langle [\![M]\!], [\![N_1]\!], [\![N_2]\!] \rangle.$$

Standard denotational approach.

- ✓ Little work involved, heavy lifting done by composition.
- ✓ Reasoning by induction on programs.
- ✓ Compositionality **by design**

However:

- ✗ Composition operator difficult to understand and untractable.
- ✗ Need to know what a strategy is.
- ✗ Methodology inefficient.

Goal: simplify description and implementation of such semantics.

When to be compositional, and when not to

When computing $[\![P]\!]$, we can take shortcuts:

$$[\![\mathtt{if}\,\mathtt{true}\,M\,N]\!] = [\![M]\!] \qquad \leftarrow [\![N]\!] \text{ not necessary}$$

When to be compositional, and when not to

When computing $[\![P]\!]$, we can take shortcuts:

$$[\![\texttt{if}\,\texttt{true}\,M\;N]\!] = [\![M]\!] \quad \leftarrow [\![N]\!] \text{ not necessary}$$

Requirement: compute $[\![\cdot]\!]$ somewhat lazily:

$$\text{Source} \xrightarrow{\text{Interpretation}} \text{Implicit representation} \xrightarrow{\text{Projection}} \text{Final model}$$

Theorem (Compositionality)

$\mathsf{project}(t \;\mathsf{op}\; u) = \mathsf{project}(t)\; [\![\mathsf{op}]\!]\; \mathsf{project}(u)$.

Issue 1 (semantic): Implicit representation for causal models?

How semantic intepreters work

A typical semantic interpreter for an imperative language:

```
val interpret : program -> (state -> state)

let rec interpret program state = match program with
| Assign (var, value) ->
  assign var (eval state value) state
| Seq (t, u) ->
  let state' = interpret t state in
  interpret u state'
```

$\rightsquigarrow$ We would like the same style, but with concurrency.

Monadic interpreters

Such semantic interprets are **monadic**:

```
type 'a m = state -> 'a * state
val assign : string -> int -> unit m
val eval : expression -> int m

val interpret : program -> unit m
let rec interpret = function
| Assign (var, value) ->
  let* n = eval value in
  assign var n
| Seq (t, u) ->
  let* () = interpret t in
  interpret u
```

Monadic interpreters

Such semantic interprets are **monadic**:

```
type 'a m = state -> 'a * state
val assign : string -> int -> unit m
val eval : expression -> int m

val interpret : program -> unit m
let rec interpret = function
| Assign (var, value) ->
  let* n = eval value in
  assign var n
| Seq (t, u) ->
  let* () = interpret t in
  interpret u
```

This uses a generalised `let*` on monadic computations:

```
val ( let* ) : 'a m -> ('a -> 'b m) -> 'b m
let ( let* ) m f = fun state ->
  let (v, state') = m state in
  f v state'
```

Issue 2 (syntactic): Monadic operations for concurrency?

Plan

1. A monad signature to describe concurrent computations
2. An implementation of it inspired by **closure operators**
3. A projection to event structures (implicit or explicit).
4. Causal interpreter of MiniOCaml.

I. Monadic operations for concurrency

Concurrency primitives

Most concurrency primitives mix causality and conflict:

- Shared memory, Channels à la CCS or π, ...

$\rightsquigarrow$ Difficult to use to describe precisely a causal model.

Concurrency primitives

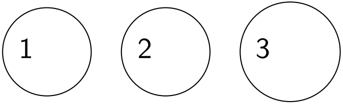
Most concurrency primitives mix causality and conflict:

- Shared memory, Channels à la CCS or π, ...

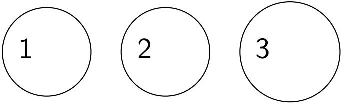
$\rightsquigarrow$ Difficult to use to describe precisely a causal model. Our recipe:

- Parallelism (independant subcomputations)
- Deterministic communication between independent subcomputations
- Nondeterminism as “incompatibility” between computations.

