An Equational Axiomatization of Dynamic Threads via Algebraic Effects

Presheaves on finite relations, labelled posets, and parameterized algebraic theories

OHAD KAMMAR, University of Edinburgh, UK
JACK LIELL-COCK, University of Oxford, UK
SAM LINDLEY, University of Edinburgh, UK
CRISTINA MATACHE, University of Edinburgh, UK
SAM STATON, University of Oxford, UK

We use the theory of algebraic effects to give a complete equational axiomatization for dynamic threads. Our method is based on parameterized algebraic theories, which give a concrete syntax for strong monads on functor categories, and are a convenient framework for names and binding.

Our programs are built from the key primitives 'fork' and 'wait'. 'Fork' creates a child thread and passes its name (thread ID) to the parent thread. 'Wait' allows us to wait for given child threads to finish. We provide a parameterized algebraic theory built from fork and wait, together with basic atomic actions and laws such as associativity of 'fork'.

Our equational axiomatization is complete in two senses. First, for closed expressions, it completely captures equality of labelled posets (pomsets), an established model of concurrency: model complete. Second, any two open expressions are provably equal if they are equal under all closing substitutions: syntactically complete.

The benefit of algebraic effects is that the semantic analysis can focus on the algebraic operations of fork and wait. We then extend the analysis to a simple concurrent programming language by giving operational and denotational semantics. The denotational semantics is built using the methods of parameterized algebraic theories and we show that it is sound, adequate, and fully abstract at first order for labelled-poset observations.

1 INTRODUCTION

The theory of algebraic effects provides a way of analyzing semantic aspects of different computational effects in isolation, and separately from other aspects of programming languages, via the algebraic theories from universal algebra. This paper provides an analysis of concurrency using the methods of algebraic effects.

A theory of algebraic effects for concurrency has proved elusive [32, 46]. This is in spite of the success of equational and compositional reasoning in process algebra [5, 15, 25, 38], and equational theories of concurrency such as concurrent Kleene algebra [13, 14]. Even more paradoxically, algebraic effects have already inspired powerful concurrency libraries [30, 39], but these software implementations do not yet tie with the theories of algebraic effects in terms of universal algebra and category theory. (See §8 for further discussion of the literature.)

The key technique in our work is *to take thread IDs seriously*. This necessitates an algebraic framework that supports abstract names or IDs, and binding and passing them. For this, we use 'parameterized algebraic theories' [41, 42], which already have a tight connection with monads and algebraic effects. There are four operations in our algebraic theory:

- fork: Forking a child thread. This is the key operation and is written $\mathsf{fork}(a.x(a), y)$. This spawns a new child thread with ID a, running continuation y, while concurrently running continuation x in the parent thread, which is passed the ID a of the child.
- wait: A command to wait for a thread to end before proceeding.

Authors' addresses: Ohad Kammar, ohad.kammar@ed.ac.uk, University of Edinburgh, UK; Jack Liell-Cock, jack.liell-cock@keble.ox.ac.uk, University of Oxford, UK; Sam Lindley, sam.lindley@ed.ac.uk, University of Edinburgh, UK; Cristina Matache, cristina.matache@ed.ac.uk, University of Edinburgh, UK; Sam Staton, sam.staton@cs.ox.ac.uk, University of Oxford, UK.

- stop: A command to end the current thread now.
- act_σ: Primitive atomic actions. Aside: going forward, we could combine with other algebraic effects, such as memory access to look at concurrent shared memory, but for now to focus on concurrency we restrict attention to primitive atomic actions.

Contributions. We present a theory with eight equations between these four operations (§4). We give a syntax-free representation theorem of terms modulo equations (§5), and show that for closed terms, the representation exactly matches the long-established model of true concurrency based on labelled posets ('pomsets', Thm. 3.15). In fact, this might be the first basic syntactic theory for labelled posets. For open terms with free variables, we prove a completeness theorem: there can be no further equations on open terms while retaining the labelled posets model on closed terms (§6).

Algebraic effects allow us to focus on a particular theory, without worrying about other programming language primitives, but it is typically easy to return to a fuller programming language having analyzed the algebraic effects. In Section 2, we give a typical functional programming language with concurrency primitives and an operational semantics. In Section 7 we use the algebraic effects and the representation theorem to build a denotational semantics for the programming language that is sound, adequate, and fully abstract at first order.

1.1 Motivating Fork and Wait with Thread IDs as Language Primitives

The style of programming with operations such as $\mathsf{fork}(a.x(a), y)$ is unusual, but according to the theory of algebraic effects, algebraic operations have a counterpart in generic effects [33], and this matches more closely to realistic languages with effects. The generic effect for 'fork' is a command $\underline{\mathsf{fork}} : \mathsf{unit} \to (\mathsf{tid}\,\mathsf{option})$,

$$\underline{\mathsf{fork}}() = \mathsf{fork}(a.\mathsf{return}\,(\mathsf{Some}\,a), \mathsf{None}).$$

We provide a mini-programming language with an operational semantics in Section 2, which works with pools of threads. There, $\underline{\mathsf{fork}}$ will spawn a new child thread into the thread pool, and the continuation is duplicated. The caller of $\underline{\mathsf{fork}}$ can check whether they are the parent or child by looking at the return value of $\underline{\mathsf{fork}}$, and if they are the parent they will be given the ID of the child, otherwise None. Indeed this generic operation $\underline{\mathsf{fork}}$ is reminiscent of the POSIX `fork` construct [1], which is typed `pid_t fork(void)`, which returns the child ID to the parent and 0 to the child.

Alongside the standard programming primitives, our other generic effects are

$$\underline{\mathsf{wait}} : \mathsf{tid} \to \mathsf{unit} \qquad \underline{\mathsf{perform}}_\sigma : \mathsf{unit} \to \mathsf{unit} \qquad \underline{\mathsf{stop}} : \mathsf{unit} \to \mathsf{empty}.$$

Here: $\underline{\mathsf{wait}}(a)$ puts the current thread into a waiting state, recording for the scheduler the thread a that it is waiting for; $\underline{\mathsf{perform}}_\sigma()$ performs the action σ immediately, which is recorded by a label in our transition system; and $\underline{\mathsf{stop}}()$ ends the current thread, unblocking all other threads that were waiting for it. Here, the type empty shows that nothing else will happen on this execution path.

Our operational semantics uses a labelled transition system that records the actions performed. Inspired by true concurrency models such as asynchronous transition systems (e.g. [27]), we also include some location information, by way of noting the ID of the thread that performed each action. In this simple situation, this is sufficient to observe not only the traces of actions but also the independence between different actions. We can thus, from the operational semantics, obtain a labelled partial order, labelled by actions σ. Labelled posets, sometimes called 'pomsets', are another model of 'true' concurrency [37]. For our semantics, the linearizations of the posets are exactly the execution traces of the program.

By defining 'well-formed configurations' for our particularly simple language, we can show that programs never deadlock, roughly because a child can never wait for its parent. We also show that every closed program determines a unique finite-labelled poset. This clarifies that our

language is very simple, in that programs all terminate, and there is no 'conflict' in the sense of event structures [28], nor are there any 'races'. These are useful properties to have, and also useful for later relating to denotational semantics. As we show, the language is still powerful enough to describe concurrency situations, and we expect future work to extend with other primitives that allow recursion and conflict.

1.2 A Simple Complete Fragment for Labelled Posets (Pomsets)

This simple language allows us to construct all labelled posets, in other words, it completely describes that model of true concurrency. To show this we define $\underline{\mathsf{node}}_\sigma : \mathsf{tid}\,\mathsf{set} \to \mathsf{tid}$ by

$$\underline{\mathsf{node}}_\sigma(a_1 \dots a_n) \stackrel{\text{def}}{=} \mathsf{case}\ \underline{\mathsf{fork}}()\ \mathsf{of}\ \{\mathsf{Some}\ b \Rightarrow \mathsf{return}\ b;\ \mathsf{None} \Rightarrow \underline{\mathsf{wait}}(a_1); \dots; \underline{\mathsf{wait}}(a_n); \underline{\mathsf{perform}}_\sigma(); \underline{\mathsf{stop}}\}.$$

In the $\underline{\mathsf{node}}_\sigma$-only fragment, every thread ID performs exactly one action. Thus the induced labelled poset is a partial order on thread IDs, recording which waits for which, each labelled with their action. The command $\underline{\mathsf{node}}_\sigma(a_1 \dots a_n)$ adds a node labelled σ to the labelled poset, setting its immediate predecessors to $\{a_1 \dots a_n\}$, and returns the name of the new node.

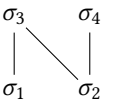


Fig. 1. N-shape poset

For example, the following program induces the N-shape poset (Fig. 1).

$$\mathsf{let}\ a_1 = \underline{\mathsf{node}}_{\sigma_1}([])\ \mathsf{in}\ \mathsf{let}\ a_2 = \underline{\mathsf{node}}_{\sigma_2}([])\ \mathsf{in}\ \mathsf{let}\ a_3 = \underline{\mathsf{node}}_{\sigma_3}([a_1, a_2])\ \mathsf{in}\ \mathsf{let}\ a_4 = \underline{\mathsf{node}}_{\sigma_4}([a_2])\ \mathsf{in}\ \underline{\mathsf{stop}}$$

We can completely axiomatize labelled posets by two axioms, written slightly informally for now (see Example 3.7):

$$\mathsf{let}\ c = \underline{\mathsf{node}}_\sigma(\vec{a})\ \mathsf{in}\ \mathsf{let}\ d = \underline{\mathsf{node}}_\sigma(\vec{b})\ \mathsf{in}\ (c, d) \quad = \quad \mathsf{let}\ d = \underline{\mathsf{node}}_\sigma(\vec{b})\ \mathsf{in}\ \mathsf{let}\ c = \underline{\mathsf{node}}_\sigma(\vec{a})\ \mathsf{in}\ (c, d)$$
$$\mathsf{let}\ b = \underline{\mathsf{node}}_\sigma(\vec{a})\ \mathsf{in}\ [b] \quad = \quad \mathsf{let}\ b = \underline{\mathsf{node}}_\sigma(\vec{a})\ \mathsf{in}\ (b, \vec{a})$$

The first law says that it does not matter in which order we add nodes, as long as ID dependencies are respected, and the second captures the transitivity of the partial order. We show that these two axioms are complete in Theorem 3.15, using the more formal framework of parameterized algebraic theories. The key point is that by passing around the thread IDs, we are able to fully describe the established model of true concurrency based on labelled posets.

1.3 Technical Setting: Functor Categories and Naturality for Syntax with Binding and Semantics with Names

To cope with the dynamic threads and varying number of thread names, we follow the long-standing tradition of using functor categories. In brief, let **FinRel** be the category of finite sets of names and relations between them, and let Set be the category of all sets and functions. Then the computations at some type A form a functor $[\![A]\!] : \mathbf{FinRel} \to \mathsf{Set}$, mapping a set w of available thread IDs to the set $[\![A]\!](w)$ of computations that use at most those thread IDs.

According to the functorial action here, for each relation $R \subseteq w \times w'$, we have a reindexing function $[\![A]\!](w) \to [\![A]\!](w')$. The idea is that if a computation in world w would wait for some thread ID $a \in w$, then we can transform it into one that instead waits for all the thread IDs in the direct image, $\{b \mid R(a, b)\}$.

Programs of type $A \to B$ are interpreted as families of functions $[\![A]\!](w) \to [\![B]\!](w)$ that are moreover natural. This, in particular, maintains the invariant that one cannot sum thread IDs, guess thread IDs, or compare them in some order. This is similar to the role of names in nominal sets [31].

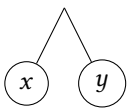
The framework of parameterized algebraic theories is an established method for algebraic effects over functor categories and admits a concrete syntax, which we use for our axiomatizations.

Moreover, every parameterized algebraic theory induces a strong monad on the functor category. Monads on functor categories have long been used for denotational semantics of dynamic allocation [26, 29, 35], and we use them for our denotational semantics (§7.1).

1.4 Fork and Wait in General, Parallel Composition, and Labelled Posets with Holes

Although the $\underline{\mathsf{node}}_\sigma$ effect is enough to build all labelled posets, it is more paradigmatic to allow higher level parallelism through $\underline{\mathsf{fork}}$ and $\underline{\mathsf{wait}}$. For example, we can define a program that puts two other programs in parallel, $\mathsf{parallel} : ((\mathsf{unit} \to \mathsf{empty}), (\mathsf{unit} \to \mathsf{empty})) \to \mathsf{empty}$, by spawning two threads and waiting for them:

$$\begin{aligned}\mathsf{parallel}(x, y) = \mathsf{case}\,(\underline{\mathsf{fork}}(), \underline{\mathsf{fork}}())\ \mathsf{of}\ \{&(\mathsf{Some}\,a, \mathsf{Some}\,b) \Rightarrow \underline{\mathsf{wait}}(a); \underline{\mathsf{wait}}(b); \underline{\mathsf{stop}}()\\ &(\mathsf{Some}\,a, \mathsf{None}) \Rightarrow y();\\ &(\mathsf{None}, \mathsf{Some}\,b) \Rightarrow x();\\ &(\mathsf{None}, \mathsf{None}) \Rightarrow \underline{\mathsf{stop}}()\}\end{aligned}$$



For this reason, we provide an equational theory for fork and wait in Section 4.1. A general idea is that $\mathsf{fork}(a.t, u)$ behaves like a monoid, with $\underline{\mathsf{wait}}(a); \underline{\mathsf{stop}}()$ like a unit, except care is needed for the thread ID parameter.

The **main result** of our paper is the representation theorem for the fork/wait theory (Theorem 5.4). This representation is along the lines of the labelled posets, except now there may be holes standing for the different continuations. For example, the 'parallel' operation becomes the 'cherries' diagram shown. This is non-trivial because any thread plugged in for x or y may have child threads that are not waited for, and may wait on other thread IDs that are not in the diagram.

We also prove a completeness theorem (Theorem 6.1), which says that if two expressions give the same labelled poset whatever we substitute into the variables x, y etc., then they are provably equal. We show this by finding special gadgets to substitute for the variables. This can be thought of as a full abstraction result, and we make this connection in Theorem 7.4.

For a final remark, we define an operation $\mathsf{series} : ((\mathsf{unit} \to \mathsf{empty}), (\mathsf{unit} \to \mathsf{empty})) \to \mathsf{empty}$ that forks a child thread and immediately waits for it:

$$\mathsf{series}(x, y) = \mathsf{case}\,\underline{\mathsf{fork}}()\ \mathsf{of}\ \{\mathsf{Some}\,a \Rightarrow \underline{\mathsf{wait}}(a); y() \mid \mathsf{None} \Rightarrow x()\}$$

We can use 'series' and 'parallel' to build series-parallel graphs [3, 24], and we can easily deduce from our algebraic theory that the equational laws of series-parallel graphs hold. But note that we can also express the N shape, which is not series-parallel.

