# An Equational Axiomatization of Dynamic Threads via Algebraic Effects

Ohad Kammar[†], Jack Liell-Cock[‡], Sam Lindley[†], Cristina Matache[†], and Sam Staton[‡]

POPL, 16 January 2026

[†]University of Edinburgh

[‡]University of Oxford

# In this work:

## Goal

Denotational semantics for **concurrency** where new threads can be created dynamically. E.g. POSIX-like `fork`.

## Main ideas

**IDs** of pools of threads are abstract names.
`fork` is an algebraic effect.

## Contribution

True concurrency semantics with **complete equational** reasoning applied to a core functional programming language.

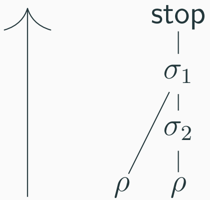
Our semantics uses: strong monads [Moggi'91], algebraic theories [Plotkin&Power].

# The denotational semantics at a glance

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

Example:

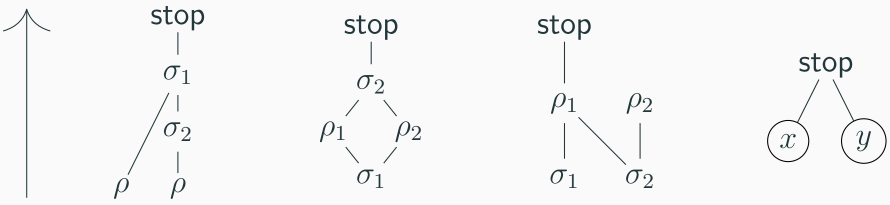
$$\text{let } y = \underline{\mathsf{fork}}() \text{ in case } \left(\underline{\mathsf{act}}_\rho(); y\right)$$
$$\text{of } \{ \ \mathsf{inj}_1(a) \Rightarrow \underline{\mathsf{wait}}(a); \underline{\mathsf{act}}_{\sigma_1}(); \underline{\mathsf{stop}}()$$
$$\mathsf{inj}_2() \Rightarrow \underline{\mathsf{act}}_{\sigma_2}();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




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

# The denotational semantics at a glance

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




Pomsets are a long-established model of true concurrency
e.g. [Nielsen et al'81], [Pratt'86].

## Main Theorem

A **complete equational axiomatization** for our pomset semantics via algebraic effects.

# Related work

- process algebra;
- concurrent Kleene algebra [Hoare et al'11];
- algebraic effects for concurrency [Stark'08], [van Glabbeek&Plotkin'10], [Abadi&Plotkin'10], [Dvir et al'22, '25];
- separation logic for effect handlers [de Vilhena et al'21, POPL'26];
- ...

**Key contribution: true concurrency** semantics for a core functional programming language, obtained via an **equational axiomatization**.

# Outline

# Effects we model

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

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

| | |
|---|---|
| tid | base type of IDs of thread pools; 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 |
| $\text{act}_\sigma$() | performs observable action $\sigma$ immediately |

# Effects we model

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

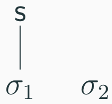
$$\text{act}_\sigma : \text{unit} \to \text{unit}$$

- An idealized version of POSIX-like `fork` and `wait`.
- Use a fine-grain call-by-value lambda calculus with these effectful operations.
- Operational semantics based on pools of threads.

# Example programs

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

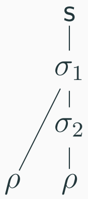
$$\text{let } y = \underline{\text{fork}}() \text{ in case } y \text{ of } \{\text{inj}_1(a) \Rightarrow \underline{\text{act}}_{\sigma_1}(); \underline{\text{stop}}() \quad \text{inj}_2() \Rightarrow \underline{\text{act}}_{\sigma_2}(); \underline{\text{stop}}()\}$$




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

# Example programs

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




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


We will axiomatize fork, wait, stop, $\text{act}_\sigma$ with 9 equations.

# Outline

# Algebraic operations vs generic effects

tid $=$ type of IDs of thread pools, has a semilattice structure. Examples:

- $a \oplus b$ : tid is the ID of the union of the thread pools $a$ and $b$
- 0 : tid is the empty thread pool.

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

Example:

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

fork binds a new ID $a$, bound in $x$. $a$ refers to the singleton pool $y$.

$a$ is a parameter in a parameterized algebraic theory [Staton'13].

# Algebraic operations vs generic effects

tid = type of IDs of thread pools, has a semilattice structure.