The mental model

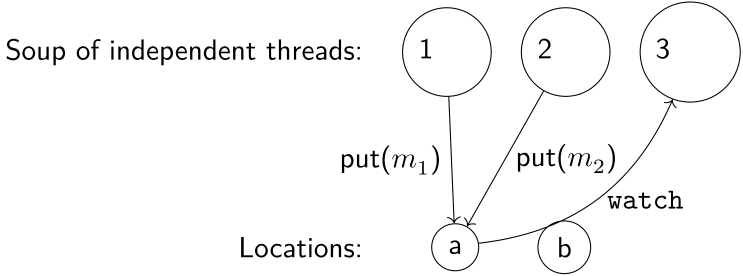
Soup of independent threads: 

The mental model

Soup of independent threads: 

Locations: 

The mental model

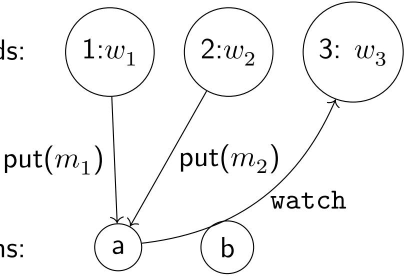


Determinism: Messages **never** consumed, only observed.

The mental model

Soup of independent threads:

Locations:



Determinism: Messages **never** consumed, only observed.

Compatibility: each $w_i \in$ World:

- 3 sees m_1 iff $w_1 \vee w_3$ is defined,
- 3 sees m_2 iff $w_2 \vee w_3$ is defined.

(World, $\vee$) must be a partial monoid, eg.

$$\text{World} = (\mathbb{N} \rightharpoonup \mathbb{N}, \cup).$$

Monadic operations

```
(* Type of computation returning values of type ['a] *)
type 'a t
val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t

(* Parallelism *)
val parallel : 'a t list -> 'a t

(* Communication *)
type 'a loc (* morally simply an ID. *)
val create : unit -> 'a loc
val put : 'a loc -> 'a -> 'b t (* Never returns *)
val watch : 'a loc -> 'a t

(* Nondeterminism: basically a state monad over world *)
type world = ..
(* must implement: val join : world -> world -> world option *)
val world_get : world t
val world_set : world -> unit t
```

Examples: determinism

```
(* To run a computation and get the final values *)
val results : 'a t -> 'a list

# results (return 1)
- int list = [1]
# results (parallel [return 1; return 2])
- int list = [2]
# results (bind (parallel [return 1; return 2])
                          (fun n -> return (n+1)))
- int list = [2; 3]
```

Examples: determinism

```
(* To run a computation and get the final values *)
val results : 'a t -> 'a list

# results (return 1)
- int list = [1]
# results (parallel [return 1; return 2])
- int list = [2]
# results (bind (parallel [return 1; return 2])
                          (fun n -> return (n+1)))
- int list = [2; 3]
let loc = create ()
# results (parallel [watch a; put a 1])
```

Examples: determinism

```
(* To run a computation and get the final values *)
val results : 'a t -> 'a list

# results (return 1)
- int list = [1]
# results (parallel [return 1; return 2])
- int list = [2]
# results (bind (parallel [return 1; return 2])
                        (fun n -> return (n+1)))
- int list = [2; 3]
let loc = create ()
# results (parallel [watch a; put a 1])
- int list = [1]
# results (parallel [watch a; watch a; put a 1; put a 2])
```

Examples: determinism

```
(* To run a computation and get the final values *)
val results : 'a t -> 'a list

# results (return 1)
- int list = [1]
# results (parallel [return 1; return 2])
- int list = [2]
# results (bind (parallel [return 1; return 2])
                          (fun n -> return (n+1)))
- int list = [2; 3]
let loc = create ()
# results (parallel [watch a; put a 1])
- int list = [1]
# results (parallel [watch a; watch a; put a 1; put a 2])
- int list = [1; 2; 1; 2]
# results (watch a)
```

Examples: determinism

```
(* To run a computation and get the final values *)
val results : 'a t -> 'a list

# results (return 1)
- int list = [1]
# results (parallel [return 1; return 2])
- int list = [2]
# results (bind (parallel [return 1; return 2])
                          (fun n -> return (n+1)))
- int list = [2; 3]
let loc = create ()
# results (parallel [watch a; put a 1])
- int list = [1]
# results (parallel [watch a; watch a; put a 1; put a 2])
- int list = [1; 2; 1; 2]
# results (watch a)
- int list = []
```

