An Equational Axiomatization of Dynamic Threads

Cristina Matache

ACV Workshop, 25 July 2026

University of Birmingham

Joint work with Ohad Kammar, Jack Liell-Cock, Sam Lindley, and Sam Staton

Introduction — Denotational semantics

Denotational semantics is about finding models of programs in order to:

- **reason** about program behaviour,
- and to compare them more easily.
- Models should be **modular**, can combine different features.

Methods:

- the simply-typed λ-calculus as a core programming language;
- category theory as a tool for organizing models.

Introduction — Effects

For functional programming: a program is a function that takes an input and returns a value.

To add **computational effects**: programs interact with the environment e.g.

- manipulating memory,
- taking input from a keyboard,
- probabilistic computation,

use **strong monads** [Moggi] for modelling.

Monads are presented using **algebraic theories** [Plotkin & Power], which give program equations.

Introduction — Concurrency

Concurrency:
Programs interact with other programs running at the same time.

Key idea

Concurrency can be treated as a computational effect.

Obtain **equational reasoning** principles for concurrent higher-order programs. Sound and complete w.r.t. operational semantics.

What I mean by concurrency in this talk

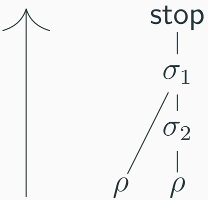
Dynamic threads

Programs create new threads dynamically, and wait for them to finish.
E.g. POSIX-like `fork`.

Concurrent programs denote **partial orders with labels** (pomsets).

Example:

$$\begin{aligned}\text{let } y = \underline{\text{fork}}() \text{ in case } \left(\underline{\text{act}}_\rho(); y\right) \\ \text{of } \{\ \text{inj}_1(a) \Rightarrow \underline{\text{wait}}(a); \underline{\text{act}}_{\sigma_1}(); \underline{\text{stop}}() \\ \text{inj}_2() \Rightarrow \underline{\text{act}}_{\sigma_2}(); \underline{\text{stop}}()\}\end{aligned}$$

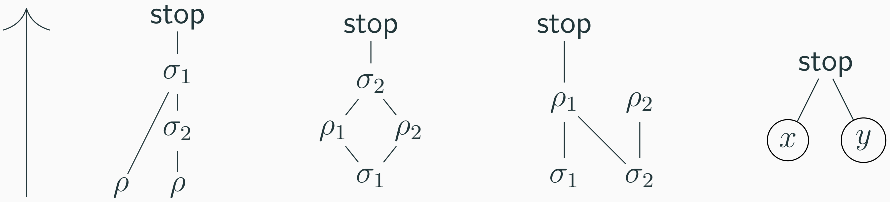


- **Labels** denote observable actions.
- **Partial order** denotes observable dependencies.

What I mean by concurrency in this talk

Concurrent programs denote **partial orders with labels** (pomsets).

More examples of pomsets that are denotations:



Pomsets are a long-established model of true concurrency.

e.g. [Nielsen et al'81], [Pratt'86].

Related work — Gap between theory and practice

Algebraic effects inspired **concurrency libraries** via effect handlers
e.g. in OCaml [Sivaramakrishnan et al'21], in WebAssembly [Phipps-Costin et al'23]
Not clear how these tie to theoretical foundations of effects.

Concurrency theory:

- process algebra e.g. ACP;
- pomsets, event structures e.g. [Nielsen et al'81], [Pratt'86];
- concurrent Kleene algebra [Hoare et al'11];

Not clear how these apply to the semantics of functional programming (i.e. extensions of simply-typed lambda calculus).

Related work — Algebraic effects for concurrency

(1) Free-algebra models of the π-calculus [Stark'08]

(2) CSP using algebraic effects and handlers [van Glabbeek&Plotkin'10]

(3) Algebraic theories for shared state concurrency [Dvir et al'22, '25]

(4) Cooperative threads with shared state [Abadi&Plotkin'10]

These are all **interleaving models**.
In (1)-(3), parallel composition is not an algebraic effect.

In this talk: true concurrency semantics for a core functional programming language. `fork` is an algebraic effect.

Summary of this work

- An **algebraic theory** that axiomatizes forking threads and waiting for them. And a characterization of the theory in terms of **pomsets**.
- The theory induces a monad in a standard way,
- and hence a denotational semantics for STLC $+$ fork, wait. Which we show matches an operational semantics.

