

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

A model of cooperative threads

Citation for published version:
Abadi, M & Plotkin, GD 2009, A model of cooperative threads. in *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009.* pp. 29-40. https://doi.org/10.1145/1480881.1480887

Digital Object Identifier (DOI):
10.1145/1480881.1480887

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.



Download date: 29. Mar. 2024

A MODEL OF COOPERATIVE THREADS

MARTÍN ABADI AND GORDON D. PLOTKIN

Microsoft Research, Silicon Valley; University of California, Santa Cruz
e-mail address: abadi@microsoft.com

Microsoft Research, Silicon Valley; LFCS, University of Edinburgh
e-mail address: gdp@inf.ed.ac.uk

Abstract. We develop a model of concurrent imperative programming with threads. We focus on a small imperative language with cooperative threads which execute without interruption until they terminate or explicitly yield control. We define and study a trace-based denotational semantics for this language; this semantics is fully abstract but mathematically elementary. We also give an equational theory for the computational effects that underlie the language, including thread spawning. We then analyze threads in terms of the free algebra monad for this theory.

1. Introduction

In the realm of sequential programming, semantics, whether operational or denotational, provides a rich understanding of programming constructs and languages, and serves a broad range of purposes. These include, for instance, the study of verification techniques and the reconciliation of effects with functional programming via monads. With notorious difficulties, these two styles of semantics have been explored for concurrent programming, and, by now, a substantial body of work provides various semantic accounts of concurrency. Typically, that work develops semantics for languages with parallel-composition constructs and various communication mechanisms.

Surprisingly, however, that work provides only a limited understanding of threads. It includes several operational semantics of languages with threads, sometimes with operational notions of equivalence, e.g., [BMT92, PR97, Jef97, JR05]; denotational semantics of those languages seem to be much rarer, and to address message passing rather than shared-memory concurrency, e.g., [FH99, Jef95]. Yet threads are in widespread use, often in the context of elaborate shared-memory systems and languages for which a clear semantics would be beneficial.

In this paper, we investigate a model of concurrent imperative programming with threads. We focus on *cooperative* threads which execute, without interruption, until they

1998 ACM Subject Classification: D.1.3; Programming Techniques, Concurrent Programming (Parallel programming); F.3.2: Logics and Meanings of Programs, Semantics of Programming Languages.

Key words and phrases: denotational semantics, monad, operational semantics, transaction.

A conference version of this paper has appeared as [AP09].

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???
© Martín Abadi and Gordon D. Plotkin
Creative Commons

either terminate or else explicitly yield control. Non-cooperative threads, that is, threads with preemptive scheduling, can be seen as threads that yield control at every step. In this sense, they are a special case of the cooperative threads that we study.

Cooperative threads appear in several systems, programming models, and languages. Often without much linguistic support, they have a long history in operating systems and databases, e.g., [SQL07]. Cooperative threads also arise in other contexts, such as Internet services and synchronous programming [AHT02, BCZ03, Bou06, Bou07, AZ06]. Most recently, cooperative threads are central in two models for programming with transactions, Automatic Mutual Exclusion (AME) and Transactions with Isolation and Cooperation (TIC) [IB07, SKB07]. AME is one of the main starting points for our research. The intended implementations of AME rely on software transactional memory [ST95] for executing multiple cooperative threads simultaneously. However, concurrent transactions do not appear in the high-level operational semantics of the AME constructs [ABH08]. Thus, cooperative threads and their semantics are of interest independently of the details of possible transactional implementations.

We define and study three semantics for an imperative language with primitives for spawning threads, yielding control, and blocking execution.

- We obtain an operational semantics by a straightforward adaptation of previous work. In this semantics, we describe the meaning of a whole program in terms of small-step transitions between states in which spawned threads are kept in a thread pool. This semantics serves as a reference point.
- We also define a more challenging compositional denotational semantics. The meaning of a command is a prefix-closed set of traces. Prefix-closure arises because we are primarily interested in safety properties, that is, in "may" semantics. Each trace is roughly a sequence of transitions, where each transition is a pair of stores, and a store is a mapping from variables to values. We establish adequacy and full-abstraction theorems with respect to the operational semantics. These results require several non-trivial choices in the definition of the denotational semantics.
- Finally, we define a semantics based on the algebraic theory of effects. More precisely, we give an equational theory for the computational effects that underlie the language, and analyze threads in terms of the free algebra monad for this theory. This definition is more principled and systematic; it explains threads with standard semantic structures, in the context of functional programming. As we show, furthermore, we obtain our denotational semantics as a special case.

Section 2 introduces our language and Section 3 defines its operational semantics. Section 4 develops its denotational semantics. Section 5 presents our adequacy and full-abstraction theorems (Theorems 5.10 and 5.15). Section 6 concerns the algebraic theory of effects and the analysis of the denotational semantics in this monadic setting (Theorem 6.4). Section 7 concludes.

2. The Language

Our language is an extension of a basic imperative language with assignments, sequencing, conditionals, and while loops (IMP [Win93]). Programs are written in terms of a finite set of variables Vars, whose values are natural numbers. In addition to those standard constructs, our language includes:

$$
\begin{array}{rcll}
b & \in & \text{BExp} & = \quad \ldots \\
e & \in & \text{NExp} & = \quad \ldots \\
C, D & \in & \text{Com} & = \quad \texttt{skip} \\
 & & & \mid \quad x := e \qquad (x \in \text{Vars}) \\
 & & & \mid \quad C; D \\
 & & & \mid \quad \texttt{if } b \texttt{ then } C \texttt{ else } D \\
 & & & \mid \quad \texttt{while } b \texttt{ do } C \\
 & & & \mid \quad \texttt{async } C \\
 & & & \mid \quad \texttt{yield} \\
 & & & \mid \quad \texttt{block}
\end{array}
$$

Figure 1: Syntax.

- A construct for executing a command in an asynchronous thread. Informally, `async` C forks off the execution of C. This execution is asynchronous, and will not happen if the present thread keeps running without ever yielding control, or if the present thread blocks without first yielding control.
- A construct for yielding control. Informally, `yield` indicates that any pending thread may execute next, as may the current thread.
- A construct for blocking. Informally, `block` halts the execution of the entire program, even if there are pending threads that could otherwise make progress.

We define the syntax of the language in Figure 1. We do not detail the constructs on numerical and boolean expressions, which are as usual.