Examples: nondeterminism

```
type world = bool option
let join b b' = match (b, b') with
| (None, x) | (x, None) -> x
| Some b, Some b' when b = b' -> Some b
| _ -> None

let coin = parallel [set_world (Some true) >> return true;
                     set_world (Some false) >> return false]
let fork = parallel [return true; return false]
let loc = create ()

# results (fork >>= if b then watch loc else put loc 1)

# results (coin >>= if b then watch loc else put loc 1)
```

Examples: nondeterminism

```
type world = bool option
let join b b' = match (b, b') with
| (None, x) | (x, None) -> x
| Some b, Some b' when b = b' -> Some b
| _ -> None

let coin = parallel [set_world (Some true) >> return true;
                     set_world (Some false) >> return false]
let fork = parallel [return true; return false]
let loc = create ()

# results (fork >>= if b then watch loc else put loc 1)
- int list = [1]
# results (coin >>= if b then watch loc else put loc 1)
- int list = []
(* Because true and false are returned in incompatible world.
   watch and put are invisible to each other. *)
```

Expressivity

Signature expressive enough for concurrent data structures.

```
let c = Cell.create 0 (* Create a concurrent memory cell *)
# results (parallel [Cell.set c 1 >> discard;
                     Cell.set c 2 >> discard;
                     Cell.get c])
- int list = [0; 1; 1; 2; 2]
```

Expressivity

Signature expressive enough for concurrent data structures.

```
let c = Cell.create 0 (* Create a concurrent memory cell *)
# results (parallel [Cell.set c 1 >> discard;
                     Cell.set c 2 >> discard;
                     Cell.get c])
- int list = [0; 1; 1; 2; 2]
```

One result per causal explanation of the read

1. 0: read is performed first
2. First 1: read is performed only after `Cell.set c 1`
3. Second 1: read is performed after `Cell.set c 2` and `Cell.set c 1` (in this order)
4. Similar explanations for the two 2s.

Expressivity

Signature expressive enough for concurrent data structures.

```
let c = Cell.create 0 (* Create a concurrent memory cell *)
# results (parallel [Cell.set c 1 >> discard;
             Cell.set c 2 >> discard;
             Cell.get c])
- int list = [0; 1; 1; 2; 2]
```

One result per causal explanation of the read

1. 0: read is performed first
2. First 1: read is performed only after `Cell.set c 1`
3. Second 1: read is performed after `Cell.set c 2` and `Cell.set c 1` (in this order)
4. Similar explanations for the two 2s.

In this case, a world is a trace t of memory operations on c:

$$t \vee t' = \text{maximum of } t \text{ and } t' \text{ for prefix ordering if defined}$$

$\rightsquigarrow$ Implementation somewhat subtle but short ($\sim$ 20lines)

II. Implementation of the signature

The mental model

Threads are modelled by **state transformers** on:

$$\mathsf{State} := \mathscr{P}(\mathsf{Msg}) \qquad \text{where } \mathsf{Msg} := \mathsf{Loc} \times \mathsf{Value} \times \mathsf{World}.$$

(Closed) programs are functions $f : \mathsf{State} \to \mathsf{State}$ such that:

- **monotonic**: $X \subseteq Y \Rightarrow f(X) \subseteq f(Y)$
- **increasing**: $X \subseteq f(X)$.

The **final state** of such an f is the limit of:

$$\emptyset \subseteq f(\emptyset) \subseteq \ldots$$

Two possibilities:

- **Mathematic world**: limit always exists but may only be reached after a transfinite number of steps.
- **Real world**: if the sequence does not converge in finite time, final state is not defined.

The monad in maths

We thus define:

$$T(X) := \mathsf{Loc}(X) \to \mathsf{World} \to (\mathsf{State} \to \mathtt{State})$$

where $\mathtt{Loc}(X)$ set of locations of type X.

$$\mathsf{return}(x) := \lambda\ell.\lambda w.\, \lambda\sigma.\, \sigma \cup \{(\ell, x, w)\}.$$

The monad in maths

We thus define:

$$T(X) := \mathsf{Loc}(X) \to \mathsf{World} \to (\mathsf{State} \to \mathtt{State})$$

where $\mathtt{Loc}(X)$ set of locations of type X.