Although 'series' is easy to program, it is not the same as the sequencing $x(); y()$ of the programming language. This can already be seen because the type is different. Another view is that the unit of $(;)$ is $(\mathsf{return}\ ())$ (return to the calling function) whereas the unit of series is $\underline{\mathsf{stop}}$ (end the current thread). The apparent similarity between 'series' and sequencing may be the reason for earlier claims that concurrency via algebraic effects has 'undesired equations' [46, §8,p33], such as parallel composition commuting with sequencing. By focusing instead on $\underline{\mathsf{fork}}$, $\underline{\mathsf{wait}}$, and dynamic threads, we make clear the distinction between sequencing and series: even though 'parallel' commutes with sequencing, it does not commute with 'series' as expected.

Paper summary. The operational and denotational semantics are given in Sections 2 and 7 respectively; the main programming language results are soundness, adequacy and full abstraction (§7.2). The method for building the denotational semantics is via algebraic effects, developed in Sections 3–4, shown to have a representation (§5) and thereby to be complete (§6).

2 BACKGROUND CONCURRENT PROGRAMMING LANGUAGE

To precisely frame the situation, we discuss a fairly standard concurrent programming language and operational semantics.

2.1 Language and Type System

We consider a standard higher-order order programming language with finite product and sum types (e.g. [22, 26]). The grammar of types is:

$$A, B \stackrel{\text{def}}{=} \mathsf{tid} \mid \prod_{i=1}^{k} A_i \mid \sum_{i=1}^{k} A_i \mid A \to B$$

When $k = 0$ we get the empty product and sum types, denoted by 1 and 0 respectively, instead of unit and empty in the introduction. The base type of thread IDs tid is specific to our setting.

The language is fine-grain call-by-value [22], meaning that terms are stratified into values and computations. The grammar of values is:

$$v ::= x \mid (v_1, \ldots, v_k) \mid \mathsf{inj}_i\, v \mid \lambda x.\, t \mid a \mid g$$

It contains variables, the usual constructors for product and sum types, and functions; the body of a function, t, is a computation. The symbol a ranges over a countably infinite set $\mathbb{T}$ of actual thread IDs. The symbol g ranges over a fixed set $\mathbb{F}$ of typed term constants $g : A \to B$. These constants allow us to add concurrency features to our languages.

The grammar of computations contains the usual destructors for product, sum and function types, and a let-construct for explicitly sequencing computations:

$$t ::= \mathsf{return}\, v \mid \mathsf{proj}_i\, v \mid \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inj}_i(x_i) \Rightarrow t_i\}_{i=1}^{k} \mid v_1\, v_2 \mid \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$$

We include the following set $\mathbb{F}$ of value constants $g : A \to B$ in our language, where σ ranges over a fixed set of observable actions Σ:

$$\underline{\mathsf{fork}} : 1 \to \mathsf{tid} + 1 \qquad \underline{\mathsf{stop}} : 1 \to 0 \qquad \underline{\mathsf{wait}} : \mathsf{tid} \to 1 \qquad \underline{\mathsf{perform}}_\sigma : 1 \to 1 \tag{1}$$

Intuitively, $\underline{\mathsf{fork}}()$ forks a new child thread and returns its ID to the parent thread, in the left branch of the sum type $\mathsf{tid} + 1$. The right branch is for the child thread which receives the unit value $()$. The continuation of $\underline{\mathsf{fork}}()$ will run twice, once in the parent thread and once in the child thread.

The computation $\underline{\mathsf{stop}}()$ signals that the current thread has finished and its continuation will be discarded. Once a thread has invoked $\underline{\mathsf{stop}}()$ it cannot resume running. The computation $\underline{\mathsf{wait}}(v)$ waits for all the threads with IDs in v to finish by invoking $\underline{\mathsf{stop}}()$, then returns unit. Finally, $\underline{\mathsf{perform}}_\sigma()$ performs the observable action σ then returns unit.

When writing programs, we use some syntactic sugar, such as $(t_1; t_2)$ for $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$ where t_2 does not depend on x, and $\mathsf{case}\, t\, \mathsf{of} \ldots$ instead of $\mathsf{let}\, x = t\, \mathsf{in}\, \mathsf{case}\, v\, \mathsf{of} \ldots$ where it is unambiguous.

Example 2.1. Consider the computations below:

$$\mathsf{let}\, y = \underline{\mathsf{fork}}()\, \mathsf{in}\, \mathsf{case}\, y\, \mathsf{of}\, \{\mathsf{inj}_1(x_1) \Rightarrow \underline{\mathsf{wait}}(x_1); \underline{\mathsf{perform}}_{\sigma_1}(); \underline{\mathsf{stop}}(),\ \mathsf{inj}_2() \Rightarrow \underline{\mathsf{perform}}_{\sigma_2}(); \underline{\mathsf{stop}}()\}$$

$$\mathsf{let}\, y = \underline{\mathsf{fork}}()\, \mathsf{in}\, \mathsf{case}\, y\, \mathsf{of}\, \{\mathsf{inj}_1(x_1) \Rightarrow \underline{\mathsf{perform}}_{\sigma_1}(); \underline{\mathsf{wait}}(x_1); \underline{\mathsf{stop}}(),\ \mathsf{inj}_2() \Rightarrow \underline{\mathsf{perform}}_{\sigma_2}(); \underline{\mathsf{stop}}()\}$$

In both, the main thread forks a new child thread that performs action σ_2 then stops; the ID of this new thread is bound to x_1. In the first, the main thread waits for the child to finish before performing action σ_1. So the sequence of observed actions is σ_2 followed by σ_1. In the second, the main thread does not wait, so we may also see the other order, or σ_1 and σ_2 concurrently.

For equational reasoning about programs, it is very helpful to combine thread IDs into *compound* thread IDs. For example, suppose that a thread a has nothing left to do but is waiting for b and c before it finishes. Then thread a is rather redundant, and the only reason to keep it is that the

$$\frac{}{\Gamma, x : A, \Gamma' \vdash^{\mathrm{v}}_w x : A} \qquad \frac{\left(\Gamma \vdash^{\mathrm{v}}_w v_i : A_i\right)^k_{i=1}}{\Gamma \vdash^{\mathrm{v}}_w (v_1, \dots, v_k) : \prod^k_{i=1} A_i} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v_i : A_i}{\Gamma \vdash^{\mathrm{v}}_w \mathsf{inj}_i\, v_i : \sum^k_{i=1} A_i} \qquad \frac{\Gamma, x : A \vdash^{\mathrm{c}}_w t : B}{\Gamma \vdash^{\mathrm{v}}_w \lambda x.\, t : A \to B}$$

$$\frac{\Gamma \vdash^{\mathrm{v}}_w v : A}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{return}\, v : A} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v : \prod^k_{i=1} A_i}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{proj}_i\, v : A_i} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v : \sum^k_{i=1} A_i \qquad \left(\Gamma, x_i : A_i \vdash^{\mathrm{c}}_w t_i : B\right)^k_{i=1}}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{case}\, v\, \mathsf{of}\, \{\mathsf{inj}_i(x_i) \Rightarrow t_i\}^k_{i=1} : B}$$

$$\frac{\Gamma \vdash^{\mathrm{v}}_w v_1 : A \to B \qquad \Gamma \vdash^{\mathrm{v}}_w v_2 : A}{\Gamma \vdash^{\mathrm{c}}_w v_1\, v_2 : B} \qquad \frac{\Gamma \vdash^{\mathrm{c}}_w t_1 : A \qquad \Gamma, x : A \vdash^{\mathrm{c}}_w t_2 : B}{\Gamma \vdash^{\mathrm{c}}_w \mathsf{let}\, x = t_1\, \mathsf{in}\, t_2 : B} \qquad \frac{(g : A \to B) \in \mathbb{F}}{\Gamma \vdash^{\mathrm{v}}_w g : A \to B}$$

Fig. 2. Standard typing rules for a fine-grain call-by-value programming language. Here $\mathbb{F}$ is given in (1).

parent thread might at some point wait for a. If the parent could instead wait for both b *and* c, then we can indeed finish a already. (This is an instance of axiom (17).) To reason in this example it is helpful to substitute the compound thread ID $(b \oplus c)$ for a. To facilitate this equational reasoning, which is the aim of this paper, we have the following constructions of compound threads ids.

$$\frac{a \in w}{\Gamma \vdash^{\mathrm{v}}_w a : \mathsf{tid}} \qquad \frac{}{\Gamma \vdash^{\mathrm{v}}_w \emptyset : \mathsf{tid}} \qquad \frac{\Gamma \vdash^{\mathrm{v}}_w v_1 : \mathsf{tid} \qquad \Gamma \vdash^{\mathrm{v}}_w v_2 : \mathsf{tid}}{\Gamma \vdash^{\mathrm{v}}_w v_1 \oplus v_2 : \mathsf{tid}}$$

2.2 Operational Semantics

We now define an operational semantics for the language, based on a labelled transition relation over configurations. These are pools of threads, some of which are ready to run, some are finished, and some are stuck waiting for others to finish before they can run. We include a relation stating which threads are waiting for which others to finish.

2.2.1 Alternative language construct: $\underline{\mathsf{act}}_\sigma$. In order to set up the operational semantics, it is convenient to consider the following operation

$$\underline{\mathsf{act}}_\sigma : 1 \to 0$$

which performs action σ and then finishes immediately. This is interdefinable with $\underline{\mathsf{perform}}_\sigma : 1 \to 1$:

$$\begin{aligned} \underline{\mathsf{act}}_\sigma() &= \underline{\mathsf{perform}}_\sigma(); \underline{\mathsf{stop}}() \\ \text{and} \quad \underline{\mathsf{perform}}_\sigma() &= \mathsf{let}\, x = \underline{\mathsf{fork}}()\, \mathsf{in}\, \mathsf{case}\, x\, \mathsf{of}\, \{\mathsf{inj}_1(a) \Rightarrow \underline{\mathsf{wait}}(a); \mathsf{return}\,(),\ \mathsf{inj}_2() \Rightarrow \underline{\mathsf{act}}_\sigma()\}. \end{aligned}$$

That is, an action that may be followed by other commands can be achieved by forking a new thread that merely performs the action, and then waiting for it.

Arguably, $\underline{\mathsf{perform}}_\sigma$ is more natural in programming, but $\underline{\mathsf{act}}_\sigma$ has an easier semantics. We focus on semantics, and so we focus on $\underline{\mathsf{act}}_\sigma$ as a primitive, regarding $\underline{\mathsf{perform}}_\sigma$ as derived.

2.2.2 Configurations.

Definition 2.2. A *configuration* is a triple $\langle w; \prec; \mathit{thread} \rangle$ where

- $w \subseteq \mathbb{T}$ is a finite set of thread IDs that are involved in this configuration.
- $(\prec) \subseteq \mathbb{T} \times w$ is a relation, relating a thread id with the id's it is waiting for.
- $\mathit{thread} : w \to \{\text{computations } t\} \uplus \mathit{finished}$ is a function assigning to each active thread id the computation that it runs, or '*finished*' if the thread has finished. We often enumerate the map e.g. writing $([a]t, [b]u)$, if $a \mapsto t$ and $b \mapsto u$.

We abbreviate the configuration when there is one thread, writing $\langle [a]t \rangle$ instead of $\langle \{a\}; \emptyset; [a]t \rangle$.

2.2.3 Transition relation. The transition system is given inductively in Figure 3. We define two transition relations between configurations: silent reductions $\longrightarrow$ and labelled reductions $\xrightarrow{\sigma}$, where σ is an action. We also annotate our transition relation with the thread which reduced (a), following [27]; this is not necessary and can be erased, but is useful in the metatheory.

The relation ($\prec$) specifies which threads are waiting for which other threads to finish. The last transition rule in Figure 3 says that a thread can step if indeed all the threads it was waiting for have finished. After a step, the waiting relation ($\prec$) needs to be updated with any new waits ($\prec'$).

The other transition rules are for the reduction of single threads. Wait reduces by recording what it is waiting for. Fork spawns a new child thread b, passing its identifier b to the parent thread a.

In this simple language, there is only one evaluation context, let $x = [-]$ in s. To evaluate here depends on which threads are reduced, spawned or finished by the expression in the context (t). We then continue to evaluate the let-expression with all of the existing and new threads; any finished threads will finish without evaluating the continuation s.

There are a couple of subtle points about the $\prec$ relation. First, a configuration $\langle w; \prec; \mathit{thread}\rangle$ may have $b \prec a$ for some b not in w. Thus a thread may wait on thread IDs not in the current pool. This is to allow us to restrict our view to particular threads, but will not happen at the top level.

Second, a configuration may have $a \prec b$ even if both a and b are finished. One could garbage-collect this redundant information, as any efficient implementation would do, but this is not necessary, and the metatheory is easier without it.

2.2.4 Observation as a labelled poset. We focus on true-concurrency semantics, and so, instead of considering only linear traces, we include the dependency order $\prec$. This gives a labelled poset (pomset [34, 37], equivalently conflict-free event structure [28]).

Definition 2.3. Let Σ be a set. A Σ-*labelled poset* is a partially ordered set $P = (X, \leq)$ equipped with a function $\ell : X \to \Sigma$. (We omit Σ where it is clear from the context.)

Definition 2.4. A *terminal configuration* $\langle w; \prec; \mathit{thread}\rangle$ is one where all threads have finished: $\mathit{thread}(a) = \mathit{finished}$ for all $a \in w$.

Let $(X, \leq, \ell)$ be a labelled poset, and C a configuration. We write $C \Downarrow (X, \leq, \ell)$ when the following conditions hold: $X = \{a_1, \ldots, a_n\}$; $a \leq b$ in X iff $a \prec b$ or $a = b$; and there is some terminal configuration $C' = \langle w; \prec; \mathit{thread}\rangle$ such that, letting $\sigma_i = \ell(a_i)$, C admits a sequence of transitions:

$$C \longrightarrow^* \xrightarrow{\sigma_1} a_1 \longrightarrow^* \xrightarrow{\sigma_2} a_2 \longrightarrow^* \xrightarrow{\sigma_n} a_n \longrightarrow^* C'$$

Recall from the reduction rules in Figure 3 that each action σ_i happens in a separate thread that finishes immediately, so the thread IDs $a_1, \ldots, a_n$ above are all distinct. In Example 2.1, the first program is related by $\Downarrow$ to the order $(\sigma_1 < \sigma_2)$, whereas the second one is related to the discrete order $\{\sigma_1, \sigma_2\}$. See [34] for further discussion of these concurrent notions of observation.

