An Equational Axiomatization of Dynamic Threads:

Towards Concurrency as an Algebraic Effect

Cristina Matache

MFPS, 2 June 2026

University of Birmingham

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

Introduction

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

These methods work well for functional programming: a program is a function that takes an input and returns a value.

But programs interact with the environment e.g.:

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

} **Computational effects**

[Moggi], [Plotkin&Power]

Programs interact with other programs running **concurrently**.

In this talk:

Concurrency can be treated as a computational effect.

To obtain equational reasoning in a programming language setting.

Related work: ACP, concurrent Kleene algebra, [van Glabbeek&Plotkin], [Abadi&Plotkin]

Outline

Moggi's setting for denotational semantics of effects [LICS'89]

- A cartesian category $\mathcal{C}$ (binary products, terminal object),
- a strong monad T on $\mathcal{C}$, and
- Kleisli exponentials (objects $B \Rightarrow TC$).

Enough to model simply-typed lambda-calculus (STLC) with effects.

Types and contexts are interpreted as objects $[\![A]\!]$, $[\![\Gamma]\!]$ of $\mathcal{C}$.

Effectful terms are interpreted as morphisms:

$$[\![\Gamma \vdash^{\mathsf{eff}} M : A]\!] : [\![\Gamma]\!] \to T[\![A]\!]$$

Moggi's setting for denotational semantics of effects [LICS'89]

- A cartesian category $\mathcal{C}$ (binary products, terminal object),
- a strong monad T on $\mathcal{C}$, and
- Kleisli exponentials (objects $B \Rightarrow TC$).

Enough to model simply-typed lambda-calculus (STLC) with effects.

Example of a strong monad (on **Set**):

$$TX = 2 \Rightarrow (X \times 2)$$

which models reading and writing one bit of memory, and returning a value of type X.

Plotkin's and Power's framework of algebraic effects

Idea: effects are realised by operations and equations that **generate** a monad, but are not identified with the monad.

They work with presentations of algebraic theories, in the sense of universal algebra.

Theorem [Lawvere, Linton]

To give a finitary monad on the category of sets is equivalent to giving an algebraic theory.
(Here "algebraic theory" is a suitable presentation-independent notion.)

An algebraic theory determines a monad by the free-algebra construction.

Plotkin's and Power's framework of algebraic effects

Idea: effects are realised by operations and equations that **generate** a monad, but are not identified with the monad.

Example. The one-bit state monad $TX = 2 \Rightarrow (X \times 2)$ is generated by operations:

$$\frac{\Gamma \vdash^{\mathsf{eff}} M_0 : A \qquad \Gamma \vdash^{\mathsf{eff}} M_1 : A}{\Gamma \vdash^{\mathsf{eff}} \mathsf{get}(M_0, M_1) : A} \qquad \frac{\Gamma \vdash^{\mathsf{eff}} M : A}{\Gamma \vdash^{\mathsf{eff}} \mathsf{put}_0(M) : A} \qquad \frac{\Gamma \vdash^{\mathsf{eff}} M : A}{\Gamma \vdash^{\mathsf{eff}} \mathsf{put}_1(M) : A}$$

and equations, where $i, i' \in \{0, 1\}$

$$\mathsf{put}_i(\mathsf{get}(x_0, x_1)) = \mathsf{put}_i(x_i) \qquad \mathsf{put}_i(\mathsf{put}_{i'}(x)) = \mathsf{put}_{i'}(x)$$

$$\mathsf{get}(\mathsf{put}_0(x), \mathsf{put}_1(x)) = x$$

Outline

What I mean by concurrency in this talk

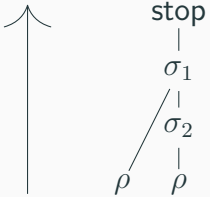
Dynamic threads

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:

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

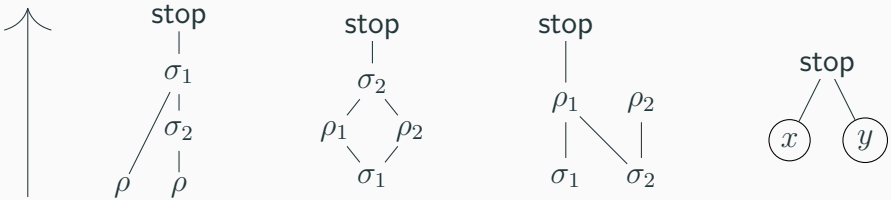


- **Labels** denote observable actions.
- **Partial order** denotes observable dependencies.

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]. (In this work, pomsets without the conflict of event structures.)

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;
- 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

Idea: treat forking threads and waiting for them as operations in the framework of algebraic effects.

(1) A natural operational semantics.

Summary of this work

Idea: treat forking threads and waiting for them as operations in the framework of algebraic effects.

(1) A natural operational semantics.

(2) An algebraic theory that matches the operational semantics.

Summary of this work

Idea: treat forking threads and waiting for them as operations in the framework of algebraic effects.

(1) A natural operational semantics.

(2) An algebraic theory that matches the operational semantics.

(3) **Main theorem:** characterize the free models of this algebraic theory using pomsets.
$\Longrightarrow$ A monad for denotational semantics of STLC + fork, wait.

Summary of this work

Idea: treat forking threads and waiting for them as operations in the framework of algebraic effects.

(1) A natural operational semantics.

(2) An algebraic theory that matches the operational semantics.

(3) **Main theorem:** characterize the free models of this algebraic theory using pomsets.
$\Longrightarrow$ A monad for denotational semantics of STLC + fork, wait.

(4) Prove the denotational semantics matches the operational semantics. (With the help of the algebraic presentation of the monad.)