Outline

Calculus

Call-by-value lambda calculus extended with:

$$\text{fork} : \text{unit} \to \text{tid} + \text{unit} \qquad \text{wait} : \text{tid} \to \text{unit} \qquad \text{stop} : \text{unit} \to \text{empty}$$

$$\text{act}_\sigma : \text{unit} \to \text{unit}$$

tid	base type of IDs of **sets of** threads; only introduced by fork
fork$()$	spawns new child thread, copying the parent's continuation; can check whether parent or child by looking at result of fork
wait(a)	the current thread waits for all threads in a to finish
stop$()$	end current thread, unblocks all threads waiting for it
act$_\sigma()$	performs observable action σ

Operational semantics by example

Operational semantics is a relation between pools of threads.

The observable behaviour of programs is captured by partial orders labelled by observable actions.

$$\begin{aligned}\mathsf{let}\ y = \underline{\mathsf{fork}}()\ \mathsf{in\ case}\ y\ \mathsf{of}\ \{\mathsf{inj}_1(a) &\Rightarrow \underline{\mathsf{act}}_{\sigma_1}(); \underline{\mathsf{stop}}()\\ \mathsf{inj}_2() &\Rightarrow \underline{\mathsf{act}}_{\sigma_2}(); \underline{\mathsf{stop}}()\}\end{aligned}$$

$$\begin{tikzcd} \mathsf{stop} & \\ \sigma_1 \arrow[u, dash] & \sigma_2 \end{tikzcd}$$

$$\begin{aligned}\mathsf{let}\ y = \underline{\mathsf{fork}}()\ \mathsf{in\ case}\ y\ \mathsf{of}\ \{\mathsf{inj}_1(a) &\Rightarrow \underline{\mathsf{wait}}(a); \underline{\mathsf{act}}_{\sigma_1}(); \underline{\mathsf{stop}}()\\ \mathsf{inj}_2() &\Rightarrow \underline{\mathsf{act}}_{\sigma_2}(); \underline{\mathsf{stop}}()\}\end{aligned}$$

$$\begin{tikzcd} \mathsf{stop} \\ \sigma_1 \arrow[u, dash] \\ \sigma_2 \arrow[u, dash] \end{tikzcd}$$

stop is extra structure, a maximal element.

Operational semantics by example

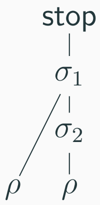
Operational semantics is a relation between pools of threads.

fork duplicates the continuation.

$$\Big\{[b]\,\mathsf{let}\;y = \underline{\mathsf{fork}}()\;\mathsf{in\;case}\;\big(\underline{\mathsf{act}}_\rho(); y\big)$$

$$\mathsf{of}\;\{\;\mathsf{inj}_1(a) \Rightarrow \underline{\mathsf{wait}}(a); \underline{\mathsf{act}}_{\sigma_1}(); \underline{\mathsf{stop}}()$$

$$\mathsf{inj}_2() \Rightarrow \underline{\mathsf{act}}_{\sigma_2}(); \underline{\mathsf{stop}}()\}\Big\}$$



$$\longrightarrow \Big\{[b]\,\mathsf{case}\;\big(\underline{\mathsf{act}}_\rho(); \mathsf{inj}_1(a)\big)\;\mathsf{of}\ldots,$$

$$[a]\,\mathsf{case}\;\big(\underline{\mathsf{act}}_\rho(); \mathsf{inj}_2()\big)\;\mathsf{of}\ldots\Big\}$$

Operational semantics by example

stop discards the continuation.

$$\text{let } y = \mathsf{fork}() \text{ in case } y$$
$$\text{of } \{ \ \mathsf{inj}_1(a) \Rightarrow \mathsf{wait}(a); \mathsf{act}_{\sigma_1}(); \mathsf{stop}()$$
$$\mathsf{inj}_2() \Rightarrow \mathsf{act}_{\sigma_2}(); \mathsf{stop}(); \mathsf{act}_{\rho}() \}$$

$$\begin{array}{c} \mathsf{stop} \\ | \\ \sigma_1 \\ | \\ \sigma_2 \end{array}$$

We will axiomatize fork, wait, stop, act_σ with 9 equations.

