The Causality Monad
A Monad for Truly-Concurrent Computations

Simon Castellan
Causal and Interactive Semantics

\[
\text{[fun } f \rightarrow f 1 + f 2] = \begin{align*}
\text{Call } \alpha \text{ with } \beta \\
\text{Call } \beta \text{ with 1} \\
\text{Call } \beta \text{ with 2} \\
\text{Ret}(n) \\
\text{Ret}(m) \\
\text{Ret}(n + m)
\end{align*}
\]

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 language 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:

   \[
   [\text{if } M \; N_1 \; N_2] = \text{if} \odot \langle [M], [N_1], [N_2] \rangle.
   \]
How to model a language with concurrent games?

1. Model each construct by a strategy.
   E.g. conditional gives rise to a strategy if.

2. Use **strategy composition** \((\circ)\) to define the interpretation:

   \[
   \left[ \text{if } M \ N_1 \ N_2 \right] = \text{if} \circ \langle [M], [N_1], [N_2] \rangle.
   \]

   **Standard denotational approach.**

   - Little work involved, heavy lifting done by composition.
   - Reasoning by induction on programs.
   - Compositionality **by design**
How to model a language with concurrent games?

1. Model each construct by a strategy.
   E.g. conditional gives rise to a strategy if.

2. Use strategy composition \( (\circ) \) to define the interpretation:

\[
\text{[if } M \text{ } N_1 \text{ } N_2 \text{]} = \text{if } \circ \langle [M], [N_1], [N_2] \rangle.
\]

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:

\[
\begin{align*}
\text{if true } M &\  N = [M] \quad \leftarrow [N] \text{ not necessary}
\end{align*}
\]
When to be compositional, and when not to

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

$$[[\text{if true } M \ N]] = [[M]] \leftarrow [[N]] \text{ not necessary}$$

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

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

**Theorem (Compositionality)**

$$\text{project}(t \text{ op } u) = \text{project}(t) \ [\text{op}] \text{ project}(u).$$

**Issue 1 (semantic)**: Implicit representation for causal models?
A typical semantic interpreter for an imperative language:

```ocaml
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'
```

⇝ We would like the same style, but with concurrency.
Monadic interpreters

Such semantic interprets are **monadic**:

```ml
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:

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

**Issue 2 (syntactic):** Monadic operations for concurrency?
Monadic interpreters

Such semantic interprets are **monadic**:  
\[
\text{type } 'a\ m = \text{state} \rightarrow 'a \times \text{state}
\]
\[
\text{val assign : string} \rightarrow \text{int} \rightarrow \text{unit} \ m
\]
\[
\text{val eval : expression} \rightarrow \text{int} \ m
\]
\[
\text{val interpret : program} \rightarrow \text{unit} \ m
\]
\[
\text{let rec interpret } = \text{function}
\]
\[
| \text{Assign} (\text{var}, \text{value}) \rightarrow
\]
\[
| \text{let* } n = \text{eval value in}
\]
\[
\text{assign var n}
\]
\[
| \text{Seq} (t, u) \rightarrow
\]
\[
| \text{let* } () = \text{interpret t in}
\]
\[
\text{interpret u}
\]

This uses a generalised \text{let*} on monadic computations:
\[
\text{val ( let* ) : 'a\ m \rightarrow ('a \rightarrow 'b\ m) \rightarrow 'b\ m}
\]
\[
\text{let ( let* ) m f = fun state } \rightarrow
\]
\[
| \text{let (v, state')} = m\ state \in
\]
\[
| f\ v\ state'
\]

**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).
I. Monadic operations for concurrency
Most concurrency primitives mix causality and conflict:

- Shared memory, Channels à la CCS or $\pi$, ...

$\iff$ Difficult to use to describe precisely a causal model.
Most concurrency primitives mix causality and conflict:

▶ Shared memory, Channels à la CCS or $\pi$, ...

⇝ 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: 1 2 3

Locations:
put(\(m_1\)) put(\(m_2\)) watch

Determinism: Messages never consumed, only observed.

Compatibility: each \(w_i \in \text{World}: \)

\(\exists \) sees \(m_1\) iff \(w_1 \lor w_3\) is defined,

\(\exists \) sees \(m_2\) iff \(w_2 \lor w_3\) is defined.

\(\text{World}, \lor\) must be a partial monoid, eg. \(\text{World} = (\mathbb{N} \rightharpoonup \mathbb{N}, \cup)\).
The mental model

Soup of independent threads: 1 2 3

Locations: a b

Determinism: Messages never consumed, only observed.

Compatibility: each \( w_i \in \text{World} \):

\( \triangleright w_1 \lor w_3 \) sees \( m_1 \) iff \( w_1 \lor w_3 \) is defined,

\( \triangleright w_2 \lor w_3 \) sees \( m_2 \) iff \( w_2 \lor w_3 \) is defined.

\((\text{World}, \lor)\) must be a partial monoid, eg. \(\text{World} = (\mathbb{N} \to \mathbb{N}, \cup)\).
The mental model

Soup of independent threads:

Locations:

Determinism: Messages never consumed, only observed.
The mental model

Soup of independent threads:

Locations:

Determinism: Messages never consumed, only observed.

Compatibility: each $w_i \in \text{World}$:

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

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

$$\text{World} = (\mathbb{N} \rightarrow \mathbb{N}, \lor).$$
Monadic operations

(* Type of computation returning values of type ['a] *)