2.3 Operational Meta-theory

A well-typed program will never deadlock, and moreover that it induces a unique observed labelled poset. More elaborate languages would not have these properties, but in this simple setting they are useful for connecting exactly with the denotational semantics in Section 7.

2.3.1 Well-formed configurations. Well-typed programs never deadlock, that is, there are never two threads that are waiting for each other. To show this, we consider well-formed configurations for which there exists a linear order, which encodes a *potential creation order* of threads. The idea is that the configuration appears as if the greatest thread is a parent thread that has itself forked *all* the other threads in the configuration; the smallest thread is the child that was forked first, then the second child etc.. A child can only refer to siblings that were forked earlier, so are smaller in

Reduction rules for the main language constructs.

$$\langle [a]\underline{\mathsf{wait}}(v)\rangle \longrightarrow_a \langle \{a\}; \{b \prec a \mid b \in \mathit{tids}(v)\}; [a]\mathsf{return}\ ()\rangle$$
$$\langle [a]\underline{\mathsf{fork}}()\rangle \longrightarrow_a \langle \{a,b\}; \emptyset; [a]\mathsf{return}\ (\mathsf{inj}_1\ (b)), [b]\mathsf{return}\ (\mathsf{inj}_2\ ())\rangle \quad (a \neq b)$$
$$\langle [a]\underline{\mathsf{stop}}()\rangle \longrightarrow_a \langle [a]\mathit{finished}\rangle$$
$$\langle [a]\underline{\mathsf{act}_\sigma}()\rangle \xrightarrow{\sigma}_a \langle [a]\mathit{finished}\rangle$$

Standard reduction rules for products, sums, functions, and let binding.

$$\langle [a]\mathsf{proj}_i\ (v_1, \dots, v_k)\rangle \longrightarrow_a \langle [a]\mathsf{return}\ v_i\rangle$$
$$\langle [a]\mathsf{case}\ \mathsf{inj}_i\ (v)\ \mathsf{of}\ \{\mathsf{inj}_k(x_k) \Rightarrow t_k\}_{k=1}^{l}\rangle \longrightarrow_a \langle [a]t_i[v/x_i]\rangle$$
$$\langle [a](\lambda x.t)\ v\rangle \longrightarrow_a \langle [a]t[v/x]\rangle$$
$$\langle [a]\mathsf{let}\ x = \mathsf{return}\ v\ \mathsf{in}\ t\rangle \longrightarrow_a \langle [a]t[v/x]\rangle$$

Reducing in evaluation context:

$$\frac{\langle [a]t\rangle \xrightarrow{(\sigma)}_a \langle w; \prec; \mathit{thread}\rangle}{\langle [a]\mathsf{let}\ x = t\ \mathsf{in}\ s\rangle \xrightarrow{(\sigma)}_a \left\langle w; \prec; \begin{array}{l}\{[b]\mathsf{let}\ x = t'\ \mathsf{in}\ s \mid b \in w, \mathit{thread}(b) = t'\} \cup \\ \{[b]\mathit{finished} \mid b \in w, \mathit{thread}(b) = \mathit{finished}\}\end{array}\right\rangle}$$

Converting thread-local transitions to global transitions on the configuration:

$$\frac{\langle \{a\}; \emptyset; [a]t\rangle \xrightarrow{(\sigma)}_a \langle w; \prec'; \mathit{thread}\rangle \qquad \forall b.\ b \prec a \implies \mathit{thread}(b) = \mathit{finished}}{\begin{array}{l}\langle w_0 \uplus \{a\}; \prec; \mathit{thread}_0 \uplus \{[a]t\}\rangle \\ \xrightarrow{(\sigma)}_a \langle w_0 \uplus w; (\prec \cup \prec' \cup \{(b,c) \mid b \prec a, c \in w\})^*; \mathit{thread}_0 \uplus \mathit{thread}\rangle\end{array}}$$

where $(-)^*$ denotes transitive closure.

Fig. 3. Operational semantics for our concurrent programming language (Sec. 2.2). We write $\xrightarrow{(\sigma)}$ with parentheses to indicate two copies of the rule, one with the label and one without. Here $\mathit{tids}(a) = a$, $\mathit{tids}(v \oplus v') = \mathit{tids}(v) \cup \mathit{tids}(v')$, and $\mathit{tids}(\emptyset) = \emptyset$.

the order; the parent can refer to any of the children. Note that the threads might not have been created in this (or any other) linear order, and there may have been more complex parent-child relationships. The operational semantics does not depend on the creation order and there may be multiple linear orders that are all consistent with a given configuration.

Definition 2.5. Consider a configuration $C = \langle w; \prec; \mathit{thread}\rangle$. Consider a linear order $<$ on w, and a type A. Let $w' \subseteq \mathbb{T}$ be disjoint from w. (The idea is that $<$ is the creation order, and w' describes some threads not in the current pool, which may be useful when we are zooming in on single threads.) We say C is *well-formed* for $(A, <, w')$ if

- The waiting order $\prec$ is transitive.
- A thread only waits on threads in the pool or in w': if $a \prec b$ then $a \in w \uplus w'$.
- Threads only wait for siblings that were created earlier: if $a \prec b$ and $a, b \in w$ then $a < b$.
- All threads have type A, and only rely on previously created siblings: for all $a \in w$ we have $\vdash^{\mathsf{c}}_{\{b<a\}\uplus w'} \mathit{thread}(a) : A$; (the parent i.e. greatest in $<$, can rely and wait on all its children).

Proposition 2.6 (Preservation). *Let $C_1 = \langle w_1; <_1; thread_1\rangle$ and $C_2 = \langle w_2; <_2; thread_2\rangle$. If C_1 is well-formed for $(A, <_1, w')$ and $C_1 \xrightarrow{(\sigma)} C_2$ and w' is disjoint from w_2, then there is a linear order $<_2$ extending $<_1$ such that C_2 is well-formed for $(A, <_2, w')$.*

Proof notes. By induction on the derivation of transitions. □

2.3.2 *Labelled posets uniquely determined from terms of empty type.*

Proposition 2.7. *Consider a term $\vdash^{\mathsf{c}}_{\emptyset} t : 0$ of empty type in the empty world $\emptyset$.*

(1) *The configuration $\langle [a]t\rangle$ reaches a terminal configuration, i.e. there exists a labelled poset (P, ℓ) such that $\langle [a]t\rangle \Downarrow (P, \ell)$.*
(2) *If $\langle [a]t\rangle \Downarrow (P_1, \ell_1)$ and $\langle [a]t\rangle \Downarrow (P_2, \ell_2)$, then the labelled posets are isomorphic, i.e. there is an order isomorphism $f : P_1 \cong P_2$ such that $\ell_1(e) = \ell_2(f(e))$.*

Proof notes. Part 1 holds in greater generality: every reduction sequence starting in a well-typed term $\vdash^{\mathsf{c}}_{\emptyset} t : A$ is finite. Our proof uses a straightforward combination of Tait's method [44] and Kőnig's tree lemma [21]. Each reduction sequence of a program induces a finitely-branching tree in which each branch corresponds to the sequential execution of a single thread that does not mention the other threads. These thread-local executions include transitions steps in which the environment changes the status of a known tid to *finished*. Each infinite reduction sequence induces an infinite such tree, and Kőnig's tree lemma implies it has an infinite branch. We then use Tait's method, i.e., design an appropriate Kripke logical relation, that shows that in all well-typed programs every thread has only finite sequential executions. The Kripke property of the relation is with respect to injective relabelling of tids. We define two 'value' relations: one indexed by types over closed values, and the other indexed by contexts over closed environments. The computation relation, indexed by types, over closed computations states that the computation has no infinite reduction sequence, and whenever the computation evaluates to return v, the value v satisfies the appropriate value relation. We then prove the Fundamental Property of these relations: every well-typed value, computation, and substitution maps closed environments satisfying the value relation to values, computations, and closed environments satisfying the relation.

For part 2, we prove a confluence result. First, we pick a deterministic naming scheme for the fresh thread IDs introduced by fork(), so that fresh thread IDs are independent of the evaluation order. One good scheme (e.g. [27]) is that a thread ID is a finite sequence of numbers, with the idea that the ID $(m_1 m_2 m_3 \ldots m_k m_{k+1})$ is the m_{k+1}th thread spawned directly by the thread $(m_1 \ldots m_k)$.

We then show that

(1) If $C \xrightarrow{(\sigma)}_a C_1$ and $C \xrightarrow{(\sigma)}_a C_2$ then $C_1 = C_2$; and
(2) If $C \xrightarrow{(\sigma)}_a C_1$ and $C \xrightarrow{(\tau)}_b C_2$ then there is C' such that $C_1 \xrightarrow{(\tau)}_b C'$ and $C_2 \xrightarrow{(\sigma)}_a C'$.

The first is local determinacy within each thread, which is straightforwardly proved by induction on the structure of transition derivations. The second is also proved by induction on the structure of transition derivations. However, some care is needed that the transitive closure in the local-to-global rule for a step in a particular enabled thread does not introduce dependencies that would cause a different currently enabled thread to have to wait. Here the key strengthening of the induction hypothesis is to show that

$$\text{If } \langle w; <; thread\rangle \xrightarrow{(\sigma)}_a \langle w'; <'; thread'\rangle \text{and } c <' b \text{ and } b \in w \text{ then either } c < b \text{ or } a = b \text{ or } a < b.$$

□

3 PARAMETRIZED ALGEBRAIC THEORIES, ILLUSTRATED VIA A NEW THEORY OF LABELLED POSETS

Algebraic effects are formalized using algebraic theories from universal algebra. This section recalls the concepts of algebraic theories and their generalisation, parametrized algebraic theories, along with a novel running example, the theory of labelled posets.

3.1 Algebraic Theories

Definition 3.1. A (first-order finitary) *algebraic signature* $O = \langle |O|, \mathrm{ar} \rangle$ is a collection of operations $|O|$ and a function $\mathrm{ar} : |O| \to \mathbb{N}$, associating a natural number to each operation, called its *arity*.

We write $\mathrm{O} : n$ for an operation O with arity n. A *context* $\Delta = a_1, \ldots, a_n$ is a list of distinct variables. *Terms* in a context Δ are inductively generated by:

$$\frac{}{\Delta, a, \Delta' \vdash a} \qquad \frac{\Delta \vdash u_1 \quad \cdots \quad \Delta \vdash u_n \qquad \mathrm{O} : n}{\Delta \vdash \mathrm{O}(u_1, \ldots, u_n)}$$

Definition 3.2. A (first-order finitary) *algebraic theory* $\mathcal{T} = (O, E)$ is a pair of an algebraic signature O and a set of *equations* E, where an equation is a pair of terms in the same context, $\Delta \vdash t_1$ and $\Delta \vdash t_2$, which we write $\Delta \vdash t_1 = t_2$.

Example 3.3. The algebraic theory of semi-lattices $\mathcal{L}$ has two operations: $\oplus : 2$ and $\mathtt{0} : 0$. The equations are that $\oplus$ is associative, symmetric, idempotent, and has $\mathtt{0}$ as its unit:

$$a, b, c \vdash (a \oplus b) \oplus c = a \oplus (b \oplus c) \qquad a, b \vdash a \oplus b = b \oplus a \qquad a \vdash a \oplus a = a \qquad a \vdash a \oplus \mathtt{0} = a$$

The theory of semi-lattices is often used as a semantics for non-deterministic choice with failure [26].

3.2 Parametrized Algebraic Theories

Parametrized algebraic theories are an extension of plain algebraic theories that allow the binding of abstract parameters. They have been used to axiomatize effects that involve a kind of resource, such as new memory locations in local state and channels in the π-calculus [42]. We introduce the concept in this section, highlighted by an example of a theory of labelled posets.

A parametrized algebraic theory is parametrized over an ordinary algebraic theory in the sense of Definition 3.2. For the rest of the paper, we fix this ordinary algebraic theory to be the theory of semi-lattices from Example 3.3. We recall the definition of parametrized algebraic theories along with a running example of a novel theory of labelled posets.

Definition 3.4. A *parametrized signature* $O = \langle |O|, \mathrm{ar} \rangle$ is a collection of operations $|O|$ along with a function $\mathrm{ar} : |O| \to \mathbb{N} \times \mathbb{N}^*$, associating to each operation O an arity consisting of a natural number and a list of natural numbers: $\mathrm{ar}(\mathrm{O}) = (p \mid m_1, \ldots, m_k)$. This means O takes p parameters and k continuations, binding m_i parameters in the ith continuation.

Example 3.5. Consider operations $\mathrm{node}_\sigma : (1 \mid 1)$ and $\mathrm{end} : (0 \mid)$ where σ ranges over a finite set of observable actions Σ. The node_σ operation takes one free parameter and one continuation binding one parameter; end takes zero parameters and no continuations. A parameter stands for a term in the theory of semi-lattices.

A parametrized context $\Gamma \mid \Delta$ is a list Δ of *parameter variables* (i.e. an ordinary algebraic context) and a list of distinct *computation variables* $\Gamma = x_1 : m_1, \ldots, x_k : m_k$, where each x_i is annotated with the number m_i of parameters it uses. Terms in context $\Gamma \mid \Delta$ are inductively generated by:

$$\frac{(\Delta \vdash u_i)_{i=1}^{m}}{\Gamma, x : m, \Gamma' \mid \Delta \vdash x(u_1, \ldots, u_m)} \qquad \frac{(\Delta \vdash u_i)_{i=1}^{p} \qquad \left(\Gamma \mid \Delta, b_1, \ldots, b_{m_i} \vdash t_i\right)_{i=1}^{k} \qquad \mathrm{O} : (p \mid m_1, \ldots, m_k)}{\Gamma \mid \Delta \vdash \mathrm{O}(u_1, \ldots, u_p,\ b_1 \ldots b_{m_1}.t_1,\ \ldots,\ b_1 \ldots b_{m_k}.t_k)}$$

where $\Delta \vdash u_i$ is a term judgement in the ordinary algebraic theory of semi-lattices from Example 3.3. Both contexts admit all the usual structural rules and we treat all terms up to renaming of variables.

Using the signature from Example 3.5 we can build the following terms in context:

$$x : 1 \mid a \vdash \mathsf{node}_\sigma(a, b.x(b)) \tag{2}$$

$$x : 2 \mid a_1, a_2 \vdash \mathsf{node}_\sigma(a_1 \oplus a_2, b_1.\mathsf{node}_\tau(a_1, b_2.x(b_2, b_1))) \tag{3}$$

In the term (2), a is a free parameter while b is bound in x. From a concurrency perspective, we interpret $\mathsf{node}_\sigma(a, b.x(b))$ as forking a new child thread that performs action σ after the thread with ID a has performed its action. The thread ID of the child performing σ is b and the continuation $x(b)$ is executed concurrently.