Figure 2 gives an illustrative example. It shows a piece of code that spawns the asynchronous execution of $x := 0$, then executes $x := 1$ and yields, then resumes but blocks unless the predicate $x = 0$ holds, then executes $x := 2$. The execution of $x := 0$ may hap-

```
async x := 0;
x := 1;
yield;
if x = 0 then skip else block;
x := 2
```

Figure 2: Example command.

pen once the `yield` statement is reached. With respect to safety properties, the conditional blocking amounts to waiting for $x = 0$ to hold. More generally, AME's `blockUntil` b can be written `if` b `then skip else block`.

More elaborate uses of blocking are possible too, and supported by lower-level semantics and actual transactional implementations [IB07, ABH08]. In those implementations, blocking may cause a roll-back and a later retry at an appropriate time. We regard roll-back as an interesting aspect of some possible implementations, but not as part of the high-level semantics of our language, which is the subject of this work.

$$\begin{array}{rcrcl} \Gamma & \in & \text{State} & = & \text{Store} \times \text{ComSeq} \times \text{Com} \\ \sigma & \in & \text{Store} & = & \text{Vars} \to \text{Value} \\ n & \in & \text{Value} & = & \mathbb{N} \\ T & \in & \text{ComSeq} & = & \text{Com}^* \end{array}$$

Figure 3: State space.

Thus, our language is basically a fragment of the AME calculus [ABH08]. It omits higher-order functions and references. It also omits "unprotected sections" for non-cooperative code, particularly legacy code. Non-cooperative code can however be modeled as code with pervasive calls to `yield` (at least with respect to the simple, strong memory models that we use throughout this paper; cf. [GMP06]). See Section 7 for further discussion of possible extensions to our language.

3. Operational Semantics

We give an operational semantics for our language. Despite some subtleties, this semantics is not meant to be challenging. It is given in terms of small-step transitions between states. Accordingly, we define states, evaluation contexts, and the transition relation.

3.1. **States.** As described in Figure 3, a *state* $\Gamma = \langle \sigma, T, C \rangle$ consists of the following components:

- a *store* σ which is a mapping of the given finite set Vars of variables to a set Value of values, which we take to be the set of natural numbers;
- a finite sequence of commands T which we call the *thread pool*;
- a distinguished *active* command C.

We write $\sigma[x \mapsto n]$ for the store that agrees with σ except at x, which is mapped to n. We write $\sigma(b)$ for the boolean denoted by b in σ, and $\sigma(e)$ for the natural number denoted by e in σ, similarly. We write $T.T'$ for the concatenation of two thread pools T and T'.

3.2. **Evaluation Contexts.** As usual, a context is an expression with a hole [ ], and an evaluation context is a context of a particular kind. Given a context $\mathcal{C}$ and an expression C, we write $\mathcal{C}[C]$ for the result of placing C in the hole in $\mathcal{C}$. We use the evaluation contexts defined by the grammar:

$$\mathcal{E} \quad = \quad [\,] \mid \mathcal{E}; C$$

$$
\begin{array}{lcll}
\langle \sigma, T, \mathcal{E}[x := e] \rangle & \longrightarrow & \langle \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}] \rangle & (\text{if } \ \sigma(e) = n) \\
\langle \sigma, T, \mathcal{E}[\texttt{skip}; C] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[C] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[C] \rangle & (\text{if } \ \sigma(b) = \texttt{true}) \\
\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[D] \rangle & (\text{if } \ \sigma(b) = \texttt{false}) \\
\langle \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C] \rangle & \longrightarrow & \langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } (C; \texttt{while } b \texttt{ do } C) \texttt{ else skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{async } C] \rangle & \longrightarrow & \langle \sigma, T.C, \mathcal{E}[\texttt{skip}] \rangle & \\
\langle \sigma, T, \mathcal{E}[\texttt{yield}] \rangle & \longrightarrow & \langle \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip} \rangle & \\
\langle \sigma, T.C.T', \texttt{skip} \rangle & \longrightarrow & \langle \sigma, T.T', C \rangle &
\end{array}
$$

Figure 4: Transition rules of the abstract machine.

3.3. **Steps.** A transition $\Gamma \longrightarrow \Gamma'$ takes an execution from one state to the next. Figure 4 gives rules that specify the transition relation. According to these rules, when the active command is `skip`, a command from the pool becomes the active command. It is then evaluated as such until it produces `skip`, yields, or blocks. No other computation is interleaved with this evaluation. Each evaluation step produces a new state, determined by decomposing the active command into an evaluation context and a subexpression that describes a computation step (for instance, a yield or a conditional).

In all cases at most one rule applies. In two cases, no rule applies. The first is when the active command is `skip` and the pool is empty; this situation corresponds to *normal* termination. The second is when the active command is *blocked*, in the sense that it has the form $\mathcal{E}[\texttt{block}]$; this situation is an *abnormal* termination.

We write $\Gamma \longrightarrow_c \Gamma'$ when $\Gamma \longrightarrow \Gamma'$ via the last rule, and call this a *choice* transition. We write $\Gamma \longrightarrow_a \Gamma'$ when $\Gamma \longrightarrow \Gamma'$ via the other rules, and call this an *active* transition. Active transitions are deterministic, i.e., if $\Gamma \longrightarrow_a \Gamma'$ and $\Gamma \longrightarrow_a \Gamma''$ then $\Gamma' = \Gamma''$.

4. Denotational Semantics

Next we give a compositional denotational semantics for the same language. Here, the meaning of a command is a prefix-closed set of traces, where each trace is roughly a sequence of transitions, and each transition is a pair of stores.

The use of sequences of transitions goes back at least to Abrahamson's work [Abr79] and appears in various studies of parallel composition [AP93, HdeBR94, Bro96, Bro02]. However, the treatment of threads requires some new non-trivial choices. For instance, transition sequences, as we define them, include markers to indicate not only normal termination but also the return of the main thread of control. Moreover, although these markers are similar, they are attached to traces in different ways, one inside pairs of stores, the other not. Such details are crucial for adequacy and full abstraction.

Also crucial to full abstraction is minimizing the information that the semantics records. More explicit semantics will typically be more transparent, for instance, in detailing that a particular step in a computation causes the spawning of a thread, but will consequently fail to be fully abstract.

Section 4.1 is an informal introduction to some of the details of the semantics. Section 4.2 defines transition sequences and establishes some notation. Sections 4.3 and 4.4 define the interpretations of commands and thread pools, respectively. Section 4.5 discusses semantic equivalences.

4.1. **Informal Introduction.** As indicated above, the meaning of a command will be a prefix-closed set of traces, where each trace is roughly a sequence of transitions, and each transition is a pair of stores. Safety properties—which pertain to what "may" happen—are closed under prefixing, hence the prefix-closure condition. Intuitively, when the meaning of a command includes a trace $(\sigma_1, \sigma_1')(\sigma_2, \sigma_2')\ldots$, we intend that the command may start executing with store σ_1, transform it to σ_1', yield, then resume with store σ_2, transform it to σ_2', yield again, and so on.