Outline

Calculus

Call-by-value lambda calculus extended with:

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

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

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

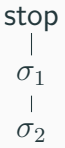
Operational semantics by example

Operational semantics is a relation between pools of threads.

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

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

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



stop is extra structure, a maximal element.

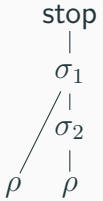
Operational semantics by example

Operational semantics is a relation between pools of threads.

fork duplicates the continuation.

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

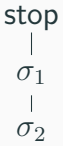
$$\longrightarrow \big\{[b]\mathsf{case}\ \big(\mathsf{act}_\rho(); \mathsf{inj}_1(a)\big)\ \mathsf{of}\ \ldots,$$
$$[a]\mathsf{case}\ \big(\mathsf{act}_\rho(); \mathsf{inj}_2()\big)\ \mathsf{of}\ \ldots\big\}$$



Operational semantics by example

stop discards the continuation.

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



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$(a.x(a),\ y)$ $\quad$ fork : unit $\to$ tid $+$ unit

wait$(u;\ x)$ $\quad$ wait : tid $\to$ unit

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

stop $\quad$ stop : unit $\to$ empty

has no continuation

act$_\sigma(x)$ $\quad$ 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 wait$(a;$ 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:

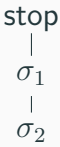
- Pomsets with holes form a free model for the algebraic theory of dynamic threads.
- The equations of the algebraic theory are **sound and complete** w.r.t. equality of pomsets with holes.

The proof uses a normalization-by-evaluation argument.

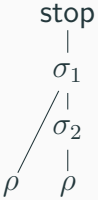
Closed terms denote pomsets

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

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



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



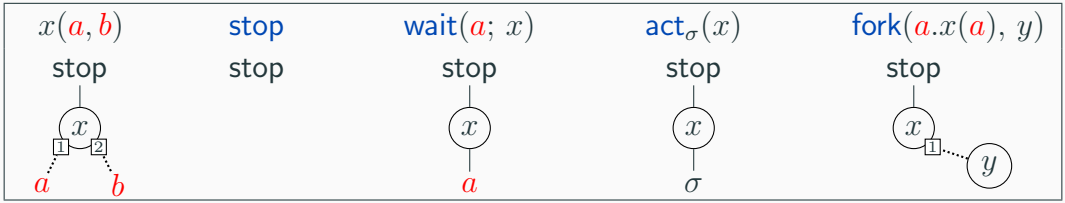
What about terms with free variables (i.e. continuations) and free tid's? Example:
$x : 1 \mid a \vdash \mathsf{fork}(b.\mathsf{wait}(a;\ x(b)),\ \mathsf{act}_{\rho}(\mathsf{stop}))$

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).

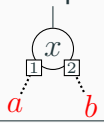
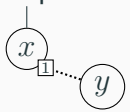
Terms denote pomsets. Algebraic operations act on pomsets.



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

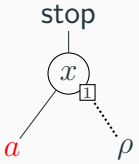
Pomsets with holes

Terms denote pomsets. Algebraic operations act on pomsets.

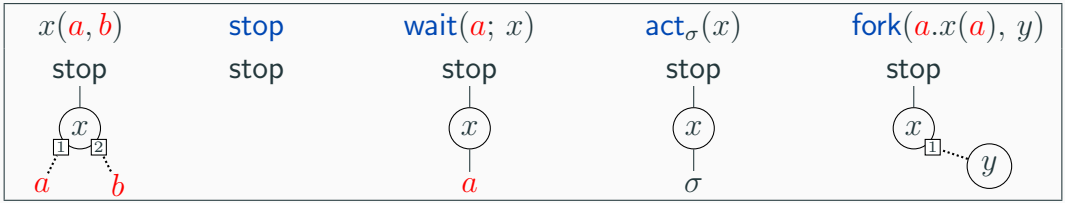
$x(a, b)$	stop	wait$(a;\ x)$	act$_\sigma(x)$	fork$(a.x(a),\ y)$
stop	stop	stop	stop	stop
				

Example denotation:

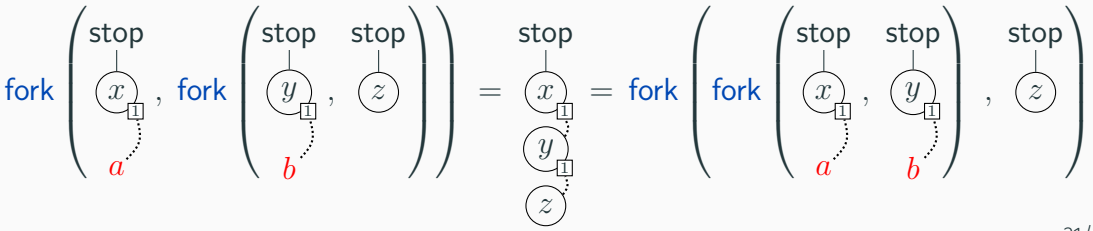
$x : 1 \mid a \vdash \mathsf{fork}(b.\mathsf{wait}(a;\ x(b)),\ \mathsf{act}_\rho(\mathsf{stop}))$



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

Framework of algebraic effects: 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:

Theorem

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

Outline

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]

Ongoing work on **handlers** for parameterized effects (Paella) [Sigal et al'HOPE 24].

Future work

Advantage of algebraic effects: study the concurrent behaviour independently of the programming language.

Towards a theory of concurrent programming with effects

- Can we combine with other algebraic effects e.g. state?
- Can we model other types of concurrency implemented with handlers?
- Do we need new notions of algebraic theories?
- Can we get useful equational reasoning principles?

Future work

- Give more structure to the observable actions, σ.
- Model more sophisticated concurrency:
  - forking might fail (like in POSIX);
  - communication between parent and child;
  - allowing conflicts between actions (as in event structures).

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

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