Outline

Algebraic operations vs generic effects

- So far, **generic effects** e.g. fork, wait, natural for programming,
- correspond to **algebraic operations**, natural for algebra.

Given a **generic effect** $\underline{\mathsf{op}} : A \to B$,
the **algebraic operation** op takes a value of type A and B continuations.

Example:

$$\underline{\mathsf{fork}} : \mathsf{unit} \to \mathsf{tid} + \mathsf{unit} \qquad \text{vs} \qquad \mathsf{fork}(a.x(a),\ y)$$

fork binds a new ID a, bound in $x(a)$.
a refers to the single thread y.

Algebraic operations vs generic effects

Given a **generic effect** $\underline{\mathsf{op}} : A \to B$,
the **algebraic operation** op takes a value of type A and B continuations.

Example:

$$\underline{\mathsf{fork}} : \mathsf{unit} \to \mathsf{tid} + \mathsf{unit} \qquad \text{vs} \qquad \mathsf{fork}(a.x(a),\ y)$$

tid $=$ type of IDs of sets of threads, has a semilattice structure.

- $a \oplus b$: tid is the ID of the union of the sets of threads a and b
- 0 : tid is the empty set of threads.

Waiting on $a \oplus b$ means waiting on **all** threads in a and b to stop.

Algebraic operations vs generic effects

Given a **generic effect** $\underline{\mathsf{op}} : A \to B$,
the **algebraic operation** op takes a value of type A and B continuations.

Example:

$$\underline{\mathsf{fork}} : \mathsf{unit} \to \mathsf{tid} + \mathsf{unit} \qquad \text{vs} \qquad \mathsf{fork}(a.x(a),\ y)$$

a is a parameter in a parameterized algebraic theory [Staton'13]. Extension of algebraic theories with binding. Has correspondence to monads.

Variables x, y, z take (a finite number of) parameters as input. Example:

$$x : 1, y : 0 \mid \cdot \vdash \mathsf{fork}(a.x(a), y) \qquad z : 2 \mid a, b \vdash z(a, b)$$

Algebraic operations vs generic effects

tid $=$ type of IDs of sets of threads, has a semilattice structure.

fork$(a.x(a),\ y)$	fork : unit $\to$ tid $+$ unit
wait$(u;\ x)$	wait : tid $\to$ unit

u is the ID of a set of threads; wait on all threads in u, continue as x

stop	stop : unit $\to$ empty

has no continuation

act$_\sigma(x)$	act$_\sigma$: unit $\to$ unit

performs action σ and continues as x

Could rewrite the example programs using algebraic operations.

An algebraic theory of dynamic threads

Interaction of wait with the semilattice structure of thread IDs.

$$\mathsf{wait}(0; x) = x \tag{1}$$

$$\mathsf{wait}(a; \mathsf{wait}(b; x)) = \mathsf{wait}(a \oplus b; x) \tag{2}$$

$$\mathsf{wait}(a; x(b)) = \mathsf{wait}(a; x(a \oplus b)) \tag{3}$$

An algebraic theory of dynamic threads

Interaction of wait with the semilattice structure of thread IDs.

$$\mathsf{wait}(0; x) = x \tag{1}$$

$$\mathsf{wait}(a; \mathsf{wait}(b; x)) = \mathsf{wait}(a \oplus b; x) \tag{2}$$

$$\mathsf{wait}(a; x(b)) = \mathsf{wait}(a; x(a \oplus b)) \tag{3}$$

The term $\mathsf{wait}(a;\ \mathsf{stop})$ acts as a unit for fork.

$$\mathsf{fork}(a.\mathsf{wait}(a; \mathsf{stop}), x) = x \tag{4}$$

$$\mathsf{fork}(b.x(b), \mathsf{wait}(a; \mathsf{stop})) = x(a) \tag{5}$$

An algebraic theory of dynamic threads

Interaction of wait with the semilattice structure of thread IDs.

$$\mathsf{wait}(0; x) = x \tag{1}$$

$$\mathsf{wait}(a; \mathsf{wait}(b; x)) = \mathsf{wait}(a \oplus b; x) \tag{2}$$

$$\mathsf{wait}(a; x(b)) = \mathsf{wait}(a; x(a \oplus b)) \tag{3}$$

The term $\mathsf{wait}(a; \mathsf{stop})$ acts as a unit for fork.