```plaintext
type 'a t
val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t
```

(* Parallelism *)

```plaintext
val parallel : 'a t list -> 'a t
```

(* Communication *)

```plaintext
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 *)

```plaintext
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 *)

```ocaml
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 *)

```ocaml
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 *)

```ocaml
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])
```

The Causality Monad · Simon Castellan
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 >>= fun b -> if b then watch loc else put loc 1)
# results (coin >>= fun b -> if b then watch loc else put loc 1)
Examples: nondeterminism

```ocaml
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. *)
```

The Causality Monad · Simon Castellan
Expressivity

Signature expressive enough for concurrent data structures.

```ocaml
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.

```haskell
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.

```haskell
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 $c$:

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

Implementation somewhat subtle but short ($\sim$ 20 lines)
II. Implementation of the signature
The mental model

Threads are modelled by state transformers on:

\[ \text{State} := \mathcal{P}(\text{Msg}) \quad \text{where} \quad \text{Msg} := \text{Loc} \times \text{Value} \times \text{World}. \]

(Closed) programs are functions \( f : \text{State} \rightarrow \text{State} \) 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:

\[ T(X) := \text{Loc}(X) \rightarrow \text{World} \rightarrow (\text{State} \rightarrow \text{State}) \]

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

\[ \text{return}\,(x) := \lambda \ell.\lambda w.\lambda \sigma. \sigma \cup \{(\ell, x, w)\}. \]
The monad in maths

We thus define:

\[ T(X) := \text{Loc}(X) \rightarrow \text{World} \rightarrow (\text{State} \rightarrow \text{State}) \]

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

\[
\begin{align*}
\text{return}(x) & := \lambda \ell. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, x, w)\}. \\
\text{put}(\ell, v) & := \lambda _. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, v, w)\}
\end{align*}
\]
The monad in maths

We thus define:

\[ T(X) := \text{Loc}(X) \rightarrow \text{World} \rightarrow (\text{State} \rightarrow \text{State}) \]

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

\[
\begin{align*}
\text{return}(x) & := \lambda \ell. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, x, w)\}. \\
\text{put}(\ell, v) & := \lambda \_._. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, v, w)\} \\
\text{parallel}(l) & := \lambda \ell. \lambda w. \lambda \sigma. \bigcup_{c \in l} c(\ell, w, \sigma)
\end{align*}
\]
The monad in maths

We thus define:

\[ T(X) := \text{Loc}(X) \to \text{World} \to (\text{State} \to \text{State}) \]

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

\[
\begin{align*}
\text{return}(x) & := \lambda \ell. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, x, w)\}. \\
\text{put}(\ell, v) & := \lambda _. \lambda w. \lambda \sigma. \sigma \cup \{(\ell, v, w)\} \\
\text{parallel}(l) & := \lambda \ell. \lambda w. \lambda \sigma. \bigcup_{c \in l} c(\ell, w, \sigma) \\
\text{watch}(\ell) & := \lambda \ell_r. \lambda w_1. \lambda \sigma. \sigma \cup \bigcup_{(l, v, w_2) \in \sigma} \{(\ell_r, v, w)\} \cup \{(\ell, v, w_2) \in \sigma \text{ s.t. } w_1 \lor w_2 = w\}
\end{align*}
\]
The monad in maths