In particular, the meaning of `block` will consist of the empty sequence ε. The meaning of `yield`; `block` will consist of the empty sequence ε plus every sequence of the form (σ, σ), where σ is any store. Here, the pair (σ, σ) is a "stutter" that represents immediate yielding.

If the meaning of a command C includes $(\sigma_1, \sigma_1')\ldots(\sigma_n, \sigma_n')$ and the meaning of a command D includes $(\sigma_n', \sigma_n'')\ldots(\sigma_m, \sigma_m')$, one might naively expect that the meaning of $C; D$ would contain $(\sigma_1, \sigma_1')\ldots(\sigma_n, \sigma_n'')\ldots(\sigma_m, \sigma_m')$, which is obtained by concatenation plus a simple local composition between (σ_n, σ_n') and (σ_n', σ_n''). Unfortunately, this naive expectation is incorrect. In a trace $(\sigma_1, \sigma_1')(\sigma_2, \sigma_2')\ldots$, some of the pairs may represent steps taken by commands to be executed asynchronously. Those steps need not take place before any further command D starts to execute.

Accordingly, computing the meaning of $C; D$ requires shuffling suffixes of traces in C with traces in D. The shuffling represents the interleaving of C's asynchronous work with D's work. We introduce a special return marker "Ret" in order to indicate how the traces in C should be parsed for this composition. In particular, when C is of the form C_1; `async` (C_2), any occurrence of "Ret" in the meaning of C_2 will not appear in the meaning of C. The application of `async` erases any occurrence of "Ret" from the meaning of C_2—intuitively, because C_2 does not return control to its sequential context.

For example, the meaning of the command

$$x := n; \texttt{yield}; x := n'$$

will contain the trace

$$(\sigma, \sigma[x \mapsto n])(\sigma', \sigma'[x \mapsto n'] \text{ Ret})$$

for every σ and σ'. On the other hand, the meaning of the command

$$x := n; \texttt{async } (x := n'); \texttt{yield}$$

will contain the trace

$$(\sigma, \sigma[x \mapsto n] \text{ Ret})(\sigma', \sigma'[x \mapsto n'])$$

for every σ and σ'. The different positions of the marker Ret correspond to different junction points for any commands to be executed next.

If the meaning of C contains $u(\sigma_n, \sigma_n' \text{ Ret})u'$ and the meaning of D contains $(\sigma_n', \sigma_n'')v$, then the meaning of $C; D$ contains $u(\sigma_n, \sigma_n'')w$, where w is a shuffle of u' and v. Notice that

the marker from $u(\sigma_n, \sigma'_n \text{ Ret})u'$ disappears in this combination. The marker in $u(\sigma_n, \sigma''_n)w$, if present, comes from $(\sigma'_n, \sigma''_n)v$. An analogous combination applies when the meaning of C contains $u(\sigma_n, \sigma'_n \text{ Ret})u'$ and the meaning of D contains $(\sigma'_n, \sigma''_n \text{ Ret})v$ (a trace that starts with a transition with a marker). Moreover, if the meaning of C contains a trace without any occurrence of the marker Ret, then this trace is also in the meaning of $C; D$: the absence of a marker makes it impossible to combine this trace with traces from D.

An additional marker, "Done", ends traces that represent complete normally terminating executions. Thus, the meaning of `skip` will consist of the empty sequence ε and every sequence of the form $(\sigma, \sigma \text{ Ret})$ plus every sequence of the form $(\sigma, \sigma \text{ Ret})$Done. Contrast this with the meaning of `yield`; `block` given above.

It is possible for a trace to contain a Ret marker but not a Done marker. Thus, the meaning of `async` (`block`) will contain the empty sequence ε plus every sequence of the form $(\sigma, \sigma \text{ Ret})$, but not $(\sigma, \sigma \text{ Ret})$Done.

More elaborately, the meaning of the code of Figure 2 will contain all traces of the form

$$(\sigma, \sigma[1])(\sigma[1], \sigma[0])(\sigma[0], \sigma[2] \text{ Ret})\text{Done}$$

where we write $\sigma[n]$ as an abbreviation for $\sigma[x \mapsto n]$. These traces model normal termination after taking the `true` branch of the conditional `if` $x = 0$ `then` $x := 2$ `else block`. The meaning will also contain all prefixes of those traces, which model partial executions—including those that take the `false` branch of the conditional and terminate abnormally.

The two markers are somewhat similar. However, note that $(\sigma, \sigma' \text{ Ret})$ is a prefix of $(\sigma, \sigma' \text{ Ret})$Done, but (σ, σ') is not a prefix of $(\sigma, \sigma' \text{ Ret})$. Such differences are essential.

4.2. **Transitions and Transition Sequences.** A *plain transition* is a pair of stores (σ, σ'). A *return transition* is a pair of stores $(\sigma, \sigma' \text{ Ret})$ in which the second is adorned with the marker Ret. A *transition* is a plain transition or a return transition.

A *main-thread transition sequence* (hereunder simply: *transition sequence*) is a finite (possibly empty) sequence, beginning with a sequence of transitions, of which at most one (not necessarily the last) is a return transition, and optionally followed by the marker Done if one of the transitions is a return transition. We write TSeq for the set of transition sequences.

A *pure transition sequence* is a finite sequence of plain transitions, possibly followed by a marker Done. Note that such a sequence need not be a transition sequence. It is *proper* if it is not equal to Done. We write PSeq for the set of pure transition sequences, and PPSeq for the subset of the proper ones.

We use the following notation:

- We typically let u, v, and w range over transition sequences or pure transition sequences, and let t range over non-empty ones.
- We write $u \leq_p v$ for the prefix relation between sequences u and v (for both kinds of sequences, pure or not). For example, as mentioned above, we have that $(\sigma, \sigma' \text{ Ret}) \leq_p (\sigma, \sigma' \text{ Ret})$Done, but $(\sigma, \sigma') \not\leq_p (\sigma, \sigma' \text{ Ret})$.
- A set P is *prefix-closed* if whenever $u \leq_p v \in P$ then $u \in P$. We write $P{\downarrow}$ for the least prefix-closed set that contains P.
- For a non-empty sequence of transitions t, we write $\text{fst}(t)$ for the first store of the first transition of t.
- For a transition sequence u, we write u^c for the pure transition sequence obtained by *cleaning* u, which means removing the Ret marker, if present, from u.