$$\mathsf{return}(x) := \lambda \ell. \lambda w.\, \lambda \sigma.\, \sigma \cup \{(\ell, x, w)\}.$$
$$\mathsf{put}(\ell, v) := \lambda \_ . \lambda w.\, \lambda \sigma. \sigma \cup \{(\ell, v, w)\}$$

The monad in maths

We thus define:

$$T(X) := \mathsf{Loc}(X) \to \mathsf{World} \to (\mathsf{State} \to \mathtt{State})$$

where $\mathtt{Loc}(X)$ set of locations of type X.

$$\begin{aligned}
\mathsf{return}(x) &:= \lambda\ell.\lambda w.\,\lambda\sigma.\,\sigma \cup \{(\ell, x, w)\}.\\
\mathsf{put}(\ell, v) &:= \lambda_.\lambda w.\,\lambda\sigma.\sigma \cup \{(\ell, v, w)\}\\
\mathsf{parallel}(l) &:= \lambda\ell.\,\lambda w.\,\lambda\sigma.\bigcup_{c\in l} c(\ell, w, \sigma)
\end{aligned}$$

The monad in maths

We thus define:

$$T(X) := \mathsf{Loc}(X) \to \mathsf{World} \to (\mathsf{State} \to \mathtt{State})$$

where $\mathtt{Loc}(X)$ set of locations of type X.

$$\begin{aligned}
\mathsf{return}(x) &:= \lambda\ell.\lambda w.\,\lambda\sigma.\,\sigma \cup \{(\ell, x, w)\}.\\
\mathsf{put}(\ell, v) &:= \lambda_.\lambda w.\,\lambda\sigma.\sigma \cup \{(\ell, v, w)\}\\
\mathsf{parallel}(l) &:= \lambda\ell.\,\lambda w.\,\lambda\sigma.\bigcup_{c\in l} c(\ell, w, \sigma)\\
\mathsf{watch}(\ell) &:= \lambda\ell_r.\lambda w_1.\,\lambda\sigma.\sigma \cup \bigcup_{\substack{(\ell, v, w_2)\in\sigma\\ w_1 \vee w_2 = w}} \{(\ell_r, v, w)\}
\end{aligned}$$

The monad in maths

We thus define:

$$T(X) := \mathsf{Loc}(X) \to \mathsf{World} \to (\mathsf{State} \to \mathtt{State})$$

where $\mathtt{Loc}(X)$ set of locations of type X.

$$\begin{aligned}
\mathsf{return}(x) &:= \lambda\ell.\lambda w.\, \lambda\sigma.\, \sigma \cup \{(\ell, x, w)\}. \\
\mathsf{put}(\ell, v) &:= \lambda_.\lambda w.\, \lambda\sigma.\sigma \cup \{(\ell, v, w)\} \\
\mathsf{parallel}(l) &:= \lambda\ell.\, \lambda w.\, \lambda\sigma. \bigcup_{c \in l} c(\ell, w, \sigma) \\
\mathsf{watch}(\ell) &:= \lambda\ell_r.\lambda w_1.\, \lambda\sigma.\sigma \cup \bigcup_{\substack{(\ell, v, w_2) \in \sigma \\ w_1 \vee w_2 = w}} \{(\ell_r, v, w)\} \\
\mathsf{bind}(m, f) &:= \lambda\ell.\lambda w.\lambda\sigma.\mathsf{newloc}\ \alpha\ \mathsf{in} \\
&\qquad \mathsf{parallel}[m\, \alpha\, w\, \sigma; \bigcup_{\substack{(\alpha, v, w_2) \in \sigma \\ w_1 \vee w_2 = w}} f\, v\, w\, \sigma]
\end{aligned}$$

The monad in code

State transformers are very inefficient to itererate.
$\rightsquigarrow$ Alternative representation:

$$\mathscr{C} ::= \mathsf{State} \times \mathcal{P}(\mathsf{Loc} \times (\mathsf{Msg} \to \mathscr{C}))$$

The monad in code

State transformers are very inefficient to itererate.
$\rightsquigarrow$ Alternative representation:

$$\begin{aligned} \mathscr{C} &::= \mathsf{State} \times \mathcal{P}(\mathsf{Loc} \times (\mathsf{Msg} \to \mathscr{C})) \\ [\![\cdot]\!] &: \mathscr{C} \to (\mathsf{State} \to \mathsf{State}) \end{aligned}$$

The monad in code

State transformers are very inefficient to itererate.