$$\mathsf{fork}(a.\mathsf{wait}(a; \mathsf{stop}), x) = x \tag{4}$$

$$\mathsf{fork}(b.x(b), \mathsf{wait}(a; \mathsf{stop})) = x(a) \tag{5}$$

Operations wait and fork commute; fork is commutative and associative.

$$\mathsf{wait}(b;\ \mathsf{fork}(a.x(a),\ y)) = \mathsf{fork}(a.\mathsf{wait}(b;\ x(a)),\ \mathsf{wait}(b;\ y)) \tag{6}$$

$$\mathsf{fork}(a.\mathsf{fork}(b.x(a,b),\ y),\ z) = \mathsf{fork}(b.\mathsf{fork}(a.x(a,b),\ z),\ y) \tag{7}$$

$$\mathsf{fork}(a.x(a),\ \mathsf{fork}(b.y(b),\ z)) = \mathsf{fork}(b.\mathsf{fork}(a.x(a),\ y(b)),\ z) \tag{8}$$

$$\mathsf{act}_\sigma(x) = \mathsf{fork}(a.\mathsf{wait}(a, x),\ \mathsf{act}_\sigma(\mathsf{stop})) \tag{9}$$

Outline

The pomset model

Main theorem

Terms in the algebraic theory of dynamic threads correspond exactly to (isomorphism classes of) “pomsets with holes”.

Put differently:

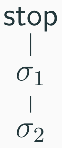
- Pomsets with holes form a free model for the algebraic theory of dynamic threads.
- The equations of the algebraic theory are **sound and complete** w.r.t. equality of pomsets with holes.

The proof uses a normalization-by-evaluation argument.

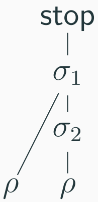
Closed terms denote pomsets

Pomset = partial order labelled by observable actions e.g. σ.
The partial order encodes observable dependencies.

$\mathsf{fork}\big(a.\mathsf{wait}(a;\ \mathsf{act}_{\sigma_1}(\mathsf{stop})),\ \mathsf{act}_{\sigma_2}(\mathsf{stop})\big)$



$\mathsf{fork}\big(a.\mathsf{act}_{\rho}(\mathsf{wait}(a;\ \mathsf{act}_{\sigma_1}(\mathsf{stop}))),\ \mathsf{act}_{\rho}(\mathsf{act}_{\sigma_2}(\mathsf{stop}))\big)$



What about terms with free variables (i.e. continuations) and free tid's? Example:
$x : 1 \mid a \vdash \mathsf{fork}(b.\mathsf{wait}(a;\ x(b)),\ \mathsf{act}_{\rho}(\mathsf{stop}))$

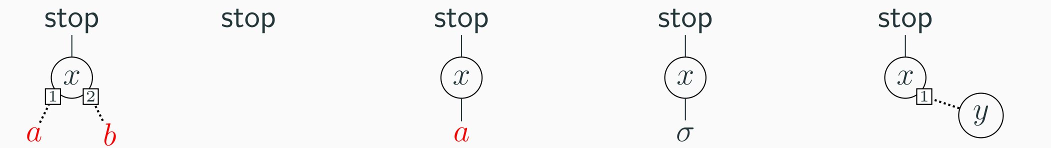
Pomsets with holes

Pomsets with extra structure:

- minimal elements representing free thread IDs (in red);
- elements labelled by variables (the holes);
- a relation of potential dependencies (dotted lines).

Terms denote pomsets. Algebraic operations act on pomsets.

$x(a, b)$	stop	$\mathsf{wait}(a;\ x)$	$\mathsf{act}_\sigma(x)$	$\mathsf{fork}(a.x(a),\ y)$

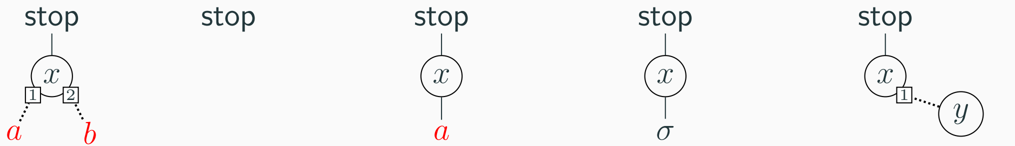


Define substitution of another pomset for all holes labelled x (monadic bind). The dotted lines become important.