We thus define:

\[ T(X) := \text{Loc}(X) \to \text{World} \to (\text{State} \to \text{State}) \]

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

- \( \text{return}(x) := \lambda \ell . \lambda w . \lambda \sigma . \sigma \cup \{(\ell, x, w)\} \).
- \( \text{put}(\ell, v) := \lambda _. \lambda w . \lambda \sigma . \sigma \cup \{(\ell, v, w)\} \)
- \( \text{parallel}(l) := \lambda \ell . \lambda w . \lambda \sigma . \bigcup_{c \in l} c(\ell, w, \sigma) \)
- \( \text{watch}(\ell) := \lambda \ell_r . \lambda w_1 . \lambda \sigma . \sigma \cup \bigcup_{(\ell, v, w_2) \in \sigma} \{(\ell_r, v, w)\} \)
  \[ w_1 \lor w_2 = w \]
- \( \text{bind}(m, f) := \lambda \ell . \lambda w . \lambda \sigma . \text{newloc } \alpha \ \text{in} \)
  \[ \text{parallel}[m \alpha w \sigma; \bigcup_{(\alpha, v, w_2) \in \sigma} f v w \sigma] \]
  \[ w_1 \lor w_2 = w \]
The monad in code

State transformers are very inefficient to iterate.
⇒ Alternative representation:

\[ C ::= \text{State} \times \mathcal{P}(\text{Loc} \times (\text{Msg} \rightarrow C)) \]
State transformers are very inefficient to iterate.

Inlining Alternative representation:

\[
C ::= \text{State} \times \mathcal{P} (\text{Loc} \times (\text{Msg} \rightarrow C)) \\
\cdot : C \rightarrow (\text{State} \rightarrow \text{State})
\]
The monad in code

State transformers are very inefficient to iterate.
⇝ Alternative representation:

\[ C ::= \text{State} \times \mathcal{P}(\text{Loc} \times (\text{Msg} \rightarrow C)) \]
\[ \llbracket \cdot \rrbracket : C \rightarrow (\text{State} \rightarrow \text{State}) \]
\[ \llbracket (\sigma_0, H) \rrbracket (\sigma) = \sigma \cup \sigma_0 \bigcup (\ell, v, w) \in \sigma \bigcup (\ell, f) \in H \llbracket f(v) \rrbracket (\sigma). \]

⇝ Fixpoint reached through a transition system avoiding recalculations.
What is swept under the rug

1. How to send the same value twice?
   ⇝ Pseudo-Nominal solutions

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

3. Choices of data structure to represent State, $C$

4. Bind is costly (allocation of a name) + parallel
   ⇝ 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:

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

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

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

Then, on $\sigma \in \text{State}$, we define:

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

This creates an **event structure**.

$\Rightarrow$ In general, only consider certain locations.
Writing an interpreter 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.

```ocaml
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.

\[
\text{type value} = \text{Int of int} \mid \text{Function of (value } \rightarrow \text{ value) t}
\]

\[
\text{let rec eval : expression } \rightarrow \text{ value t} = \text{function}
| \text{AstInt n } \rightarrow \text{return (Int n)}
| \text{AstApp (t, u) } \rightarrow \text{eval t } \gg= \text{function}
| \text{Function f } \rightarrow \text{eval u } \gg= f
\]

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

\[
\text{type msg} = \{
    \text{polarity: Program } \mid \text{ Context;}
    \text{kind: Call } \mid \text{ Return;}
    \text{value: Int of int } \mid \text{ Function;}
    \ldots
\}
\]

\[
\text{val recv : msg location } \rightarrow \text{ value t}
\]

\[
\text{val send : msg location } \rightarrow \text{ value } \rightarrow \text{ unit t}
\]
Combining the two steps

We thus get:

\[
\text{val sem : expression } \rightarrow \text{ msg location } \rightarrow \text{ unit t}
\]

This allows interacting with an expression through a location:

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

Main contributions:
- Framework for operational description of causal semantics
  Useful for: weak memory, distributed systems, ...

Perspectives:
- **Reasoning:** Lots of difficulties (e.g. nominal aspects)
- **Formalisation:** Could be easier than usual causal models on a proof assistant.
- **Real case study:** Weak memory?
- **Static analysis:** e.g. 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.