$\rightsquigarrow$ Alternative representation:

$$\mathscr{C} ::= \mathsf{State} \times \mathcal{P}(\mathsf{Loc} \times (\mathsf{Msg} \to \mathscr{C}))$$

$$[\![\cdot]\!] : \mathscr{C} \to (\mathsf{State} \to \mathsf{State})$$

$$[\![(\sigma_0, H)]\!](\sigma) = \sigma \cup \sigma_0 \bigcup_{(\ell,v,w)\in\sigma} \bigcup_{(\ell,f)\in H} [\![f(v)]\!](\sigma).$$

$\rightsquigarrow$ Fixpoint reached through a transition system avoiding recalculations.

What is swept under the rug

1. How to send the same value twice?
   $\rightsquigarrow$ Pseudo-Nominal solutions

2. Handling of public/private names to get a well-behaved equivalence.
   $\rightsquigarrow$ Nominal techniques.

3. Choices of data structure to represent State, $\mathscr{C}$

4. Bind is costly (allocation of a name) + parallel
   $\rightsquigarrow$ Free monads to eliminate binds.

III. Generating event structures

The event structure spanned by a state

From a term $t : T(X)$, we can extract a final state:

$$\mathsf{final}(t) = \mathsf{lfix}(t\,\alpha\,\bot)(\alpha \text{ fresh}).$$

This final state is a *soup of messages*.
To recover some structure, we extend worlds with a *causal view*

$$\mathsf{World} = \{\mathsf{view} : \mathcal{P}(\mathsf{Msg}); \ldots\}$$

Then, on $\sigma \in$ State, we define:

- A **partial order**: $m \leq_\sigma m'$ iff $m \in m'$.world.view.
- A **conflict relation**: $m \#_\sigma m'$ iff $(m.\mathsf{world} \vee m'.\mathsf{world})$ undef.

This creates an **event structure**.
$\rightsquigarrow$ In general, only consider certain locations.

Writing an inteprreter using Causality

IV. Semantics of Mini-OCaml

Illustration: Semantics of a functional language

Source:

- **MiniOCaml**: OCaml restricted to basic datatypes, functions.
- **Concurrent semantics**: eg. (M, N)

Target: Causal and Interactive semantics

- Final Object: Event Structure
- Event are messages between program and its context
- Two kinds of events: Call and Returns
- All messages are "first-order": functions are passed by name

Protocol described by game semantics. (GS is model-agnostic)

Two steps

1. Monadic translation of the language using `Causality`.

```
type value = Int of int | Function of (value -> value t)

let rec eval : expression -> value t = function
| AstInt n -> return (Int n)
| AstApp (t, u) -> eval t >>= function
  | Function f -> eval u >>= f
```

Two steps

1. Monadic translation of the language using `Causality`.

```
type value = Int of int | Function of (value -> value t)

let rec eval : expression -> value t = function
| AstInt n -> return (Int n)
| AstApp (t, u) -> eval t >>= function
  | Function f -> eval u >>= f
```

2. Game Semantics is used to describe the communication protocol between a program and the context.

```
type msg = {
  polarity: Program | Context;
  kind: Call | Return;
  value: Int of int | Function;
  ...
}
val recv : msg location -> value t
val send : msg location -> value -> unit t
```

Combining the two steps

We thus get:

```
val sem : expression -> msg location -> unit t
```

This allows interacting with an expression through a location:

- Exploring (partially) the behaviour of the program.
  $\rightsquigarrow$ Replicate certain races hard to simulate.
- Causal debugging.
- Dynamic analysis.

Demo

Conclusion

Main contributions:

- Framework for operational description of causal semantics

Useful for: weak memory, distributed systems, ...

Perspectives:

- **Reasoning:** Lots of difficulties (eg. nominal aspects)
- **Formalisation:** Could be easier than usual causal models on a proof assistant.
- **Real case study:** Weak memory?
- **Static analysis**: eg. write abstract interpreters
- **Finitary infinity**: how to generate Petri Nets?

  ```
  (* Generates a syntactic loop (à la Petri nets) *)
  val fix : ('a t -> 'a t) -> 'a t
  ```

- and many other things.