An algebraic theory of named threads

Ohad Kammar **Jack Liell-Cock** Sam Lindley Cristina Matache Sam Staton

Oxford Computer Science Conference 2025

Denotational semantics

- Formally define the meaning of programming languages by assigning programs to mathematical objects
- Allows the unambiguous specification of programs, facilitiates language design, and allows program verification and reasoning

$$\begin{array}{l} \textbf{function } f(x)\ \{ \\ \quad x = x + 2; \\ \quad \text{return } x; \\ \} \end{array} \quad \rightsquigarrow \quad \begin{array}{r} f : \text{Integer} \to \text{Integer} \\ x \mapsto x + 2 \end{array}$$

Algebraic effects

- What if we print to screen, access memory, or invoke randomness?

$$
\begin{array}{l}
\textbf{function } f(x)\ \{ \\
\quad \text{print}\,(\text{"Hello World!"}); \\
\quad x = x + 2; \\
\quad \text{return}\ \ x; \\
\}
\end{array}
\quad \rightsquigarrow \quad f : ???
$$

Algebraic effects

- What if we print to screen, access memory, or invoke randomness?
- **Computational effects** can be modelled with **strong monads** [Moggi'91] and **algebraic theories** [Plotkin & Power]

$$\begin{array}{l} \textbf{function } f(x)\ \{ \\ \qquad \text{print}\,(\text{"Hello World!"}); \\ \qquad x = x + 2; \\ \qquad \text{return } x; \\ \} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} f : \text{Integer} \to T(\text{Integer}) \\ \qquad x \mapsto \mathsf{p}_{\text{Hello World!}}(x+2) \end{array}$$

Contextual equivalence

- We would like our denotations to capture **contextual equivalence**
- **Soundness**: Contextual equivalence implies equal denotations
- **Adequacy**: Equal denotations implies contextual equivalence
- **Full abstraction**: Both soundness and adequacy

function $f(x)$ $\{$

$x = x + 2;$

return $x;$

$\}$

function $f(x)$ $\{$

return $x + 2;$

$\}$

Contextual equivalence

- We would like our denotations to capture **contextual equivalence**
- **Soundness**: Contextual equivalence implies equal denotations
- **Adequacy**: Equal denotations implies contextual equivalence
- **Full abstraction**: Both soundness and adequacy

function $f(x)$ {
    $x = x + 2$;
    return x;
}

function $f(x)$ {
    return $x + 2$;
}

Can be distinguished by the program:

$$x = 2;\ f(x);\ \ \text{print}\,(x);$$

What about dynamic effects?

Effects that dynamically allocate resources (e.g. local state) need more sophisticated algebraic theories/monads. [Plotkin & Power'02], [Power'06], [Melliès'10,'14], [Staton'13]

Question

Can concurrency (forking threads and waiting for them) be axiomatized as a local algebraic effect?

Ongoing work using **parameterized algebraic theories** [Staton'13].

Outline

1. Parameterized algebraic theories
2. Parameterized theory of named threads
3. Operational semantics for named threads

Parameterized algebraic theories [Staton FOSSACS'13, LICS'13, POPL'15]

- Uniform framework for axiomatizing local effects:

Example	Parameters
local state	location names

$\mathsf{read}(a, x, y)$ read the bit stored in location a and continue as either x or y
a is a free parameter

$\mathsf{new}_0(a.x(a))$ create a new location a
a is a fresh parameter, bound in x

and other operations and equations...

- Extend algebraic theories by allowing binding of abstract parameters.

Parameterized algebraic theories [Staton FOSSACS'13, LICS'13, POPL'15]

- Uniform framework for axiomatizing local effects:

Example	Parameters
local state	location names
π-calculus (fragment)	communication channels
program jumps	code pointers
quantum computation	qubits

- Extend algebraic theories by allowing binding of abstract parameters.
- Correspondence to monads on a functor category.

π-calculus (fragment): does not contain parallel composition as an operation

see also [Stark'08], [van Glabbeek & Plotkin'10]

Outline

1 Parameterized algebraic theories

2 Parameterized theory of named threads

3 Operational semantics for named threads

Parameterized theory of named threads

Parameters = thread IDs

Operations:

$\mathsf{fork}(a.x(a),\, y)$	x = parent thread; variable standing for another term y = child thread; variable a = ID of child; bound name that x can use, but y can't
$\mathsf{wait}(a,\, x)$	wait for the thread named a to finish, continue as x
stop	this thread has finished (no continuation)
$\mathsf{print}_s(x)$	print s (observable behaviour), continue as x

Forking and waiting are similar to the ones in Unix.