- We let τ range over stores and stores with return markers.

4.3. **Interpretation of Commands.**

Preliminaries. We let Proc be the collection of the non-empty prefix-closed sets of transition sequences, and let Pool be the collection of the non-empty prefix-closed sets of pure transition sequences. Under the subset partial ordering, Proc and Pool are both ω-cpos (i.e., partial orders with sups of increasing sequences) with least element $\{\varepsilon\}$. We interpret commands as elements of Proc. We use Pool as an auxiliary ω-cpo; below it also serves for the semantics of thread pools. We also let AProc be the sub-ω-cpo of Pool of all non-empty prefix-closed sets of proper pure transition sequences. We think of such sets as modeling asynchronous threads, spawned by an active thread; the difference from Pool is that the latter also contains an element that models the empty thread pool.

We define a continuous cleaning function

$$-^c : \mathrm{Proc} \to \mathrm{AProc}$$

by:

$$P^c = \{u^c \mid u \in P\}$$

(Continuous functions are those preserving all sups of increasing sequences.)

We define the set $u \bowtie v$ of *shuffles* of a pure transition sequence u with a sequence v, whether a transition sequence or a pure transition sequence, as follows:

- If neither finishes with Done, their set of shuffles is defined as usual for finite sequences.
- If u does not finish with Done, then a shuffle of u and v Done is a shuffle of u and v. Similarly, if v does not finish with Done, then a shuffle of u Done and v is a shuffle of u and v.
- A shuffle of u Done and v Done is a shuffle of u and v followed by Done.

If both u and v are pure transition sequences then so is every element of $u \bowtie v$; if u is a pure transition sequence and v is a transition sequence, then every element of $u \bowtie v$ is a transition sequence.

Lemma 4.1. For any u,v, and w where either:

- all three are pure transition sequences, or
- u and v are pure transition sequences, and w is a transition sequence

we have:

$$\bigcup\{v' \bowtie w \mid v' \in u \bowtie v\} = \bigcup\{u \bowtie v' \mid v' \in v \bowtie w\}$$

□

$$
\begin{array}{rcl}
[\![\texttt{skip}]\!] & = & * \\
[\![x := e]\!] & = & \{(\sigma, \sigma[x \mapsto n] \text{ Ret})\text{Done} \mid \sigma \in \text{Store}, \sigma(e) = n\}\downarrow \\
[\![C; D]\!] & = & [\![C]\!] \circ [\![D]\!] \\
[\![\texttt{if } b \texttt{ then } C \texttt{ else } D]\!] & = & \{t \mid t \in [\![C]\!], \text{non-empty}, \text{fst}(t)(b) = \texttt{true}\}\downarrow \\
 & & \cup\{t \mid t \in [\![D]\!], \text{non-empty}, \text{fst}(t)(b) = \texttt{false}\}\downarrow \\
[\![\texttt{while } b \texttt{ do } C]\!] & = & \cup_i [\![(\texttt{while } b \texttt{ do } C)_i]\!] \\
[\![\texttt{async } C]\!] & = & \text{async}([\![C]\!]^c) \\
[\![\texttt{yield}]\!] & = & \text{d}(*) \\
[\![\texttt{block}]\!] & = & \{\varepsilon\}
\end{array}
$$

Figure 5: Denotational semantics.

We define a continuous *composition* function

$$\circ : \text{Proc}^2 \to \text{Proc}$$

by:

$$
\begin{array}{rcl}
P \circ Q & = & \{u(\sigma, \tau)v \mid \exists \sigma', w, w'.\ u(\sigma, \sigma' \text{ Ret})w \in P, \\
 & & \quad (\sigma', \tau)w' \in Q, v \in w \bowtie w'\} \\
 & & \cup\ \{u \mid u \in P \text{ with no return transition}\}
\end{array}
$$

Composition is associative with two-sided unit, given by:

$$* \ = \ \{(\sigma, \sigma \text{ Ret})\text{Done} \mid \sigma \in \text{Store}\}\downarrow$$

We also define a continuous *delay* function

$$\text{d} : \text{Proc} \to \text{Proc}$$

by:

$$\text{d}(P) \ = \ \{(\sigma, \sigma)u \mid \sigma \in \text{Store}, u \in P\}\downarrow$$

Thus, $\text{d}(P)$ is P preceded by all possible stutters (plus ε). Similarly, we define a continuous function

$$\text{async} : \text{AProc} \to \text{Proc}$$

by:

$$\text{async}(Q) \ = \ \{(\sigma, \sigma \text{ Ret})u \mid \sigma \in \text{Store}, u \in Q\}\downarrow$$

Thus, for $P \in \text{Proc}$, $\text{async}(P^c)$ differs from $\text{d}(P)$ only in the placement of the marker Ret.

4.3.1. *Interpretation.* The denotational semantics

$$[\![\cdot]\!] : \text{Com} \longrightarrow \text{Proc}$$

maps a command to a non-empty prefix-closed set of transition sequences. We define it in Figure 5. There, the interpretation of loops relies on the following approximations:

$$
\begin{array}{rcl}
(\texttt{while } b \texttt{ do } C)_0 & = & \texttt{block} \\
(\texttt{while } b \texttt{ do } C)_{i+1} & = & \texttt{if } b \texttt{ then } (C; (\texttt{while } b \texttt{ do } C)_i) \texttt{ else skip}
\end{array}
$$

The 0-th approximant corresponds to divergence, which here we identify with blocking.

We straightforwardly extend the semantics to contexts, so that

$$[\![\mathcal{C}]\!] : \text{Proc} \to \text{Proc}$$

is a continuous function on Proc. This function is defined by induction on the form of $\mathcal{C}$, with the usual clauses of the definition of $[\![\cdot]\!]$ plus $[\![[\ ]]\!](P) = P$.

Proposition 4.2. $[\![\mathcal{C}[C]]\!] = [\![\mathcal{C}]\!]([\![C]\!])$. Therefore, if $[\![C]\!] \subseteq [\![D]\!]$ then $[\![\mathcal{C}[C]]\!] \subseteq [\![\mathcal{C}[D]]\!]$. □

4.4. **Interpretation of Thread Pools.** As an auxiliary definition, it is important to have also an interpretation of thread pools as elements of Pool. We develop one in this section.

4.4.1. *Preliminaries.* We define a continuous shuffle operation

$$\bowtie : (\mathrm{Pool})^2 \to \mathrm{Pool}$$

at this level by:

$$P \bowtie Q = \bigcup_{u \in P, v \in Q} u \bowtie v$$

The shuffle operation is commutative and associative, with unit $I =_{\mathrm{def}} \{\varepsilon, \mathrm{Done}\}$; associativity follows from Lemma 4.1.