The term (3) uses the operation $\oplus$ from the theory of semi-lattices to wait on both thread IDs a_1 and a_2 before executing σ. Figure 4 (c) is a graphical representation of the term (3), where a_1 and a_2 are inputs at bottom, and the two parameters that $x : 2$ takes are outputs at the top. The names of bound parameters b_1 and b_2 do not appear. The solid line signifies causal dependency.

We define two substitution operations on terms, one for parameters variables and one for computation variables, in the standard capture-avoiding way as to admit the following rules:

$$\frac{\Gamma \mid \Delta, a \vdash t \qquad \Delta \vdash u}{\Gamma \mid \Delta \vdash t[u/a]} \qquad \frac{\Gamma, x : m \mid \Delta \vdash t \qquad \Gamma \mid \Delta, b_1, \dots, b_m \vdash s}{\Gamma \mid \Delta \vdash t[b_1 \dots b_m.s/x]} \tag{4}$$

The notation $b_1 \dots b_m.s/x$ emphasises that the bound parameters $b_1, \dots, b_m$ in s will be replaced with the parameters passed to x.

Below are examples of each kind of substitution. They can be understood graphically: the first transforms Figure 4 (b) into Figure 4 (c) and the second transforms Figure 4 (c) into Figure 4 (d).

$$\mathsf{node}_\sigma(a_3, b_1.\mathsf{node}_\tau(a_1, b_2.x(b_2, b_1)))[a_1 \oplus a_2/a_3] = \mathsf{node}_\sigma(a_1 \oplus a_2, b_1.\mathsf{node}_\tau(a_1, b_2.x(b_2, b_1))) \tag{5}$$

$$\mathsf{node}_\sigma(a_1 \oplus a_2, c_1.\mathsf{node}_\tau(a_1, c_2.x(c_2, c_1)))[b_1 b_2.y(b_1 \oplus b_2)/x] = \mathsf{node}_\sigma(a_1 \oplus a_2, c_1.\mathsf{node}_\tau(a_1, c_2.y(c_2 \oplus c_1))) \tag{6}$$

Definition 3.6. A *parametrized algebraic theory* $\mathcal{T} = (O, E)$ is a pair of a parametrized signature O and a set E of equations. An equation is a pair of terms in the same context $\Gamma \mid \Delta$, which we write as $\Gamma \mid \Delta \vdash t_1 = t_2$.

Example 3.7. The parameterized *theory of labelled posets* consists of the signature from Example 3.5, containing node_σ and end, and of the following two equations:

$$x : 2 \mid a_1, a_2 \vdash \mathsf{node}_\sigma(a_1, b_1.\mathsf{node}_\tau(a_2, b_2.x(b_1, b_2))) = \mathsf{node}_\tau(a_2, b_2.\mathsf{node}_\sigma(a_1, b_1.x(b_1, b_2))) \tag{7}$$

$$x : 1 \mid a \vdash \mathsf{node}_\sigma(a, b.x(b)) = \mathsf{node}_\sigma(a, b.x(a \oplus b)) \tag{8}$$

The first equation states that independent actions may happen in any order. The second equation encodes the transitivity of causal dependencies. There are no equations involving end; intuitively end finishes the execution of the whole program.

In Section 4 we will present an extended example of a parameterized algebraic theory for forking threads, together with examples of equational reasoning in Examples 4.2 to 4.4.

Given a parameterized theory $\mathcal{T} = (O, E)$, we can form an equivalence relation $=_\mathcal{T}$ on the terms of $\mathcal{T}$, called *derivable equality*, by closing all *simultaneous substitution instances* of the equational axioms from E under reflexivity, symmetry, transitivity, and two congruence rules, one for variables and one for the operations in O. Below is the congruence rule for variables; it allows us to use the equations of the theory of semi-lattices when reasoning about parameterized terms.

$$\frac{\left(\Delta \vdash u_i = u_i'\right)_{i=1}^m}{\Gamma, x : m \mid \Delta \vdash x(u_1, \dots, u_m) =_\mathcal{T} x(u_1', \dots, u_m')}$$

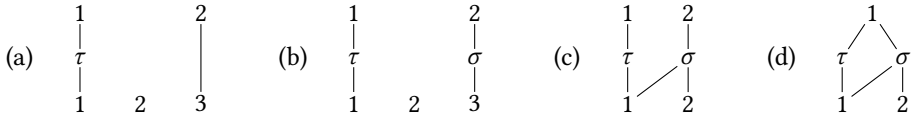


Fig. 4. Graphical examples of terms built from the node operation. (a) is the term $\mathrm{node}_\tau(a_1, b.x(b, a_3))$; numbers 1 and 2 at the top correspond to the two inputs of variable $x : 2$. (b) is the application of $\mathrm{node}_\sigma(a_3', a_3.-)$ to (a). (c) is the term in eq. (3); it is obtained from (b) by substituting $a_1 \oplus a_2$ for a_3' as in eq. (5); (d) is obtained from (c) by substituting a term for the computation variable $x : 2$, as in eq. (6).

3.3 Models of Parameterized Algebraic Theories

We recall models of parameterized algebraic theories by analogy with models for ordinary algebraic theory. For this paper, it is sufficient to consider the case where the parameterizing theory is that of semi-lattices (Example 3.3), but more general notions of models exist [41–43]. In Section 3.3.3 we illustrate models by considering the theory of labelled posets from Example 3.7.

3.3.1 Connection between the category of finite sets and relations and the theory of semi-lattices. We define the objects of the category FinRel to be natural numbers n and morphisms $n \to n'$ to be relations $R \subseteq [n] \times [n']$, where $[n]$ denotes the set $\{1, \dots, n\}$. Composition $S \circ R$ of $R \subseteq [n] \times [n']$ with $S \subseteq [n'] \times [n'']$ is the usual composition of relations.

For each p, we can define an isomorphism between the set of (equivalence classes of) terms $\{[a_1, \dots, a_p \vdash u]\}$ in the theory of semi-lattices and the set of morphisms $\mathsf{FinRel}(1, p)$:

$$[\![a_1, \dots, a_p \vdash a_i]\!] = \{(1, i)\} \qquad [\![a_1, \dots, a_p \vdash 0]\!] = \emptyset$$

$$[\![a_1, \dots, a_p \vdash u_1 \oplus u_2]\!] = [\![a_1, \dots, a_p \vdash u_1]\!] \cup [\![a_1, \dots, a_p \vdash u_2]\!]$$

3.3.2 Models of parameterized theories in $\mathsf{Set}^{\mathsf{FinRel}}$. A model of an ordinary algebraic theory consists of a set, the carrier, together with structure for interpreting the operations in the theory, such that all equational axioms are satisfied. We are studying theories parameterized by the algebraic theory of semi-lattices, so we will consider models where, instead of a set, the carrier is a family of sets indexed by the objects of the category FinRel.

Definition 3.8. Let $\mathcal{O}$ be a parameterized signature. A *$\mathcal{O}$-structure* $\mathcal{X}$ is an object X in the functor category $\mathsf{Set}^{\mathsf{FinRel}}$ equipped with for each operation $\mathsf{O} : (p \mid m_1, \dots, m_k)$ a family of functions indexed by natural numbers n, and respecting naturality with respect to morphisms in FinRel:

$$\mathsf{O}_{\mathcal{X},n} : X(n + m_1) \times \ldots \times X(n + m_k) \to X(n + p)$$

Given a $\mathcal{O}$-structure $\mathcal{X}$, the interpretation of operations can be extended to all terms using the interpretation of semi-lattices terms from Section 3.3.1. A term $x_1 : m_1, \dots, x_k : m_k \mid a_1, \dots, a_p \vdash t$ is interpreted as a family of functions

$$[\![t]\!]_{\mathcal{X},n} : X(n + m_1) \times \ldots \times X(n + m_k) \to X(n + p)$$

natural in n, defined by structural recursion. A computation variable

$$x_1 : m_1, \dots, x_k : m_k \mid a_1, \dots, a_p \vdash x_i(u_1, \dots, u_{m_i})$$

is interpreted as projection followed by the interpretation of its semi-lattice term inputs

$$X(n + m_1) \times \ldots \times X(n + m_k) \xrightarrow{\pi_i} X(n + m_i) \xrightarrow{X(n + [[\![u_1]\!], \dots, [\![u_{m_i}]\!]])} X(n + p)$$

where $[\![u_1]\!], \dots, [\![u_{m_i}]\!] : 1 \to p$ are morphisms in FinRel.

A term of the form $\Gamma \mid a_1, \dots, a_p \vdash \mathrm{O}(u_1, \dots, u_{p'},\ b_1 \dots b_{m'_1}.t_1,\ \dots,\ b_1 \dots b_{m'_{k'}}.t_{k'})$ is interpreted as the n-indexed family,

$$X(n + [p, [\![u_1]\!], \dots, [\![u_{p'}]\!]]) \circ \mathrm{O}_{\mathcal{X},n+p} \circ \langle [\![t_1]\!]_{\mathcal{X},n}, \dots, [\![t_{k'}]\!]_{\mathcal{X},n} \rangle,$$

where $\mathrm{O} : (p' \mid m'_1, \dots, m'_{k'})$. The map $[p, [\![u_1]\!], \dots, [\![u_{p'}]\!]] : p + p' \to p$ interprets the p' arguments of O using the parameter variables $a_1, \dots, a_p$.

Definition 3.9. Let $\mathcal{T} = (O, E)$ be a parameterized theory. A O-structure $\mathcal{X}$ is a *model* for the theory $\mathcal{T}$ if for every equational axiom from E, $\Gamma \mid \Delta \vdash s = t$, and for every natural number n, the following functions are equal:

$$[\![\Gamma \mid \Delta \vdash s]\!]_{\mathcal{X},n} = [\![\Gamma \mid \Delta \vdash t]\!]_{\mathcal{X},n}.$$

Proposition 3.10. *For a parameterized theory $\mathcal{T}$, the derivable equality $=_{\mathcal{T}}$ is sound: if $s =_{\mathcal{T}} t$ is derivable, then $[\![s]\!]_{\mathcal{X}} = [\![t]\!]_{\mathcal{X}}$ in any $\mathcal{T}$-model $\mathcal{X}$.*

3.3.3 A model of the theory of labelled posets. To illustrate the notion of model from the previous section, we build a model for the parameterized algebraic theory of labelled posets from Example 3.7. To do this we generalize the notion of Σ-labelled poset from Definition 2.3.

Definition 3.11. An *n-input m-output Σ-labelled poset* $P = \langle V_P, \leq_P, l_P \rangle$ consists of a set V_P of elements labelled by a function $l_P : V_P \to \Sigma$, and a partial order $\leq_P$ on the set $[n] \uplus V_P \uplus [m]$, such that the n input elements are minimal and the m output elements are maximal.

Examples of such posets appear in Figure 4, with inputs at the bottom and outputs at the top. If there are no inputs and outputs, Definition 3.11 reduces to that of an ordinary Σ-labelled poset. An isomorphism between two n-input m-output Σ-labelled posets P and Q is a bijection $f : V_P \to V_Q$ that preserves the labels, and such that $\mathrm{id}_{[n]} \uplus f \uplus \mathrm{id}_{[m]}$ preserves and reflects the order.

For each natural number n, define the set $S_m(n)$ to contain *isomorphism classes of n-input m-output Σ-labelled posets*. Given a relation $R \subseteq [n] \times [n']$, $S_m(R)$ acts on a labelled poset by updating $i \leq e$ to $i' \leq e$ if $(i, i') \in R$. The poset in Figure 4 (c) is obtained from Figure 4 (b) via this action.

For natural numbers m and n, we equip S_m with an operation

$$\mathrm{node}_{\sigma,m,n} : S_m(n+1) \to S_m(n+1)$$

which given a labelled poset, labels its $(n+1)$-th input by σ and creates a new $(n+1)$ input just below σ. The poset in Figure 4 (b) is the result of applying $\mathrm{node}_{\sigma,2,3}$ to Figure 4 (a).

Remark. Labelled posets can be organised into a PROP (see [23]), where morphisms $n \to m$ are labelled posets $S_m(n)$, identities are given by the poset with no labelled elements, composition "plugs" the outputs of a poset into the inputs of another, and monoidal composition is juxtaposition. This categorical formulation was valuable for proving Proposition 3.12 and Theorem 3.15. Similar categorical ideas appear elsewhere, e.g. [6, 12, 18], but with a typically a first-order emphasis, whereas we are aiming at a semantics for programming language via monads (§7).

Let Γ be a context of computation variables. For each natural number n, define the set

$$\mathrm{S}_\Gamma(n) = \biguplus_{x:m \in \Gamma} S_m(n) \uplus S_0(n).$$

We equip $\mathrm{S}_\Gamma(n)$ with an operation $\mathrm{node}_{\sigma,n} : \mathrm{S}_\Gamma(n+1) \to \mathrm{S}_\Gamma(n+1)$ by applying $\mathrm{node}_{\sigma,m,n}$ pointwise. Let $\mathrm{end}_n : 1 \to \mathrm{S}_\Gamma(n)$ be the function that selects, from the right injection $S_0(n)$, the labelled poset with only the n inputs as elements and with discrete order.

Proposition 3.12. *For each context Γ, the functor S_Γ, together with the natural transformations* node_σ *and* end *defined above, is a model of the parametrized theory of labelled posets from Example 3.7.*

3.4 Free Models of Parametrized Algebraic Theories

We now return to the study of models of parameterized algebraic theories in general. Using the evident notion of homomorphism between $\mathcal{O}$-structures, we can discuss $\mathcal{O}$-structures and $\mathcal{T}$-models that are *free* over a collection $X \in \mathsf{Set}^{\mathsf{FinRel}}$ of generators.

Definition 3.13. Consider a $\mathcal{T}$-model $\mathcal{Y}$ with carrier $Y \in \mathsf{Set}^{\mathsf{FinRel}}$ and a morphism $\eta_X : X \to Y$ in $\mathsf{Set}^{\mathsf{FinRel}}$. The model $\mathcal{Y}$ is *free on* X if for any other model $\mathcal{Z}$ and any morphism $f : X \to Z$ in $\mathsf{Set}^{\mathsf{FinRel}}$, there exists a unique homomorphism of models $\hat{f} : \mathcal{Y} \to \mathcal{Z}$ that extends f, meaning $\hat{f} \circ \eta_X = f$ in $\mathsf{Set}^{\mathsf{FinRel}}$.

Given a context of computation variables Γ, consider the functor V_Γ where:

$$V_\Gamma(n) = \{[\Gamma \mid a_1, \dots, a_n \vdash x(u_1, \dots, u_m)]_{=_\mathcal{T}}\}.$$

The equivalence relation on terms in V_Γ is non-trivial because the parameter terms u_i are quotiented by the semi-lattice equations.