Equations for the parameterized theory of threads

$$y : 0 \mid - \vdash \mathsf{fork}(a.x,\ \mathsf{stop}) = x$$

Equations for the parameterized theory of threads

$$y : 0 \mid - \vdash \mathsf{fork}(a.x,\ \mathsf{stop}) = x \qquad x : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{stop}), x) = x$$

Equations for the parameterized theory of threads

$$y : 0 \mid - \vdash \mathsf{fork}(a.x, \mathsf{stop}) = x \qquad x : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{stop}), x) = x$$

And many more:

$$x : 1 \mid b \vdash \mathsf{fork}(a.x(a), \mathsf{wait}(b, \mathsf{stop})) = x(b)$$

$$x : 1, y : 0 \mid b \vdash \mathsf{wait}(b, \mathsf{fork}(a.x(a), y)) = \mathsf{fork}(a.\mathsf{wait}(b, x(a)), \mathsf{wait}(b, y))$$

$$x : 0 \mid a, b \vdash \mathsf{wait}(a, \mathsf{wait}(b, x)) = \mathsf{wait}(a \oplus b, x)$$

$$x : 0 \mid - \vdash \mathsf{wait}(\bot, x) = x$$

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

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

$$x : 0 \mid - \vdash \mathsf{print}_\alpha(x) = \mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{print}_\alpha(\mathsf{stop}))$$

$$x : 1 \mid a, b \vdash \mathsf{wait}(a, x(b)) = \mathsf{wait}(a, x(a \oplus b))$$

Equations for the parameterized theory of threads

$$y : 0 \mid - \vdash \mathsf{fork}(a.x,\ \mathsf{stop}) = x \qquad\qquad x : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{stop}), x) = x$$

And many more:

$$x : 1 \mid b \vdash \mathsf{fork}(a.x(a), \mathsf{wait}(b, \mathsf{stop})) = x(b)$$

$$x : 1, y : 0 \mid b \vdash \mathsf{wait}(b, \mathsf{fork}(a.x(a), y)) = \mathsf{fork}(a.\mathsf{wait}(b, x(a)), \mathsf{wait}(b, y))$$

$$x : 0 \mid a, b \vdash \mathsf{wait}(a, \mathsf{wait}(b, x)) = \mathsf{wait}(a \oplus b, x)$$

$$x : 0 \mid - \vdash \mathsf{wait}(\perp, x) = x$$

$$\cdots$$

Goal

Compare the equations with an operational semantics.

Outline

1 Parameterized algebraic theories

2 Parameterized theory of named threads

3 Operational semantics for named threads

Operational semantics for named threads

Configuration S = a set of running (named) threads

Labels = printed symbols

Labelled transition system:

$$S \uplus \{[a]\mathsf{fork}(b.t_1,\ t_2)\} \to S \uplus \{[a]t_1,\ [b]t_2\} \qquad b \text{ fresh}$$
$$S \uplus \{[a]\mathsf{wait}(b, t),\ [b]\mathsf{stop}\} \to S \uplus \{[a]t,\ [b]\mathsf{stop}\}$$
$$S \uplus \{[a]\mathsf{print}_s(t)\} \xrightarrow{s} S \uplus \{[a]t\}$$

Terms t_1 and t_2 are **contextually equivalent** if they have the same sets of traces in all contexts.

Contextual equivalence of named threads

The following terms have the same trace:

$$t_1 = \mathsf{fork}(a.\mathsf{stop},\ \mathsf{print}_1(\mathsf{stop})) \qquad \{1\}$$

$$t_2 = \mathsf{print}_1(\mathsf{stop}) \qquad \{1\}$$

But they are not **contextually equivalent**:

$$C = \mathsf{fork}(b.\mathsf{wait}(b,\ \mathsf{print}_2(\mathsf{stop})),\ \Box)$$

$$C[t_1] \qquad \{21,\ 12\}$$

$$C[t_2] \qquad \{12\}$$

Contextual equivalence is hard because of the quantification over all contexts.

Results

Theorem 1

The algebraic theory is **adequate** with respect to contextual equivalence

Theorem 2

The corresponding monad generalises pomsets (common causal semantics for concurrency)

Summary

This work is about:

- axiomatizing Unix fork and wait
- as an algebraic theory, parameterized by thread ID's
- and comparing to an operational semantics

Results:

- The algebra is adequate with respect to the operational semantics
- The corresponding monad generalises current semantic approaches

Future work:

- Refine semantics to get soundness of the theory
- Build a first-order programming language with concurrency as an effect