We define the set of right shuffles $u \rhd v$ of a pure transition sequence u with a transition sequence v by setting

$$u \rhd (\sigma, \tau)v = \{(\sigma, \tau)w \mid w \in u \bowtie v\}$$

and

$$u \rhd \varepsilon = \{\varepsilon\}$$

We then define

$$\mathrm{async} : \mathrm{Pool} \times \mathrm{Proc} \longrightarrow \mathrm{Proc}$$

by:

$$\mathrm{async}(P, Q) = \bigcup_{u \in P, v \in Q} u \rhd v$$

The use of the notation async for both a unary and a binary operation is a slight abuse, though in line with the algebraic theory of effects: see the discussion in Section 6. In this regard note the equality $\mathrm{async}(P) \circ Q = \mathrm{async}(P, Q)$ (and the equality $[\![\texttt{yield}]\!] \circ P = \mathrm{d}(P)$ points to the corresponding relationship between d and $[\![\texttt{yield}]\!]$).

4.4.2. *Interpretation.* We define the semantics of thread pools by:

$$[\![C_1, \ldots, C_n]\!] = [\![C_1]\!]^c \bowtie \ldots \bowtie [\![C_n]\!]^c \quad (n \geq 0)$$

intending that $[\![\varepsilon]\!] = I$. For any thread pool T, $\mathrm{Done} \in [\![T]\!]$ iff $T = \varepsilon$ (because, for all C, $\mathrm{Done} \notin [\![C]\!]^c$ and, for all P and Q, $I \subseteq P \bowtie Q$ iff $I \subseteq P$ and $I \subseteq Q$). Further, we set $[\![T, C]\!] = \mathrm{async}([\![T]\!], [\![C]\!])$.

Lemma 4.3. For all $P, Q \in \mathrm{Pool}$ and $R \in \mathrm{Proc}$ we have:

(1) $\mathrm{async}(P \bowtie Q, R) = \mathrm{async}(P, \mathrm{async}(Q, R))$
(2) $\mathrm{async}(I, R) = R$

Proof. For the first part, one shows for all pure transition sequences u and v and transition sequences w that:

$$\bigcup \{v' \rhd w \mid v' \in u \bowtie v\} = \bigcup \{u \rhd v' \mid v' \in v \rhd w\}$$

To this end, one proceeds by cases on w, using Lemma 4.1. The second part is obvious. □

4.5. **Equivalences.** An attractive application of denotational semantics is in proving equivalences and implementation relations between commands. Such denotational proofs tend to be simple calculations. Via adequacy and full-abstraction results (of the kind established in Section 5), one then obtains operational results that would typically be much harder to obtain directly by operational arguments.

As an example, we note that we have the following equivalence:

$$[\![\texttt{async}\ (C; \texttt{yield}; D)]\!] = [\![(\texttt{async}\ (C; \texttt{async}\ (D))]\!]$$

This equivalence follows from three facts:

- We have:
$$\begin{array}{rcl} [\![\texttt{yield}; D]\!]^c & = & [\![\texttt{async}\ (D)]\!]^c \\ & = & \{(\sigma, \sigma)u^c \mid \sigma \in \mathrm{Store}, u \in [\![D]\!]\}\downarrow; \end{array}$$
- whenever $[\![D_1]\!]^c = [\![D_2]\!]^c$, $[\![C; D_1]\!]^c = [\![C; D_2]\!]^c$;
- whenever $[\![D_1]\!]^c = [\![D_2]\!]^c$, $[\![\texttt{async}\ (D_1)]\!] = [\![\texttt{async}\ (D_2)]\!]$.

This particular equivalence is interesting for two reasons:

- It models an implementation strategy (in use in AME) where, when executing $C; \texttt{yield}; D$, the `yield` causes a new asynchronous thread for D to be added to the thread pool.
- It illustrates one possible, significant pitfall in more explicit semantics. As discussed above, such a semantics might detail that a particular step in a computation causes the spawning of a thread. More specifically, it might extend transitions with an extra trace component: a triple (σ, u, τ) might represent a step from σ to τ that spawns a thread that contains the trace u. With such a semantics, the meanings of $\texttt{async}\ (C; \texttt{yield}; D)$ and $\texttt{async}\ (C; \texttt{async}\ (D))$ would be different, since they have different spawning behavior.

Many other useful equivalences hold. For instance, we have:

$$[\![x := n; x := n']\!] = [\![x := n']\!]$$

trivially. For every C, we also have:

$$[\![\texttt{async}\ (C); x := n]\!] = [\![x := n; \texttt{async}\ (C)]\!]$$

and, for every C and D, we have:

$$[\![\texttt{async}\ (C); \texttt{async}\ (D)]\!] = [\![\texttt{async}\ (D); \texttt{async}\ (C)]\!]$$

Another important equivalence is:

$$[\![\texttt{while}\ (0 = 0)\ \texttt{do skip}]\!] = [\![\texttt{block}]\!]$$

Thus, the semantics does not distinguish an infinite loop which never yields from immediate blocking. On the other hand, we have:

$$[\![\texttt{while}\ (0 = 0)\ \texttt{do yield}]\!] \neq [\![\texttt{block}]\!]$$

The command $\texttt{while}\ (0 = 0)\ \texttt{do yield}$ generates unbounded sequences of stutters (σ, σ). Similarly, we have:

$$[\![\texttt{yield}; \texttt{yield}]\!] \neq [\![\texttt{yield}]\!]$$

Alternative semantics that would distinguish $\texttt{while}\ (0 = 0)\ \texttt{do skip}$ from `block` or that would identify $\texttt{while}\ (0 = 0)\ \texttt{do yield}$ with `block` and $\texttt{yield}; \texttt{yield}$ with `yield` are viable, however. We briefly discuss those variants and others in Section 7.

We leave as subjects for further research the problems of axiomatizing and of deciding equivalence and implementation relations, and the related problem of program verification, perhaps restricted to subsets of the language—even, for example, to the subset with just composition, spawning, and yielding. There is a large literature on axiomatization and decidability in concurrency theory; see, e.g., [AI07] for discussion and further references. Also, recent results on the automatic verification of asynchronous programs appear rather encouraging [JM07, GMR09]; some of their ideas might be applicable in our setting.

4.6. **Two Extensions.** Trace-based semantics can also be given for variants and enhancements of our basic imperative language. Here we illustrate this point by considering two such enhancements, which illustrate the use of Ret and Done. Section 7 briefly considers other possible language features.