The *term model* is given by the functor $F_\mathcal{T}(V_\Gamma)$, which contains equivalence classes of terms:

$$F_\mathcal{T}(V_\Gamma)(n) = \{[\Gamma \mid a_1, \dots, a_n \vdash t]_{=_\mathcal{T}}\} \tag{9}$$

The action on morphisms $n \to n'$, which are relations $R \subseteq [n] \times [n']$, is given by substitution of parameters. The functor $F_\mathcal{T}(V_\Gamma)$ can be made into a $\mathcal{T}$-model using the syntactic term formation rules, and we can construct a morphism $\eta_{V_\Gamma} : V_\Gamma \to F_\mathcal{T}(V_\Gamma)$ by embedding variables into terms. We use the term model to prove the completeness result below.

Proposition 3.14.

(1) *The functor $F_\mathcal{T}(V_\Gamma)$ is a $\mathcal{T}$-model and is moreover a free $\mathcal{T}$-model on V_Γ.*
(2) *The derivable equality $=_\mathcal{T}$ in a parameterized algebraic theory is complete: if an equation is valid in every $\mathcal{T}$-model then it is derivable in $=_\mathcal{T}$.*

We can now characterize the labelled posets model from Proposition 3.12 using the universal property of a free model. In particular, equivalence classes of closed terms $\{[- \mid - \vdash t]_{=_C}\}$ built from node and end (Example 3.7) are in bijection with ordinary Σ-labelled posets.

Theorem 3.15. *For each context Γ, the functor $\mathbb{S}_\Gamma$ together with the natural transformations node_σ and end is isomorphic to the free model $F_C(V_\Gamma)$ of the theory of labelled posets from Example 3.7.*

4 A PARAMETERIZED ALGEBRAIC THEORY OF DYNAMIC THREADS

In this section we introduce an equational axiomatization for the concurrency features from Section 2 (fork, wait, stop and act_σ) as a parameterized algebraic theory. In Section 5 we interpret this theory semantically, using labelled posets similar to those from Section 3.3.3. Then in Section 7 we extend the semantics to model the whole concurrent programming language from Section 2.

4.1 Presentation of the Theory

4.1.1 Signature. We introduce a *theory of dynamic threads* $\mathcal{T}$, parameterized by the theory of semi-lattices, with the following signature, where σ ranges over a finite set of observable actions Σ:

$$\mathsf{fork} : (0 \mid 1, 0) \qquad \mathsf{wait} : (1 \mid 0) \qquad \mathsf{stop} : (0 \mid) \qquad \mathrm{act}_\sigma : (0 \mid)$$

In the term $\mathsf{fork}(a.x(a), y)$ the variable x is the parent thread, while y is the child thread; the parameter a is the thread ID of the child y and is bound in x. The parent might wait for the child named a to finish, then continue as z, using the operation $\mathsf{wait}(a, z)$. The operation stop has no continuation and indicates that the current thread has finished execution; act_σ performs the

Equations describing the interaction of wait with the semi-lattice structure of thread IDs.

$$x : 0 \mid - \vdash \mathsf{wait}(\emptyset, x) = x \tag{10}$$

$$x : 0 \mid a, b \vdash \mathsf{wait}(a, \mathsf{wait}(b, x)) = \mathsf{wait}(a \oplus b, x) \tag{11}$$

$$x : 1 \mid a, b \vdash \mathsf{wait}(a, x(b)) = \mathsf{wait}(a, x(a \oplus b)) \tag{12}$$

The wait and fork operations commute; fork is commutative and associative.

$$x : 1, y : 0 \mid b \vdash \mathsf{wait}(b, \mathsf{fork}(a.x(a), y)) = \mathsf{fork}(a.\mathsf{wait}(b, x(a)), \mathsf{wait}(b, y)) \tag{13}$$

$$x : 2, y : 0, z : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{fork}(b.x(a, b), y), z) = \mathsf{fork}(b.\mathsf{fork}(a.x(a, b), z), y) \tag{14}$$

$$x : 1, y : 1, z : 0 \mid - \vdash \mathsf{fork}(a.x(a), \mathsf{fork}(b.y(b), z)) = \mathsf{fork}(b.\mathsf{fork}(a.x(a), y(b)), z) \tag{15}$$

The stop operation acts as a unit for fork.

$$x : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{stop}), x) = x \tag{16}$$

$$x : 1 \mid b \vdash \mathsf{fork}(a.x(a), \mathsf{wait}(b, \mathsf{stop})) = x(b) \tag{17}$$

Fig. 5. Equations for the parameterized algebraic theory of dynamic threads $\mathcal{T}$.

observable action σ and finishes. Parameters carry a semi-lattice structure, so it is possible to wait on a compound thread ID, e.g. $\mathsf{wait}(a_1 \oplus a_2, z)$ waits for both a_1 and a_2, or on no thread ID at all, $\emptyset$.

Example 4.1. The term t_1 encodes sequential execution of action σ_1 followed by σ_2, while t_2 and t_3 encode concurrent execution of σ_1 and σ_2:

$$t_1 = \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1}) \qquad t_2 = \mathsf{fork}(a.\mathsf{act}_{\sigma_2}, \mathsf{act}_{\sigma_1}) \qquad t_3 = \mathsf{fork}(a.\mathsf{act}_{\sigma_1}, \mathsf{act}_{\sigma_2})$$

However, t_2 and t_3 have slightly different intended semantics. In the term $\mathsf{fork}(b.\mathsf{wait}(b, \mathsf{act}_\tau), t_2)$ the ID b refers only to the thread act_{σ_2} and not to its child act_{σ_1}, so a possible execution is $\sigma_2\tau\sigma_1$. But this is not possible in $\mathsf{fork}(b.\mathsf{wait}(b, \mathsf{act}_\tau), t_3)$ because here σ_1 must happen before τ.

More generally, in the expression $\mathsf{fork}(a.x(a), y)$ we often refer to x as the *main thread* because its ID is available to the environment to wait on, while the ID of y is only available to x.

4.1.2 Equations. The equational axioms for the theory of dynamic threads appear in Figure 5. There are no equations involving observable actions act_σ. Equation (12) states that if x is waiting for a to finish, then waiting for a in the future is redundant. Commutativity of fork, eq. (14), holds only if the children y and z do not use each other's IDs. Similarly, associativity, eq. (15), holds if the parent x does not use the ID b of z. Equation (16) says that forking a child x and waiting for it to finish is the same as running x as the main thread. Equation (17) removes a child that does not perform any observable action; it involves a substitution of parameter b for a in x.

Example 4.2. Similarly to the discussion in Section 2.2.1, we can use act_σ to encode an operation $\mathsf{perform}_\sigma(x)$ which executes action σ then continues as x:

$$\mathsf{perform}_\sigma(x) \stackrel{\text{def}}{=} \mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{act}_\sigma)$$

We can recover act_σ from $\mathsf{perform}_\sigma(x)$ by setting x to be stop and using eq. (16).

Example 4.3. The node_σ operation from Example 3.7 can be encoded as:

$$\mathsf{node}_\sigma(a, b.x(b)) \stackrel{\text{def}}{=} \mathsf{fork}(b.x(b), \mathsf{wait}(a, \mathsf{act}_\sigma))$$

Equation (7) for node_σ amounts to commutativity of fork (eq. (14)), while eq. (8) can be derived:

$$\begin{aligned}\mathsf{node}_\sigma(a, b.x(b)) &= \mathsf{fork}(b.x(b), \mathsf{wait}(a, \mathsf{fork}(c.\mathsf{wait}(c, \mathsf{stop}), \mathsf{act}_\sigma))) && \text{(eq. (16))}\\ &= \mathsf{fork}(b.x(b), \mathsf{fork}(c.\mathsf{wait}(a \oplus c, \mathsf{stop}), \mathsf{wait}(a, \mathsf{act}_\sigma))) && \text{(eq. (13) and eq. (11))}\\ &= \mathsf{fork}(c.x(a \oplus c), \mathsf{wait}(a, \mathsf{act}_\sigma)) && \text{(eqs. (15) and (17))}\end{aligned}$$

Example 4.4. In the term $t = \mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_1}), \mathsf{fork}(b.\mathsf{stop}, \mathsf{act}_{\sigma_2}))$ thread ID a only refers to the thread stop and not to its child act_{σ_2}:

$$\begin{aligned}t &= \mathsf{fork}(b.\mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_1}), \mathsf{stop}), \mathsf{act}_{\sigma_1}) && \text{(eq. (15))}\\ &= \mathsf{fork}(b.\mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_1}), \mathsf{wait}(\emptyset, \mathsf{stop})), \mathsf{act}_{\sigma_1}) && \text{(eq. (10))}\\ &= \mathsf{fork}(b.\mathsf{act}_{\sigma_1}, \mathsf{act}_{\sigma_2}) && \text{(eq. (17) and eq. (10))}\end{aligned}$$

4.2 Normal Forms

In this section, we show that all the terms in the theory of dynamic threads $\mathcal{T}$ are equal, up to the derivable equality $=_{\mathcal{T}}$, to a convenient subclass of terms, which we refer to as normal forms. We define a normal form to be a term in context of the form:

$$\Gamma \mid a_1, \dots, a_n \vdash \mathsf{fork}(b_1. \dots \mathsf{fork}(b_p.\mathsf{wait}(u_{p+1}, \mathsf{stop}), \mathsf{wait}(u_p, t_p)), \dots \mathsf{wait}(u_1, t_1)) \tag{18}$$

where each subterm t_i is either an observable action act_σ or a variable $x(u_{i1}, \dots, u_{im})$, for some $x : m$ in Γ. We also require that the parameters (i.e. compound thread IDs) $u_1, \dots, u_p$, and the parameters occurring in each $t_i = x(u_{i1}, \dots, u_{im})$ satisfy closure conditions explained below.

Informally, a normal form consists of a parent that forks p child threads, waits on some collection of thread IDs, u_{p+1}, then finishes. A child must perform exactly one action, or be a free computation variable. Thanks to the term formation rules, u_i can only use thread IDs $b_1, \dots, b_{i-1}$ that have been forked earlier, or thread IDs from the context $a_1, \dots, a_n$.

Example 4.5. The term $\mathsf{fork}(b_1.\mathsf{wait}(b_1, \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1})$, from Example 4.1, is not in normal form but it is $(=_{\mathcal{T}})$-equal to the following normal form:

$$\mathsf{nf}_1 = \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(b_1 \oplus b_2, \mathsf{stop}), \mathsf{wait}(b_1, \mathsf{act}_{\sigma_2})), \mathsf{act}_{\sigma_1})$$

To show this, use eq. (16) followed by (13) and (11) to show the subterm $\mathsf{wait}(b_1, \mathsf{act}_{\sigma_2})$ equals

$$\mathsf{wait}(b_1, \mathsf{fork}(b_2.\mathsf{wait}(b_2, \mathsf{stop}), \mathsf{act}_{\sigma_2})) = \mathsf{fork}(b_2.\mathsf{wait}(b_1 \oplus b_2, \mathsf{stop}), \mathsf{wait}(b_1, \mathsf{act}_{\sigma_2})).$$

4.2.1 Closure conditions for normal forms. The closure conditions that a term of shape (18) needs to satisfy to be a normal form are that: if ID b_j appears in u_i, then all the IDs in u_j are included in u_i; the analogous condition when b_j appears in u_{ik}, where $t_i = x(u_{i1}, \dots, u_{im})$; and the IDs in u_i must appear in each u_{ik}. Imposing these closure conditions means that a normal form contains redundant information about dependencies between different threads, but this will help us formulate a correspondence between normal forms and semantic objects, in Section 5.1.2 and Theorem 5.4.

Example 4.6. The normal form from Example 4.5 satisfies these closure conditions, as does the following term, with free IDs a_1 and a_2:

$$\mathsf{nf}_2 = \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(b_1 \oplus b_2 \oplus a_1, \mathsf{stop}), \mathsf{wait}(b_1 \oplus a_1, x(b_1 \oplus a_1 \oplus a_2))), \mathsf{wait}(a_1, \mathsf{act}_{\sigma_1})) \tag{19}$$

Definition 4.7. Fix a context of computation variables Γ. For each natural number n, define the set $\mathsf{NF}_\Gamma(n)$ to contain normal forms, i.e. terms of shape (18) respecting the closure conditions above, quotiented by: the equivalence relation generated by reordering of child threads that do not depend on each other, and by the semi-lattice equations on compound thread IDs. Moreover, NF_Γ has a functorial action on relations $R \in [n] \times [n']$ given by parameter substitution.

This definition implies that two representatives of the same equivalence class of normal forms are also $(=_{\mathcal{T}})$-equal in the theory of dynamic threads (Figure 5) because the reordering of independent child threads corresponds to eq. (14), and $=_{\mathcal{T}}$ is closed under the semi-lattice equations by definition.

Theorem 4.8. *Every term $\Gamma \mid a_1, \ldots, a_n \vdash t$ in the theory of dynamic threads $\mathcal{T}$ is derivably equal to a, not necessarily unique, equivalence class of normal forms from $\mathsf{NF}_\Gamma(n)$.*

We prove the theorem by induction on terms using the equations from Figure 5. As a corollary, normal forms map surjectively onto (equivalence classes of) terms. We have not shown for now that every term is equal to a *unique* equivalence class of normal forms, we prove this in Corollary 5.5.

5 A REPRESENTATION THEOREM FOR THE THEORY OF DYNAMIC THREADS

In this section we interpret the parameterized theory of dynamic threads from Section 4 semantically, using a notion of labelled poset similar to that used in Section 3.3.3. In Section 5.1 we discuss labelled posets informally, then in Sections 5.2 and 5.3 we give their formal definition and show that they form a free model. In Section 6, we show that this model is in a certain sense complete.

5.1 Labelled Posets with Holes by Example

We introduce labelled posets by example and use terms from the theory of dynamic threads (Section 4) to motivate them. We define labelled posets formally in Definitions 5.1 and 5.2.

5.1.1 Labelled posets represent terms. Consider Figure 6 (a): two elements of the poset are labelled by observable actions σ_1 and σ_2. The solid lines represent causal dependencies and induce a partial order: σ_1 must happen before σ_2. All posets contain a distinguished maximal element s which represents the *end of the main thread*; we will use s when defining an operation analogous to fork for posets. In Figure 6 (b), σ_2 is part of the main thread but σ_1 is not, as discussed in Example 4.1. Elements of the poset may be labelled by computation variables, for example, $x : 0$ in Figure 6 (c).