Pomsets with holes

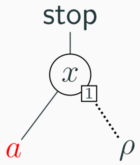
Terms denote pomsets. Algebraic operations act on pomsets.

$x(a, b)$	stop	wait$(a;\ x)$	act$_\sigma(x)$	fork$(a.x(a),\ y)$

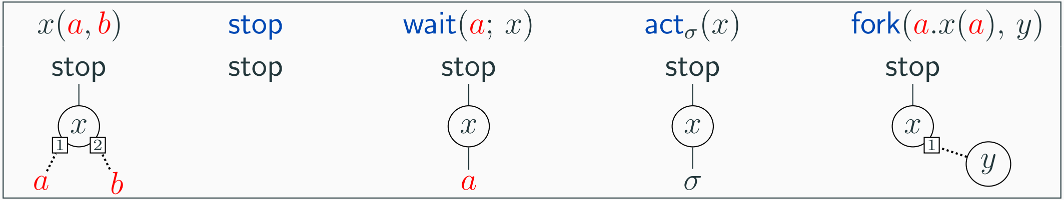


Example denotation:

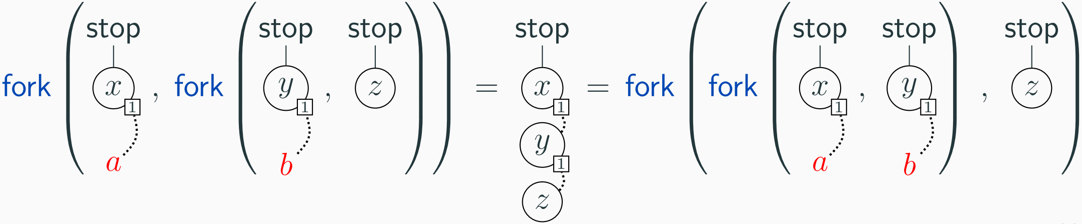
$$x : 1 \mid a \vdash \text{fork}(b.\text{wait}(a;\ x(b)),\ \text{act}_\rho(\text{stop}))$$



Reasoning graphically about the equations in the algebraic theory



$$\mathsf{fork}(a.x(a),\ \mathsf{fork}(b.y(b),\ z)) = \mathsf{fork}(b.\mathsf{fork}(a.x(a),\ y(b)),\ z) \quad (8)$$



Further results: adequacy and full abstraction at first order

The pomset model of the algebraic theory induces a **monad** on $\mathsf{Set}^{\mathsf{FinRel}}$ via the free algebra construction.

Use the monad to give a **denotational model** to a simply-typed lambda calculus with fork, wait, stop, act_σ.

The denotational semantics matches the operational semantics:

Theorems

Denotational equality implies contextual equivalence.
The converse is true for programs of first-order type.

Summary and more related work

Modelled thread creation (fork) and synchronization (wait) using a **parameterized** algebraic theory, and the induced monad of pomsets.

Parameterized theories have been used for other effects with binding e.g.:

- local state
- functional logic programming
- qubits

[Staton]

- urns in probabilistic programming [Staton et al'18]
- code pointers [Fiore&Staton'14]
- scoped effects [Matache et al'25]

Idris library for **handlers** for parameterized effects (Paella) [Sigal et al'HOPE 24].

A language for parameterized handlers (Cambria) [Liell-Cock and Staton].

Future work

- Model other types of concurrency implemented with effect handlers.
- Combine with other algebraic effects e.g. state.
- Give more structure to the observable actions, σ.
- Model more features of POSIX `fork`.
- Model communication between parent and child using conflict (as in event structures):

Example: $\underline{\text{stop}} : \text{bool} \to \text{empty}$ $\underline{\text{wait}} : \text{tid} \to \text{bool}$

$$\text{wait}(a, \text{act}_{\sigma_1}, \text{act}_{\sigma_2})$$

$$\begin{tikzcd} \sigma_1 \arrow[rr, no head, squiggly] \arrow[dr, no head] & & \sigma_2 \arrow[dl, no head] \\ & a & \end{tikzcd}$$