4.6.1. `finish`. While cleaning maps a transition sequence sequence to a proper pure transition sequence, a *marking* function maps a proper pure transition sequence to a transition sequence. For a proper pure transition sequence u, we define u^m by:

$$\begin{array}{lcll} v(\sigma,\sigma')\mathrm{Done}^m & = & v(\sigma,\sigma'\ \mathrm{Ret})\mathrm{Done} & \\ v^m & = & v & (\text{if } v \text{ does not contain Done}) \end{array}$$

Thus, u^m includes a marker Ret only if u contains a marker Done (that is, if u corresponds to a terminating execution); the marker Ret is on the last transition of u^m, intuitively indicating that control is returned to the sequential context when execution terminates.

Much as for cleaning, we extend marking to non-empty prefix-closed sets of proper pure transition sequences:

$$-^m : \mathrm{AProc} \to \mathrm{Proc}$$

Using this extension, we can define the meaning of a construct `finish`, inspired by that of the X10 language [CGA05, SJ05]. We set:

$$[\![\texttt{finish}\ C]\!] = ([\![C]\!]^c)^m$$

The intent is that `finish` C executes C and returns control when all activities spawned by C terminate. For instance, in `finish` (`async` $(x := 0)); x := 1$, the assignment $x := 1$ will execute only after $x := 0$ is done. In contrast, in `async` $(x := 0); x := 1$, the assignments have the opposite ordering. However, `finish` (`async` $(x := 0)$) is not equivalent to $x := 0$, but rather to `yield`$; x := 0$. Beyond this simple example, `finish` can be applied to more complex commands, possibly with nested forks, and ensures that all the activities forked terminate before returning control.

4.6.2. *Parallel Composition.* The definition of parallel composition relies on familiar themes: the use of shuffling, and the decomposition of parallel composition into two cases. The cases correspond to whether the left or the right argument of parallel composition takes the first step.

We define parallel composition at the level of transition sequences by letting $u \mid\mid u'$ and $u \mid\mid_l u'$ be the least sets that satisfy prefix-closure and the following clauses:

- $w \in (\varepsilon \mid\mid w)$ and $w \in (w \mid\mid \varepsilon)$,
- $(t \mid\mid_l t') \cup (t' \mid\mid_l t) \subseteq (t \mid\mid t')$,
- if $v \in (w \mid\mid t')$, then $(\sigma,\sigma')v \in (\sigma,\sigma')w \mid\mid_l t'$,
- if $v \in w \bowtie w'$ then $(\sigma,\tau)v \in (\sigma,\sigma'\ \mathrm{Ret})w \mid\mid_l (\sigma',\tau)w'$.

Extending this function to

$$- \,||\, - : \mathrm{Proc} \times \mathrm{Proc} \to \mathrm{Proc}$$

we can define the meaning of a parallel-composition construct:

$$[\![C \,||\, D]\!] = [\![C]\!] \,||\, [\![D]\!]$$

The reader may verify that parallel composition, as defined here, has the expected properties, for instance that it is commutative and associative with unit `skip`. It is also worth noting that (under mild assumptions on the available expressions) the binary nondeterministic choice operator $\cup$ considered in Section 6.1 is definable from parallel composition. The converse also holds, under restricted circumstances: if all occurrences of `yield` in C and D occur inside an `async` then we have:

$$[\![C \,||\, D]\!] = [\![C; D]\!] \cup [\![D; C]\!]$$

5. Adequacy and Full Abstraction

In this section we establish that the denotational semantics of Section 4 coincides with the operational semantics of Section 3, and is fully abstract.

The adequacy theorem (Theorem 5.10), which expresses the coincidence, says that the traces that the denotational semantics predicts are exactly those that can happen operationally. These traces may in general represent the behavior of a command in a context. As a special case, the adequacy theorem applies to runs, which are essentially traces that the command can produce on its own, i.e., with an empty context. This special case is spelled out in Corollary 5.11 which states that the runs that the denotational semantics predicts are exactly those that can happen operationally

The full-abstraction theorem (Theorem 5.15) states that two commands C and D have the same set of traces denotationally if, and only if, they produce the same runs in combination with every context. In particular, observing runs, we cannot distinguish C and D in any context. Note that, given Corollary 5.11, we may equivalently speak of runs denotationally or operationally. We comment on other possible notions of observation, and the corresponding full-abstraction results, below.

Section 5.1 defines runs precisely. Sections 5.2 and 5.3 present our adequacy and full-abstraction results, respectively.

5.1. **Runs.** A pure transition sequence *generates a run* if, however it can be written as $u(\sigma, \sigma')(\sigma'', \sigma''')v$, we have $\sigma' = \sigma''$. If $w = (\sigma_1, \sigma_2) \dots (\sigma_{n-1}, \sigma_n)$ is such a pure transition sequence, we set $\mathrm{run}(w) = \sigma_1 \dots \sigma_n$ and $\mathrm{run}(w\,\mathrm{Done}) = \sigma_1 \dots \sigma_n\mathrm{Done}$. A transition sequence u *generates a run* if u^c does, and then we set $\mathrm{run}(u) = \mathrm{run}(u^c)$.

If a pure transition sequence u generates a run, then it can be easily be recovered from $\mathrm{run}(u)$: the run $\sigma_1 \dots \sigma_n$ maps back to

$$(\sigma_1, \sigma_2) \dots (\sigma_{n-1}, \sigma_n)$$

and the run $\sigma_1 \dots \sigma_n\mathrm{Done}$ maps back to

$$(\sigma_1, \sigma_2) \dots (\sigma_{n-1}, \sigma_n)\mathrm{Done}$$

Since each non-empty run contains at least two elements, this definition applies when $n = 0$ and $n \geq 2$. We write runs(P) for the set of runs generated by (pure) transition sequences in P.

5.2. **Adequacy.**

Lemma 5.1. The following equalities hold:

(1) $[\![\mathcal{E}[\texttt{block}]]\!] = [\![\texttt{block}]\!]$
(2) $[\![\texttt{skip}; C]\!] = [\![C]\!]$
(3) $[\![\mathcal{E}[\texttt{async } D]]\!] = \mathrm{async}([\![D]\!]^c, [\![\mathcal{E}[\texttt{skip}]]\!])$
(4) $[\![\mathcal{E}[\texttt{yield}]]\!]^c = \mathrm{async}([\![\mathcal{E}[\texttt{skip}]]\!]^c, [\![\texttt{skip}]\!])^c$
(5) For all $T \neq \varepsilon$ (equivalently Done $\notin [\![T]\!]$),

$$[\![T]\!] = \bigcup\{[\![T'.T'', C]\!]^c \mid T = T'.C.T''\}$$

Proof. The first part is immediate from the semantics of `block` and the definition of composition. The second part holds as $*$ is a unit for composition. The third part follows from the facts that $\mathrm{async}(P) \circ Q = \mathrm{async}(P, Q)$ and that composition is associative with unit $*$.