A term's free thread IDs appear as minimal elements in its poset, e.g. in Figure 6 (d) they are numbered 1 and 2. Bound thread IDs do not appear in the poset. Figure 6 (d) depicts the poset of:

$$x : 1 \mid a_1, a_2 \vdash \mathsf{fork}(b_1.\mathsf{wait}(b_1, x(b_1 \oplus a_2)), \mathsf{wait}(a_1, \mathsf{act}_{\sigma_1}))$$

In the poset, there is one element labelled x which has one input. The dotted line is not part of the partial order; it represents the thread IDs passed to variable x and means that x *may* wait for a_2, depending on what computation x is. The dotted line induces a relation called *visibility* which is assumed to be downward-closed with respect to the partial order (the solid line) and to contain all elements below x in the partial order; therefore we omit to draw dotted lines from σ_1 and 1 into x.

5.1.2 Labelled posets are normal forms. Recall that a normal form has the shape below, where each t_i is either one observable action or a computation variable from Γ:

$$\Gamma \mid a_1, \ldots, a_n \vdash \mathsf{fork}(b_1. \ldots \mathsf{fork}(b_p.\mathsf{wait}(u_{p+1}, \mathsf{stop}), \mathsf{wait}(u_p, t_p)), \ldots \mathsf{wait}(u_1, t_1))$$

The corresponding poset has n minimal elements corresponding to the free thread IDs $a_1, \ldots, a_n$. There is one labelled element for each of the terms t_1 to t_p. The parent, which stops, corresponds to the distinguished maximal element s. The partial order (solid line) encodes the dependencies given by the compound thread IDs u_1 to u_{p+1}. The visibility relation (dotted line) corresponds to the thread IDs passed to a variable $t_i = x(u_{i1}, \ldots, u_{im})$. The closure conditions on normal forms from Section 4.2.1 correspond to the transitivity of the partial order and to the fact that the visibility relation is downward-closed with respect to the partial order.

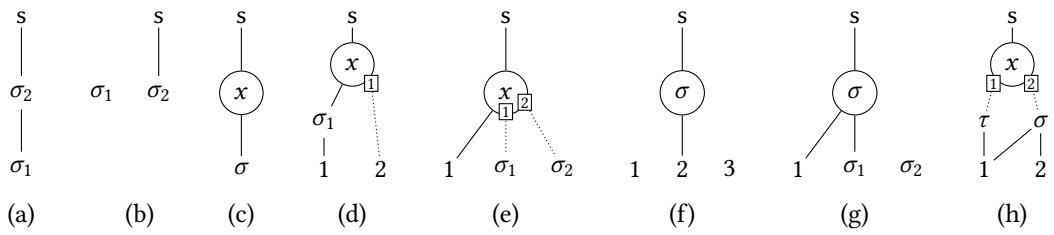


Fig. 6. Examples of labelled posets. (a) is the term $\mathsf{fork}(a.\mathsf{wait}(a, \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1})$. (b) is $\mathsf{fork}(a.\mathsf{act}_{\sigma_2}, \mathsf{act}_{\sigma_1})$. (c) is $\mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{act}_{\sigma})$. (d) is the normal form in eq. (19). (g) is the result of substituting (f) for $(x : 2)$ in (e). (h) is the representation of Figure 4 (c) using the notion of labelled poset introduced in this section.

5.1.3 Substitution for labelled posets. Just like terms in a parameterized theory, labelled posets admit substitution of another poset for a computation variable; the formal definition is discussed in Section 5.3.2. In Figure 6, posets (e) and (f) represent respectively the terms

$$x : 2 \mid a_1 \vdash \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_1, x(b_1, b_2)), \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1}) \qquad - \mid a_1, b_1, b_2 \vdash \mathsf{wait}(b_1, \mathsf{act}_{\sigma})$$

while (g), the result of substituting (f) for $x : 2$ in (e), is the term:

$$- \mid a_1 \vdash \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_1, \mathsf{wait}(b_1, \mathsf{act}_{\sigma})), \mathsf{act}_{\sigma_2}), \mathsf{act}_{\sigma_1})$$

The input 1 of both (e) and (f) gets identified, while inputs 2 and 3 of (f) are mapped to the two inputs of variable x.

Substitution of a compound thread ID for an input of the poset corresponds to parameter substitution on terms. We define this operation on posets in Definition 5.3.

5.1.4 Connection to labelled posets from Section 3.3.3 and to ordinary labelled posets. The Σ-labelled posets with n inputs and m outputs from Definition 3.11 are a special case of the labelled posets from this section. For example, the poset in Figure 4 (c), which corresponds to the term $x : 2 \mid a_1, a_2 \vdash \mathsf{node}_{\sigma}(a_1 \oplus a_2, b_1.\mathsf{node}_{\tau}(a_1, b_2.x(b_2, b_1)))$, is shown in Figure 6 (h). The two inputs of variable x correspond to the two outputs of the poset from Figure 4 (c).

A labelled poset with no inputs and no elements labelled by computation variables is an ordinary labelled poset in the sense of Definition 2.3, i.e. a partially ordered set $(X, \leq)$ with a function $X \to \Sigma$, provided we regard s as a label as well.

5.2 Labelled Posets with Holes Formally

The following two definitions formalize the ideas about labelled posets from the previous section.

Definition 5.1. Let $\Gamma = x_1 : m_1, \ldots, x_k : m_k$ be a context of computation variables, and Σ be a set of observable actions. A (Γ, Σ)-*labelled poset with n inputs* $G = \langle V_1, V_2, \leq_G, l_1, l_2 \rangle$ consists of:

- the set of n input vertices $[n] = \{1, \ldots, n\}$;
- finite disjoint sets of vertices V_1 (labelled by actions) and V_2 (labelled by variables);
- a distinguished vertex s;
- a partial order $\leq_G$ on the underlying set $|G| \stackrel{\text{def}}{=} [n] \uplus V_1 \uplus V_2 \uplus \{\mathsf{s}\}$ (depicted by solid lines);
- a labelling function $l_1 : V_1 \to \Sigma$, from the vertices in V_1 to observable actions;
- a function l_2 that labels the vertices in V_2 with variables $(x : m)$ from Γ:

$$l_2 : V_2 \to \sum_{(x:m) \in \Gamma} \big(f : [m] \to \mathcal{P}(|G|)\big)$$

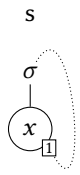


Fig. 7. Example of an ill-formed poset.

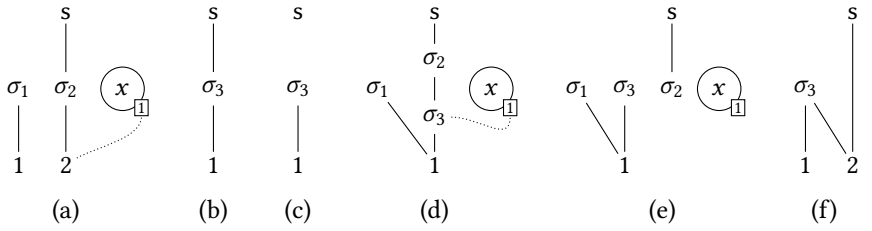


Fig. 8. Examples of forking on labelled posets: forking parent (a) with child (b) gives (d). If instead, the child is (c), the result of forking is (e). (f) is the result of applying wait_1 to (c).

and depending on the arity m of this variable, l_2 also assigns a function $f : [m] \to \mathcal{P}(|G|)$ into the powerset of $|G|$. (Each f is depicted by dotted lines).

If there are no inputs and no vertices labelled by variables, $n = 0$ and $V_2 = \emptyset$, then a labelled poset becomes an ordinary labelled poset. An isomorphism $\alpha : G \to G'$ between labelled posets consists of two bijections $\alpha_1 : V_1 \to V_1'$ and $\alpha_2 : V_2 \to V_2'$ which preserve the two labelling functions, in a suitable sense, and such that $\mathsf{id}_{[n]} \uplus \alpha_1 \uplus \alpha_2 \uplus \mathsf{id}_{\mathsf{s}}$ preserves and reflects the partial order.

To obtain the correspondence between labelled posets and normal forms explained in Section 5.1.2, we restrict our attention to labelled posets that are well-formed.

Definition 5.2. An (Γ, Σ)-labelled poset with n inputs $G = \langle V_1, V_2, \leq_G, l_1, l_2 \rangle$ is *well-formed* if:

(1) the n inputs are minimal and s is maximal, with respect to the partial order $\leq_G$;
(2) for each $e \in V_2$ such that $l_2(e) = (x : m,\ f : [m] \to \mathcal{P}(|G|))$ and for each $i \in [m]$: $e \in f(i)$ and $\mathsf{s} \notin f(i)$, and $f(i)$ is downward-closed with respect to $\leq_G$.
(3) Consider the *visibility* relation $S \subseteq |G| \times |G|$ induced by the labelling function l_2:
$$(e', e) \in S \iff e \in V_2 \text{ and if } l_2(e) = (x : m, f) \text{ then } e' \text{ is in the image of } f.$$
The transitive closure of the relation $(\leq_G) \cup S$ is anti-symmetric.

All the labelled posets discussed in Section 5.1 satisfy the well-formedness conditions above. Requiring that s is maximal and $s \notin f(i)$ ensures that child threads do not know the ID of the main thread. Downward-closure of $f(i)$ and $e \in f(i)$ correspond to some of the closure conditions on normal forms (Section 4.2.1). Condition (3) ensures that, when taking into account the visibility relation, there are no cycles in the labelled poset. Overall, well-formedness ensures that a labelled poset can be linearly ordered into a normal form of shape (18). For example, the poset in Figure 7 cannot be linearized and does not satisfy condition (3).

5.3 The Labelled Poset Model and Representation Theorem

5.3.1 Model structure. In order to build a model for the theory of dynamic threads out of labelled posets, we organize them into a functor T_Γ of type $\mathsf{FinRel} \to \mathsf{Set}$ as follows.

Definition 5.3. Let Γ be a context of computation variables. For each natural number n, define the set $T_\Gamma(n)$ to contain *isomorphism classes of well-formed* (Γ, Σ)*-labelled posets with* n *inputs.* The functorial action on relations $R \subseteq [n] \times [n']$ acts on the inputs of a poset by updating $i \leq e$ to $i' \leq e$ if $(i, i') \in R$, and updating the labelling function l_2 accordingly.

We can equip T_Γ with a model structure, in the sense of Definition 3.8, for the fork, wait, stop and act_σ operations of the theory of dynamic threads. For each natural number n, we define:
$$\mathsf{fork}_n : T_\Gamma(n+1) \times T_\Gamma(n) \to T_\Gamma(n).$$

The labelled vertices of $\mathsf{fork}_n(G_1, G_2)$ are the union of those from G_1 and G_2 and the labels are preserved. The partial order of $\mathsf{fork}_n(G_1, G_2)$ is obtained by connecting the $(n+1)$-th input of G_1 to the s element of G_2 and closing under transitivity. The visibility relation is obtained via the same connection from those of G_1 and G_2.

Figure 8 shows an example of forking. The parent (a) and the children (b) and (c) correspond respectively to the terms:

$$x : 1 \mid a_1, a_2 \vdash t_1 = \mathsf{fork}(b_1.\mathsf{fork}(b_2.\mathsf{wait}(a_2, \sigma_2), x(a_2)), \mathsf{wait}(a_1, \sigma_1))$$

$$x : 1 \mid a_1 \vdash t_2 = \mathsf{wait}(a_1, \mathsf{act}_{\sigma_3}) \qquad x : 1 \mid a_1 \vdash t_3 = \mathsf{fork}(c.\mathsf{stop}, \mathsf{wait}(a_1, \mathsf{act}_{\sigma_3}))$$

while (d) corresponds to $\mathsf{fork}(a_2.t_1, t_2)$ and (e) corresponds to $\mathsf{fork}(a_2.t_1, t_3)$.

The operation $\mathsf{wait}_n : T_\Gamma(n) \to T_\Gamma(n+1)$ adds a new input $n+1$ and connects it to all the labelled elements and to s. Figure 8 (f) is an example; it represents the term $- \mid a_1, a_2 \vdash \mathsf{wait}(a_2, t_3)$.

The operation $\mathsf{stop}_n : 1 \to T_\Gamma(n)$ picks out the poset with only the n inputs and s as elements, no labelled vertices, and with the discrete partial order. The $\mathsf{act}_{\sigma,n} : 1 \to T_\Gamma(n)$ operation gives a poset with one vertex labelled by σ, directly below s in the partial order, and with the n inputs not connected to anything.

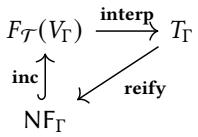
5.3.2 Main theorem. We now show that labelled posets form a model for the theory of dynamic threads and that this model is free. Recall that we described the term model $F_{\mathcal{T}}(V_\Gamma)$ as a free model in Section 3.4.

Theorem 5.4 (Representation Theorem). *For each context Γ, the functor T_Γ from Definition 5.3 and the operations on labelled posets* fork, wait, stop, act_σ *respect the equations of the theory of dynamic threads from Figure 5, and thus form a model of the theory, in the sense of Definition 3.9.*

Moreover, T_Γ is isomorphic to the term model $F_{\mathcal{T}}(V_\Gamma)$ of the theory of dynamic threads.

Recall that in Section 5.1.3 we informally discussed substitution for labelled posets. Using the isomorphism of models in the theorem above we can define substitution by translating a labelled poset into an equivalence class of terms, using term substitution, then translating back to a poset.

5.3.3 Proof sketch of the Representation Theorem. To prove Theorem 5.4 we consider the diagram below. Recall that the functor NF_Γ, from Definition 4.7, contains equivalence classes of normal forms, and $F_{\mathcal{T}}(V_\Gamma)$ contains equivalence classes of terms.



The **inc** map is given by the inclusion of normal forms into terms. The map **interp** is the unique map obtained by instantiating the universal property of the free model $F_{\mathcal{T}}(V_\Gamma)$ (Definition 3.13) for a suitable $V_\Gamma \to T_\Gamma$ that maps computation variables to labelled posets; **interp** is essentially given by the interpretation $[\![-]\!]_{T_\Gamma}$ of terms in the labelled poset model, from Section 3.3.2.

For each natural number n, we define a function $\mathbf{reify}_n$ that linearizes a labelled poset into a normal form, using the intuition from Section 5.1.2. To show that this gives a well-defined function, natural in n, we use the conditions for a well-formed labelled poset (Definition 5.2), the closure conditions and the equivalence relation on normal forms (Section 4.2.1).

We show that **reify** is both a left and a right inverse to the composite **interp** $\circ$ **inc**. To show it is a left inverse we use induction on the number of child threads in a normal form; for the right inverse, we use induction on the number of labelled elements in a poset. Knowing that **inc** is surjective (Theorem 4.8) means that **interp** is an isomorphism. We already know from the definition of the free model that **interp** is a homomorphism of models.