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

fork binds a new ID $a$, bound in $x$. $a$ refers to the singleton pool $y$.

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

$u$ is the ID of a thread pool; 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 $\sigma$ and continues as $x$

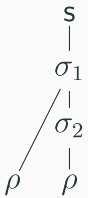
# Rewriting the example programs using algebraic operations

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


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

---

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




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

# 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

# Main result: a representation theorem for the theory of threads

## Completeness Theorem

Terms in the algebraic theory of dynamic threads correspond exactly to "labelled partial orders (pomsets) with holes".

The equations of the algebraic theory are **sound and complete** w.r.t. equality of pomsets with holes.

This theorem lets us reason **graphically** about the algebraic theory.

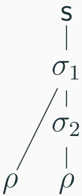
Later, we relate with the operational semantics based on pools of threads.

# Labelled partial orders (pomsets)

So far we only represented closed terms:

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


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

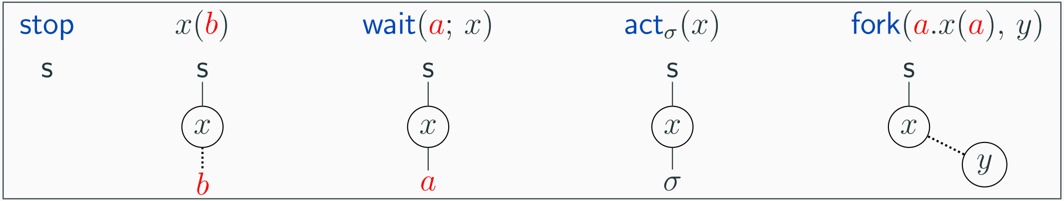



What about terms with free variables (i.e. continuations) and free tid's?

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

# Labelled partial orders (pomsets) with holes

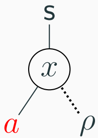
The holes and dotted lines are extra structure on a pomset:



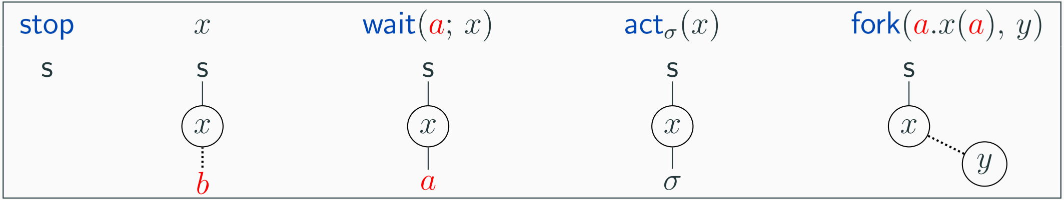

Define substitution of another pomset for all holes labelled $x$ (monadic bind). The dotted lines become important.

Example denotation:

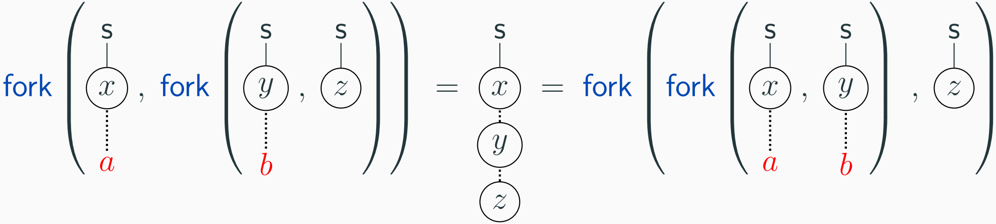
$$a \vdash \mathsf{fork}(b.\mathsf{wait}(a;\ x(b)),\ \mathsf{act}_\rho(\mathsf{stop}))$$



# 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

We already saw **completeness**: equality in the algebraic theory agrees with equality of pomsets with holes.

Therefore, pomsets with holes form a **monad** which we use to give a **denotational semantics** to a core language with fork, wait, stop, $\text{act}_\sigma$.

The denotational semantics matches the operational semantics:

**Theorem**

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

# Summary and future work

Denotational semantics for concurrent programs that can fork new threads and wait for them. Thread IDs are key.

Main theorems:

- **complete equational axiomatization** of the "pomsets with holes" semantics for a core functional programming language;
- adequacy, full abstraction at first order w.r.t. operational semantics.

Future work:

- Passing values from child to parent: stop : $A \to$ empty, wait : tid $\to A$.
- Combine with shared state.
- Explore alternative semantics for fork and wait.