For the fourth part, using the third part one sees that it is enough to show that for every $\mathcal{E}$ we have:

$$[\![\mathcal{E}[\texttt{yield}]]\!]^c = [\![\texttt{async } \mathcal{E}[\texttt{skip}]]\!]^c$$

As composition is associative with unit $*$, this is equivalent to showing that, for every C we have:

$$[\![\texttt{yield}; C]\!]^c = \mathrm{async}([\![C]\!]^c)^c$$

which follows immediately, expanding the definitions. The proof of the fifth part is a straightforward verification. □

Lemma 5.2. If C is blocked then, for all T, $[\![T, C]\!] = \{\varepsilon\}$.

Proof. We calculate:

$$\begin{array}{rcll} [\![T, \mathcal{E}[\texttt{block}]]\!] & = & \mathrm{async}([\![T]\!], [\![\mathcal{E}[\texttt{block}]]\!]) & \\ & = & \mathrm{async}([\![T]\!], [\![\texttt{block}]\!]) & \text{(by Lemma 5.1)} \\ & = & \{\varepsilon\} & \end{array}$$

□

Lemma 5.3. $[\![T, \texttt{skip}]\!] = \{(\sigma, \sigma \text{ Ret})v \mid v \in [\![T]\!]\}\downarrow$.

Proof. Immediate from the definition of async. □

The next lemma applies when C is neither `skip` nor blocked.

Lemma 5.4. Suppose that $\langle \sigma, T, C\rangle \longrightarrow_a \langle \sigma', T', C'\rangle$. Then, for any σ'', $(\sigma, \sigma'')v \in [\![T, C]\!]^c$ iff $(\sigma', \sigma'')v \in [\![T', C']\!]^c$.

Proof. We divide into cases according to the form of C. In the case where C has the form $\mathcal{E}[\texttt{skip}; D]$ we have $\sigma' = \sigma$, $T' = T$ and $C' = \mathcal{E}[D]$. So, by Lemma 5.1, we have $[\![T', C']\!] = [\![T, C]\!]$, and we are done.

In the case where C instead has the form $\mathcal{E}[\texttt{async}\ D]$, we have $\sigma' = \sigma$, $T' = T.D$ and $C' = \mathcal{E}[\texttt{skip}]$ and we calculate:

$$\begin{array}{rcll} [\![T', C']\!] & = & [\![T.D, \mathcal{E}[\texttt{skip}]]\!] & \\ & = & \mathrm{async}([\![T]\!], \mathrm{async}([\![D]\!]^c, [\![\mathcal{E}[\texttt{skip}]]\!])) & \\ & = & [\![T, \mathcal{E}[\texttt{async}\ D]]\!] & \text{(by Lemma 5.1)} \\ & = & [\![T, C]\!] & \end{array}$$

and we are done.

In the case where C instead has the form $\mathcal{E}[\texttt{yield}]$, we have $\sigma' = \sigma$, $T' = T.\mathcal{E}[\texttt{skip}]$, $C' = \texttt{skip}$ and, again using Lemma 5.1, we calculate:

$$\begin{array}{rcl} [\![T', C']\!]^c & = & [\![T.\mathcal{E}[\texttt{skip}], \texttt{skip}]\!]^c \\ & = & \mathrm{async}([\![T]\!], \mathrm{async}([\![\mathcal{E}[\texttt{skip}]]\!]^c, [\![\texttt{skip}]\!]))^c \\ & = & [\![T, \mathcal{E}[\texttt{yield}]]\!]^c \\ & = & [\![T, C]\!]^c \end{array}$$

and we are done.

In the next case, C has the form $\mathcal{E}[x := e]$, and we have $\sigma' = \sigma[x \mapsto \sigma(e)]$, $T' = T$ and $C' = \mathcal{E}[\texttt{skip}]$. Here $[\![T, C]\!] = [\![T, x := e; \mathcal{E}[\texttt{skip}]]\!]$. So we have that: $(\sigma, \tau)v \in [\![T, C]\!]$ holds iff $(\sigma', \tau) \in [\![T, \mathcal{E}[\texttt{skip}]]\!]$

Otherwise, C has one of the forms $\mathcal{E}[\texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D]$ or $\mathcal{E}[\texttt{while}\ b\ \texttt{do}\ C]$ and we proceed much as in the previous case. □

Lemma 5.5. Suppose that $\langle \sigma, T, C\rangle \longrightarrow_a{}^*$ some $\langle \sigma', T', \texttt{skip}\rangle$ with $u \in [\![T']\!]^c$. Then $(\sigma, \sigma')u \in [\![T, C]\!]^c$.

Proof. This follows from Lemmas 5.3 and 5.4. □

For the proof of the converse of this lemma, we proceed by an induction on the size of loop-free commands. We then extend to general commands by expressing their semantics in terms of the semantics of their approximations by loop-free commands. The *size* of a loop-free command is defined by structural recursion:

$$|\texttt{skip}| = |\texttt{block}| = 1 \quad |x := e| = |\texttt{async}\ C| = |\texttt{yield}| = 2$$
$$|\texttt{if}\ b\ \texttt{then}\ C\ \texttt{else}\ D| = |C; D| = |C| + |D|$$

Note that if $\langle \sigma, T, C\rangle \longrightarrow_a \langle \sigma', T', C'\rangle$ and C is loop-free, then so is C' and, further, $|C'| < |C|$.

The *approximation* relation $C \preceq D$ between loop-free commands C and general commands D is defined to be the least such relation closed under all non-looping program constructs and such that, for any b, C, D, and $i \geq 0$:

$$\texttt{block} \preceq D \qquad \frac{C \preceq D}{(\texttt{while}\ b\ \texttt{do}\ C)_i \preceq (\texttt{while}\ b\ \texttt{do}\ D)}$$

This relation is extended to thread pools and contexts in the obvious way: we write $T \preceq T'$ and $\mathcal{C} \preceq \mathcal{C}'$ for these extensions.

Lemma 5.6. Suppose that $T \preceq U$, $C \preceq D$, and, further, that $\langle \sigma, T, C\rangle \longrightarrow_a \langle \sigma', T', C'\rangle$. Then, for some U', D' with $T' \preceq U'$ and $C' \preceq D'$, $\langle \sigma, U, D\rangle \longrightarrow_a{}^* \langle \sigma', U', D'\rangle$.