Corollary 5.5. *The inclusion of equivalence classes of normal forms into equivalence classes of terms is injective. By Theorem 4.8, every term is equal to a unique equivalence class of normal forms.*

6 A COMPLETENESS THEOREM FOR THE THEORY OF DYNAMIC THREADS

In Theorem 5.4, we have seen that terms $\Gamma \mid a_1, \dots, a_n \vdash t$ in the theory of dynamic threads correspond exactly to (Γ, Σ)-labelled posets with n inputs. When the context Γ is empty and $n = 0$, these labelled posets are in fact ordinary Σ-labelled posets i.e. a partially ordered set $(X, \leq)$ with a function $X \to \Sigma$. The next theorem shows that our axiomatization of (Γ, Σ)-labelled posets from Figure 5 is complete with respect to an equivalence relation induced by isomorphism of ordinary Σ-labelled posets.

Given a term in context $\Gamma \mid \Delta \vdash t$, a closing substitution γ is one that assigns to each variable $(x : m)$ from Γ a term $\gamma(x) = (- \mid \Delta, b_1, \dots, b_m \vdash s)$ with no free computation variables, so that $- \mid \Delta \vdash t[\gamma]$ holds. A closing context $C[-]$ is a term with a hole such that given a term $- \mid \Delta \vdash t$, the judgement $- \mid - \vdash C[t]$ holds.

Theorem 6.1 (Completeness). *Consider terms $\Gamma \mid \Delta \vdash t_1, t_2$ in the theory of dynamic threads. If for all closing substitutions γ and for all closing contexts $C[-]$ the two terms are equal, meaning $- \mid - \vdash C[t_1[\gamma]] = C[t_2[\gamma]]$, then $\Gamma \mid \Delta \vdash t_1 = t_2$.*

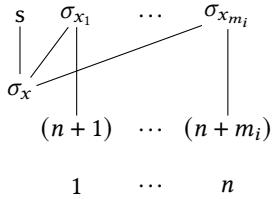
Proof sketch. First consider terms with no free computation variables $- \mid a_1, \dots, a_n \vdash t_1, t_2$. Consider the context which binds each free thread ID to an observable action, and adds an observable action $\mathsf{act}_{\sigma_{n+1}}$ at the end of the main thread:

$$C[-] = \mathsf{fork}(a_1. \dots \mathsf{fork}(a_n.\mathsf{fork}(a_{n+1}.\mathsf{wait}(a_{n+1}, \mathsf{act}_{\sigma_{n+1}}), [-]), \mathsf{act}_{\sigma_n}), \dots \mathsf{act}_{\sigma_1})$$

We assume $\sigma_1, \dots, \sigma_{n+1}$ are distinct from any observable actions occurring in t_1 and t_2.

Recall that $C[t_1]$ and $C[t_2]$ are interpreted as labelled posets in $T_\emptyset(0)$. If they are equal then by Theorem 5.4 there is an isomorphism of labelled posets between $[\![C[t_1]]\!]$ and $[\![C[t_1]]\!]$. From this we can construct an isomorphism between $[\![t_1]\!], [\![t_2]\!] \in T_\emptyset(n)$ and deduce $t_1 = t_2$.

Now consider terms in context $\Gamma \mid a_1, \dots, a_n \vdash t_1, t_2$. Consider the substitution γ which maps each computation variable $(x_i : m_i)$ from Γ to the term $- \mid a_1, \dots, a_n, b_1, \dots, b_{m_i} \vdash s_i$ depicted on the right as a labelled poset with $n + m_i$ inputs. We assume that the actions $\sigma_x, \sigma_{x_1}, \dots, \sigma_{x_{m_i}}$ are distinct for each computation variable x, and distinct from the actions in t_1 and t_2. Note, we may encode "new" actions as distinct combinations.



Assuming we have an isomorphism of labelled posets $\alpha : [\![t_1[\gamma]]\!] \to [\![t_2[\gamma]]\!]$, we can construct an isomorphism $\alpha' : [\![t_1]\!] \to [\![t_2]\!]$. On labelled elements, α' acts the same as α, forgetting about the elements labelled $\sigma_{x_1}, \dots, \sigma_{x_{m_i}}$. The elements labelled σ_x in $[\![t_1[\gamma]]\!]$ correspond exactly to the elements labelled $(x : m)$ in $[\![t_1]\!]$. To show α' is an isomorphism of (Γ, Σ)-labelled posets with n inputs, we use the well-formedness of labelled posets that represent terms. In particular, we rely on condition (3) from Definition 5.2 which says that the visibility relation does not induce cycles. □

7 DENOTATIONAL SEMANTICS, SOUNDNESS, ADEQUACY AND FULL ABSTRACTION

Sections 3–6 have developed a theory for an algebraic language based on fork and wait. The idea of algebraic effects is that this can easily be extended to a full language. To demonstrate this, we return to the programming language from Section 2, outlining how it can be given a semantics by using our complete representation, which is sound, adequate, and fully abstract at first order with respect to the operational semantics.

7.1 Interpretation

7.1.1 Summary. We interpret all types A of the programming language as functors $[\![A]\!] \in \mathrm{Set}^{\mathbf{FinRel}}$. The idea is that $[\![A]\!](w)$ is the set of interpretations of values of type A in world w, i.e. $\vdash^{\mathbf{v}}_w v : A$. Similarly, we interpret contexts Γ as functors $[\![\Gamma]\!] \in \mathrm{Set}^{\mathbf{FinRel}}$, i.e. world-indexed sets of valuations.

We will interpret value expressions $\Gamma \vdash^{\mathrm{v}} v : A$ as natural transformations $[\![v]\!] : [\![\Gamma]\!] \to [\![A]\!]$ in $\mathrm{Set}^{\mathbf{FinRel}}$. To interpret computation expressions, we build a monad T on $\mathrm{Set}^{\mathbf{FinRel}}$ from the representation, and interpret expressions $\Gamma \vdash^{\mathrm{c}} t : A$ as natural transformations $[\![t]\!] : [\![\Gamma]\!] \to T[\![A]\!]$.

7.1.2 Interpretation of first-order types. The interpretation of the type of thread IDs is:

$$[\![\mathsf{tid}]\!] = \mathbf{FinRel}(1, -) \quad \textit{i.e.}\ [\![\mathsf{tid}]\!](w) \cong \{w' \mid w' \subseteq w\}.$$

We thus interpret (compound) thread id values in world w as subsets of (non-compound) tids in w.

The interpretation of product and sum types uses the well-known and canonical categorical structure of the functor category $\mathrm{Set}^{\mathbf{FinRel}}$. Recall that products and coproducts in functor categories are computed pointwise. Moreover, we have a strong connection with Section 3.4, since for a context $(x_1 : m_1 \dots x_k : m_k)$, the functor of variables is an interpretation of a type:

$$[\![\textstyle\prod_{i=1}^k A_i]\!](p) = \prod_{i=1}^k [\![A_i]\!](p) \quad [\![\textstyle\sum_{i=1}^k A_i]\!](p) = \biguplus_{i=1}^k [\![A_i]\!](p) \quad V_{x_1:m_1 \dots x_k:m_k} \cong [\![\textstyle\sum_{i=1}^k \mathsf{tid}^{m_i}]\!]. \quad (20)$$

In fact, every first-order type is isomorphic to one of this form, since products distribute over sums.

7.1.3 Monad. We extend the parameterized algebraic theory for fork and wait to a strong monad on $\mathrm{Set}^{\mathbf{FinRel}}$, following [41, 42]. Recall (e.g. [35]) that a plain algebraic theory (such as monoids) induces a monad (such as lists) on the category of sets, by letting $T(X)$ be the free model of the theory on the set X. For a parameterized algebraic theory (such as the theory of fork and wait), we can define a strong monad T on the category $\mathrm{Set}^{\mathbf{FinRel}}$ by letting $T(X)$ be the free model of the theory on the functor $X \in \mathrm{Set}^{\mathbf{FinRel}}$. Recall that we have already characterized the free model over V_Γ, in Theorem 5.4. Thus for first-order types, we can use this characterization of free models.

(Aside: every strong monad on $\mathrm{Set}^{\mathbf{FinRel}}$ that preserves sifted colimits arises from a parameterized algebraic theory in this way, see [42]. This gives a monad-theory correspondence.)

7.1.4 Interpretation of higher order types. It is well-known that the category $\mathrm{Set}^{\mathbf{FinRel}}$ is cartesian closed. For functors G, H we have a functor H^G determined by the currying isomorphism: to give a natural transformation $F \times G \to H$ is to give a natural transformation $F \to H^G$. We then interpret the function type $A \to B$ using the monad, and this cartesian closed structure:

$$[\![A \to B]\!] = (T[\![B]\!])^{[\![A]\!]}.$$

7.1.5 Interpretation of terms. The interpretation of the concurrency-specific base constructions is through the fact that $T(X)$ is always a model of the parameterized algebraic theory, as follows. These are sometimes called the 'generic effects' of the algebraic operations [33].

$$[\![\underline{\mathsf{fork}}]\!] = \lambda().\mathsf{fork}(\lambda a.\eta(\mathsf{inj}_1\,(a)), \eta(\mathsf{inj}_2\,(\,))) : 1 \to T([\![\mathsf{tid}]\!] + 1) \quad [\![\underline{\mathsf{stop}}]\!] = \lambda().\mathsf{stop} : 1 \to T(0)$$
$$[\![\underline{\mathsf{wait}}]\!] = \lambda a.\mathsf{wait}(a, \eta()) : [\![\mathsf{tid}]\!] \to T(1) \quad [\![\underline{\mathsf{act}}_\sigma]\!] = \lambda().\mathsf{act}_\sigma : 1 \to T(0)$$

If we elide (20) we can also regard these as the canonical terms:

$$\underline{\mathsf{fork}}() = \mathsf{fork}(a.x(a), y()) \quad \underline{\mathsf{wait}}(a) = \mathsf{wait}(a, x) \quad \underline{\mathsf{stop}}() = \mathsf{stop} \quad \underline{\mathsf{act}}_\sigma() = \mathsf{act}_\sigma$$

The remainder of the interpretation of value and computation terms is the long-established interpretation of a call-by-value language in a bicartesian closed category with a monad [26]. The language constructs (sums, products and functions) match up with the categorical structure (coproducts, products, and cartesian closure).

The interpretation of $\mathsf{let}\, x = \dots \,\mathsf{in}\, \dots$ is given using the monad strength and the bind. For first-order types, this amounts to the substitution of the parameterized algebraic theory. This is informative to spell out. Let $A = \sum_{i=1}^k \mathsf{tid}^{m_i}$ and $B = \sum_{i=1}^{k'} \mathsf{tid}^{m'_i}$. Consider program expressions:

$$\vdash^{\mathrm{c}}_{a_1 \dots a_p} t : A \qquad x : A \vdash^{\mathrm{c}}_{a_1 \dots a_p} u : B$$

and we explain $[\![\mathsf{let}\, x = t \,\mathsf{in}\, u]\!]$. For $1 \le i \le k$, let $\vdash^{\mathbf{c}}_{a_1 \dots a_p, b_1 \dots b_{m_i}} u_i \stackrel{\text{def}}{=} u[^{\mathsf{inj}_i\,(\vec{b})}/_x] : B$, where each $\mathsf{inj}_i\,(\vec{b})$ has type $\sum_{i=1}^{k} \mathsf{tid}^{m_i}$. Then, by Theorem 5.4, we can regard $[\![t]\!] \in T(V_{m_1 \dots m_k})(\vec{a})$ and each $[\![u_i]\!] \in T(V_{m'_1 \dots m'_{k'}})(\vec{a}, \vec{b})$ as terms

$$x_1 : m_1 \dots x_k : m_k \mid a_1 \dots a_p \vdash \bar{t} \qquad x'_1 : m'_1 \dots x'_{k'} : m'_{k'} \mid a_1 \dots a_p, b_1 \dots b_{m_i} \vdash \bar{u}_i$$

in the parameterized algebraic theory. Then the interpretation of $\vdash^{\mathbf{c}}_{a_1 \dots a_p} \mathsf{let}\, x = t \,\mathsf{in}\, u : B$ in $T(V_{m'_1 \dots m'_{k'}})(\vec{a})$ amounts to the following substituted term in the parameterized algebraic theory:

$$x'_1 : m'_1 \dots x'_k : m'_{k'} \mid a_1 \dots a_p \vdash \bar{t}[^{\bar{u}_i}/_{x_i}]$$

For example, $[\![\underline{\mathsf{perform}}_\sigma()]\!] \in T(1)(0)$ is the semantics of both the program for $\underline{\mathsf{perform}}_\sigma()$ on the left, and the term in the parameterized algebraic theory on the right:

$$\mathsf{let}\, x = \underline{\mathsf{fork}}() \,\mathsf{in}\, \mathsf{case}\, x \,\mathsf{of}\, \begin{Bmatrix} \mathsf{inj}_1(a) \Rightarrow \underline{\mathsf{wait}}(a); \mathsf{return}\,(), \\ \mathsf{inj}_2() \Rightarrow \underline{\mathsf{act}}_\sigma(), \end{Bmatrix} : 1 \qquad x : 0 \mid - \vdash \mathsf{fork}(a.\mathsf{wait}(a, x), \mathsf{act}_\sigma).$$

7.2 Adequacy, Contextual Equivalence, Soundness, and Full Abstraction

Lemma 7.1 (Adequacy). *For all* $\vdash^{\mathbf{c}}_{\emptyset} t : 0$, *we have* $\langle [a]t \rangle \Downarrow [\![t]\!]$.

Proof outline. We prove this in three steps.

(1) We extend term interpretations $[\![t]\!]$ to well-formed configurations $[\![C]\!]$.
(2) We show a soundness property for the reduction relation: semantic interpretation is preserved by reduction. For example, if $C \longrightarrow C'$ then $[\![C]\!] = [\![C']\!]$. This is a straightforward induction proof, but the statement is subtle, requiring accumulating the action labels.
(3) We pick a reduction sequence from $\langle [a]t \rangle$, noting by Proposition 2.7 that the choice of sequence doesn't matter and that it will terminate. A finished configuration only has the waiting relation $\preceq$ remaining, and all the stopped threads. With the accumulated action labels, this labelled poset is $[\![t]\!]$, because reduction preserves semantic interpretations. □

Definition 7.2. Let Γ be a typing context and A a type. A *program context* $C[-]$ for Γ, A is a program of type 0 with a hole of type A. Thus, if $\Gamma \vdash^{\mathbf{c}}_{\emptyset} t : A$ then $\vdash^{\mathbf{c}}_{\emptyset} C[t] : 0$.

Two programs $\Gamma \vdash^{\mathbf{c}}_{\emptyset} t, u : A$ are *contextually equivalent*, written $t \stackrel{\text{ctx}}{=} u$, if for every (Γ, A) context $C[-]$, letting $(\cong)$ denote isomorphism of labelled posets, we have that

$$\langle [a]C[t] \rangle \Downarrow (P, \ell_P) \;\&\; \langle [a]C[u] \rangle \Downarrow (Q, \ell_Q) \implies (P, \ell_P) \cong (Q, \ell_Q)$$

By Proposition 2.7, the (P, ℓ_P) and (Q, ℓ_Q) are uniquely determined by $C[t]$ and $C[u]$ respectively.

Proposition 7.3 (Soundness). *Suppose that* $\Gamma \vdash^{\mathbf{c}}_{\emptyset} t, u : A$. *If* $[\![t]\!] = [\![u]\!]$ *then* $t \stackrel{\text{ctx}}{=} u$.

Proof. We first handle the case where $A = 0$. In that case, notice that $[\![t]\!] \in T(0)(0)$ can be regarded as a labelled poset, by Theorem 5.4. We deduce the result from Lemma 7.1 as follows. We consider any two terms in any typing context, $\Gamma \vdash^{\mathbf{c}}_{\emptyset} t, u : A$, and any (Γ, A)-context, $C[-]$. Suppose that $[\![t]\!] = [\![u]\!]$. From Lemma 7.1, $\langle [a]C[t] \rangle \Downarrow [\![C[t]]\!]$ and also $\langle [a]C[u] \rangle \Downarrow [\![C[u]]\!]$. Since the denotational semantics is compositional and $[\![t]\!] = [\![u]\!]$, also $[\![C[t]]\!] = [\![C[u]]\!]$. Thus $t \stackrel{\text{ctx}}{=} u$. □

In particular, the standard β/η laws are sound, as are all the equations in Figure 5, such as (13):

$$\underline{\mathsf{wait}}(b); \underline{\mathsf{fork}}() \;=\; \mathsf{let}\, x = \underline{\mathsf{fork}}() \,\mathsf{in}\, \underline{\mathsf{wait}}(b); \mathsf{return}\, x$$

Theorem 7.4 (Full abstraction at first order). *Suppose that* $a_1 : A_1 \dots a_p : A_p \vdash^{\mathbf{c}}_{\emptyset} t, u : B$ *and* $A_1 \dots A_p$ *and* B *are all first order (no function types). Then* $[\![t]\!] = [\![u]\!]$ *if and only if* $t \stackrel{\text{ctx}}{=} u$.

Proof. From left to right follows from Theorem 7.3.

From right to left, we first consider the case where $p = 0$ and $B = 0$. Then contextual equivalence with the empty context in particular, together with Lemma 7.1, gives $[\![t]\!] = [\![u]\!]$.

We next consider the case where $A_1 = A_2 = \ldots A_p = \mathsf{tid}$ and $B = \sum_{i=1}^{k} \mathsf{tid}^{m_i}$. Suppose $t \overset{\text{ctx}}{=} u$. Via (20), $[\![t]\!], [\![u]\!] \in T(V_{x_1:m_1 \ldots x_k:m_k})(p)$, that is, t and u are interpreted directly in the parameterized algebraic theory. We must show that they are equal. By Theorem 6.1, it suffices to show that $C[[\![t]\!]][\gamma] = C[[\![u]\!]][\gamma]$ for all algebraic contexts $C[-]$ and algebraic substitutions γ. We deduce this by converting $C[-]$ and γ into a program context ('full definability' at first order) so that we can use the contextual equivalence $t \overset{\text{ctx}}{=} u$.

First, we note that the programming language supports algebraic operations at all types, via the generic effects: $\mathsf{act}_\sigma = \underline{\mathsf{act}}_\sigma()$, $\mathsf{stop} = \underline{\mathsf{stop}}()$ and

$$\mathsf{fork}(t, u) = \mathsf{case}\, \underline{\mathsf{fork}}() \,\mathsf{of}\, \{\mathsf{inj}_1(a) \Rightarrow t,\ \mathsf{inj}_2(()) \Rightarrow u\} \qquad \mathsf{wait}(a, t) = \underline{\mathsf{wait}}(a); t \tag{21}$$

We use this to convert the algebraic context $C[-]$ to a program context that binds the free variables $a_1 \ldots a_p$. Moreover, each 'substitutand' $\gamma(x_i)$ has no computation variables, and hence can also be regarded as a program of type 0 under (21). Now we define the computation program expression

$$t[\gamma] \overset{\text{def}}{=} \mathsf{case}\, t \,\mathsf{of}\, \{\mathsf{inj}_i(\vec{a}) \Rightarrow \gamma(x_i)\}_{i=1}^{k}$$

so that $[\![t[\gamma]]\!] = [\![t]\!][\gamma] \in T(0)(p)$. Thus $C[[\![t]\!]][\gamma] = [\![C[t][\gamma]]\!] = [\![C[u][\gamma]]\!] = C[[\![u]\!]][\gamma]$ as required. Finally, we deduce the full result by using β/η laws for sums and the fact that every first-order type is definably isomorphic to one of the form (20). □

8 FURTHER RELATED AND FUTURE WORK AND CONCLUDING REMARKS

Before concluding, we discuss any additional related work and future directions our work enables.

8.1 Further related work

Algebraic effects for concurrency. As briefly discussed in Section 1, algebraic theories have been used to axiomatize features of process calculi, including in the style of algebraic effects. This includes an algebraic-effects analysis of name creation and communication of names over channels in the π-calculus [40], and a treatment of features of CSP such as action, choice and concealment [46] using algebraic effects and handlers. From a programming language perspective, concurrency in the presence of nondeterminism and global shared state has been modelled using algebraic effects by Abadi and Plotkin [2] and Dvir et al. [9, 11]. As discussed in Section 1.4, our work differs from this previous work in that parallel composition of programs (i.e. forking) is an operation in the equational axiomatization, whereas in previous work it was defined on top of the algebraic effects presentation. The key ingredient that makes this possible is that we treat thread IDs as primitive and use the framework of parameterized algebraic theories to capture thread creation.

Trace semantics. Brookes's influential work [7] models a preemptive concurrent programming language with global shared state. Programs denote closed sets of traces; these traces represent a protocol involving the changes to memory by the program and its environment. This form of semantics is robust under variation and extensions [4, 45, 47], including variations to weak memory models [10, 17]. Dvir et al. [11] give a *two-sorted* algebraic theory for Brookes-like traces. Their representation theorem recovers Brookes's monad when restricted to one of the sorts. Interestingly, the same representation recovers Abadi and Plotkin's [2] monad for *cooperative* concurrent programming with shared state when restricted to the other sort. Both Dvir et al.'s and Abadi and Plotkin's presentations presuppose non-deterministic choice as an algebraic operation. In contrast,

in our parameterized algebraic theory the non-deterministic behaviour emerges from the more primitive behavior of thread forking.

Effect handlers for concurrency. Effect handlers arose from the theoretical study of algebraic effects [36] as a way of supporting non-algebraic effects, such as an operation for catching exceptions. They were quickly adopted as a general feature for modular programming with effects [20], and are central to how concurrency is currently implemented in OCaml 5 [39] and in WebAssembly [30]. They provide the basis for a whole range of different concurrency effects such as actors, async/await, coroutines, generators, and green threads. Alas, the practice of programming with effect handlers departs substantially from the established theory: we do no yet how to specify the semantics of these effects using any kind of equational axiomatisation, let alone as an algebraic effect.

Hazel is a separation logic [8] for effect handlers built on the concurrent separation logic Iris [19] in the Rocq proof assistant. Hazel provides a powerful framework for reasoning about concurrency effects implemented as effect handlers, but it is quite a departure from the elementary equational reasoning provided by the theory of algebraic effects and gives little semantic insight into the effects being defined. In contrast, our work characterises a particular concurrency effect (dynamic threads) as an algebraic effect (specifically a parameterised algebraic effect) corresponding to a natural denotational model. Future work may adapt and extend our approach to support a broad range of different concurrency effects or connect to effect handlers and programming practice.

8.2 Future work

The framework of algebraic theories allows for modular combination of effects [16]. We could use this to combine concurrency based on dynamic threads with other effects such as global and local state [35] which is shared between threads, and to model probabilistic scheduling of threads.

We have used labelled posets (pomsets), which are standard in the study of true concurrency e.g. [37], as the notion of observation in our operational semantics. We hope our denotational model can connect in the future with an operational semantics based on interleaving traces, which is more standard in process calculus e.g. [25].

Possible semantic variations of fork and wait abound, such as waiting for a thread and all its descendants to finish, or limiting the number of threads that can exist at one time. Another extension involves threads that finish with a value rather than with the empty type, and so the wait operation returns that value to the parent. This extension is an abstract form of inter-thread communication.

8.3 Concluding remarks

We have studied the semantics of dynamic creation of threads using the framework of parameterized algebraic theories, by treating thread IDs as abstract parameters. In Section 4 we gave an algebraic theory that axiomatizes operations such as forking and waiting for threads. In Section 5 we provided a syntax-free characterization of terms in this theory (Theorem 5.4) based on an extension of labelled posets, which are well-established in concurrency theory. We then showed in Section 6 that our theory is in a certain sense complete with respect to equality of ordinary labelled posets.

In Section 2 we introduced a simple concurrent programming language and its operational semantics. To model this language denotationally in Section 7, we used our algebraic theory of dynamic threads and the connection between algebraic theories and monadic semantics. We proved that the denotational semantics is adequate, sound and fully abstract at first order.

In summary, our simple language demonstrates that the theory of algebraic effects applies directly to concurrency primitives, and that it is profitable to pursue this algebraic perspective.

ACKNOWLEDGMENTS

This work was funded in part by a Royal Society University Research Fellowship. Three of the authors have received funding from the UK Advanced Research and Innovation Agency (ARIA) as part of the project Qbs4Safety: Core Representation Underlying Safeguarded AI. Sam Lindley and Cristina Matache were supported by UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (MR/T043830/1 and MR/Z000351/1). For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

REFERENCES

[1] IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®) Base Specifications, Issue 8, 2024. Approved 2024. Published by IEEE and The Open Group.
[2] M. Abadi and G. D. Plotkin. A model of cooperative threads. *Log. Methods Comput. Sci.*, 6(4), 2010.
[3] R. Alur, C. Stanford, and C. Watson. A robust theory of series parallel graphs. *Proceedings of the ACM on Programming Languages*, 7(POPL):Article 37, 2023.
[4] N. Benton, M. Hofmann, and V. Nigam. Effect-dependent transformations for concurrent programs. In *PPDP*. ACM, 2016.
[5] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.
[6] F. Bonchi, P. Sobociński, and F. Zanasi. The calculus of signal flow diagrams I: Linear relations on streams. *Inform. Comput.*, pages 2–29, 2017.
[7] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2), 1996.
[8] P. E. de Vilhena and F. Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.
[9] Y. Dvir, O. Kammar, and O. Lahav. A denotational approach to release/acquire concurrency. In S. Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2024.
[10] Y. Dvir, O. Kammar, and O. Lahav. A denotational approach to release/acquire concurrency. In *ESOP, ETAPS*, volume 14577 of *LNCS*. Springer, 2024.
[11] Y. Dvir, O. Kammar, O. Lahav, and G. D. Plotkin. Two-sorted algebraic decompositions of brookes's shared-state denotational semantics. 2025.
[12] U. Fahrenberg, C. Johansen, G. Struth, and K. Ziemiański. Posets with interfaces as a model for concurrency. *Inform. Comput.*, 285, 2022.
[13] J. L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61, 1988.
[14] T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra and its foundations. *J. Logic Alg. Program.*, 80(6), 2011.
[15] T. Hoare and S. van Staden. The laws of programming unify process calculi. *Sci. Comput. Program.*, 85:102–114, 2014.
[16] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1):70–99, July 2006.
[17] R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In L. Birkedal, editor, *FOSSACS, ETAPS*, volume 7213 of *LNCS*, pages 180–194. Springer, 2012.
[18] F. R. John C. Baez, Brandon Coya. Props in network theory. *Theory and Applications of Categories*, 33:727–783, 2018.
[19] R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.
[20] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013.
[21] D. König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 8:114–134, 1926.
[22] P. B. Levy, J. Power, and H. Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, 2003.
[23] M. Markl. Operads and props. volume 5 of *Handbook of Algebra*, pages 87–140. North-Holland, 2008.
[24] T. McKee. Series-parallel graphs: A logical approach. *Journal of Graph Theory*, 7(2):229–236, 1983.
[25] R. Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.
[26] E. Moggi. Notions of computation and monads. *Information and Computation*, 1991.
[27] M. Mukund and M. Nielsen. Ccs, locations and asynchronous transition systems. In *Proc. FSTTCS*, 1992.
[28] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoret. Comput. Sci.*, 13, 1981.
[29] F. J. Oles. Type algebras, functor categories and block structure. In *Algebraic methods in semantics*. 1983.

[30] L. Phipps-Costin, A. Rossberg, A. Guha, D. Leijen, D. Hillerström, K. C. Sivaramakrishnan, M. Pretnar, and S. Lindley. Continuing webassembly with effect handlers. *Proc. ACM Program. Lang.*, 7(OOPSLA2):460–485, 2023.
[31] A. M. Pitts. *Nominal Sets:Names and Symmetry in Computer Science.* CUP, 2013.
[32] G. Plotkin. Concurrency and the algebraic theory of effects. In *Proc. CONCUR 2012*, 2012.
[33] G. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.
[34] G. Plotkin and V. Pratt. Teams can see pomsets. In *OMIV '96: Proceedings of the DIMACS workshop on Partial order methods in verification*, 1996.
[35] G. D. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference*, FOSSACS 2002, pages 342–356. Springer, 2002.
[36] G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9(4), 2013.
[37] V. R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.
[38] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.
[39] K. C. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. Retrofitting effect handlers onto ocaml. In S. N. Freund and E. Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 206–221. ACM, 2021.
[40] I. Stark. Free-algebra models for the pi -calculus. *Theor. Comput. Sci.*, 390(2-3):248–270, 2008.
[41] S. Staton. An algebraic presentation of predicate logic - (extended abstract). In *FOSSACS 2013*, 2013.
[42] S. Staton. Instances of computational effects: An algebraic perspective. In *LICS 2013*, 2013.
[43] S. Staton. Algebraic effects, linearity, and quantum programming languages. In *POPL 2015*, 2015.
[44] W. W. Tait. Intensional interpretations of functionals of finite type i. *Journal of Symbolic Logic*, 32(2):198–212, 1967.
[45] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In *POPL*. ACM, 2011.
[46] R. J. van Glabbeek and G. D. Plotkin. On CSP and the algebraic theory of effects. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C. A. R. Hoare*, pages 333–369. Springer, 2010.
[47] Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects Comput.*, 9(2), 1997.