Proof. One first notes that, for any C, D, if $\mathcal{E}[C] \preceq D$ then D has the form $\mathcal{E}'[D']$ where $\mathcal{E} \preceq \mathcal{E}'$ and $C \preceq D'$. The proof then divides into cases according to the rule used to show that $\langle \sigma, T, C\rangle \longrightarrow_a \langle \sigma', T', C'\rangle$.

For example, suppose we have $C = \mathcal{E}[\texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2]$ and $\sigma(b) = \texttt{true}$. We know that D must have the form $\mathcal{E}'[D']$ where $\mathcal{E} \preceq \mathcal{E}'$ and $(\texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2) \preceq D'$. Suppose now that D' has the form $\texttt{while } b \texttt{ do } D''$. Then we must have, for some $i \geq 0$ that $C_1 = C''; (\texttt{while } b \texttt{ do } C'')_i$ where $C'' \preceq D''$. But then we observe that

$$\langle \sigma, U, D\rangle \longrightarrow_a \langle \sigma, U, \mathcal{E}'[\texttt{if } b \texttt{ then } D''; D' \texttt{ else skip}]\rangle \longrightarrow_a \langle \sigma, U, \mathcal{E}'[D''; D]\rangle$$

and the conclusion follows. The other cases are straightforward. □

Next we define the *approximants* $C^{(i)}$ of a command C by induction on i and structural recursion on C, beginning with the case where C has one of the forms `skip`, `block`, $x := e$, or `yield`, when $C^{(i)} = C$, and continuing with:

$$\begin{aligned}(\texttt{async } C)^{(i)} &= \texttt{async } C^{(i)}\\ (\texttt{if } b \texttt{ then } C \texttt{ else } D)^{(i)} &= \texttt{if } b \texttt{ then } C^{(i)} \texttt{ else } D^{(i)}\\ (C; D)^{(i)} &= C^{(i)}; D^{(i)}\\ (\texttt{while } b \texttt{ do } C)^{(i)} &= (\texttt{while } b \texttt{ do } C^{(i)})_i\end{aligned}$$

For any C one shows that $C^{(i)} \preceq C^{(i+1)} \preceq C$.

Lemma 5.7.

(1) If $C \preceq D$ then $[\![C]\!] \subseteq [\![D]\!]$.
(2) For any command D:

$$[\![D]\!] = \bigcup_i [\![D^{(i)}]\!]$$

Proof. The first part is evident using the monotonicity of the semantics of the program constructors and the semantic of loops. For the second part, we proceed by structural induction on D. All cases are straightforward, using the continuity of the program constructors, except for loops where we calculate:

$$\begin{aligned}[\![\texttt{while } b \texttt{ do } D]\!] &= \textstyle\bigcup_i [\![(\texttt{while } b \texttt{ do } D)_i]\!]\\ &= \textstyle\bigcup_i [\![(\texttt{while } b \texttt{ do } [\,])_i]\!]([\![D]\!])\\ &= \textstyle\bigcup_i [\![(\texttt{while } b \texttt{ do } [\,])_i]\!](\bigcup_i [\![D^{(i)}]\!])\\ &= \textstyle\bigcup_i [\![(\texttt{while } b \texttt{ do } D^{(i)})_i]\!]\\ &= \textstyle\bigcup_i [\![(\texttt{while } b \texttt{ do } D)^{(i)}]\!]\end{aligned}$$

□

We can now establish the converse of Lemma 5.5.

Lemma 5.8. Suppose that $(\sigma, \sigma')u \in [\![T, C]\!]^c$. Then $\langle \sigma, T, C\rangle \longrightarrow_a{}^* \langle \sigma', T', \texttt{skip}\rangle$ for some T' with $u \in [\![T']\!]^c$.

Proof. We begin by proving this for loop-free commands C. The proof is by induction on the size of C. If C is `skip` we have $\langle \sigma, T, \texttt{skip}\rangle \longrightarrow_a{}^* \langle \sigma, T, \texttt{skip}\rangle$ and the conclusion follows, as, by Lemma 5.3, $(\sigma, \sigma')u \in [\![T, \texttt{skip}]\!]^c$ iff $\sigma' = \sigma$ and $u \in [\![T]\!]^c$. If C is blocked, the conclusion holds trivially, by Lemma 5.2.

If C is neither `skip` nor blocked we have $\langle \sigma, T, C\rangle \longrightarrow_a \langle \sigma'', T'', C''\rangle$ (and then C'' is loop-free and $|C''| < |C|$). Then, by Lemma 5.4, $(\sigma, \sigma')u \in [\![T, C]\!]^c$ iff $(\sigma'', \sigma')u \in [\![T'', C'']\!]^c$ which latter, by the induction hypothesis, implies $\langle \sigma'', T'', C''\rangle \longrightarrow_a{}^*$ some $\langle \sigma', T', \texttt{skip}\rangle$ with $u \in [\![T']\!]^c$ which, in turn, implies $\langle \sigma, T, C\rangle \longrightarrow_a{}^*$ some $\langle \sigma', T', \texttt{skip}\rangle$ with $u \in [\![T']\!]^c$, as desired.

Next suppose that $(\sigma,\sigma')u \in [\![T,D]\!]^c$, where now D is not loop-free. By Lemma 5.7 $(\sigma,\sigma')u \in [\![T,C]\!]^c$ for some $C \preceq D$. So, by the above, $\langle \sigma,T,C\rangle \longrightarrow_a{}^*$ some $\langle \sigma',T',\mathtt{skip}\rangle$ with $u \in [\![T']\!]^c$. The desired conclusion follows immediately, using Lemma 5.6. □

Lemma 5.9.

(1) For any proper non-empty pure transition sequence u, $(\sigma,\sigma')u \in [\![T,C]\!]^c$ iff for some T',C', $\langle \sigma,T,C\rangle \longrightarrow_a{}^* \longrightarrow_c \langle \sigma',T',C'\rangle$ with $u \in [\![T',C']\!]^c$.
(2) For any σ, σ', T, C, $(\sigma,\sigma')\mathrm{Done} \in [\![T,C]\!]^c$ iff $\langle \sigma,T,C\rangle \longrightarrow_a{}^* \langle \sigma',\varepsilon,\mathtt{skip}\rangle$.

Proof. By Lemma 5.8, $(\sigma,\sigma')u \in [\![T,C]\!]^c$ holds iff $\langle \sigma,T,C\rangle \longrightarrow_a{}^*$ some $\langle \sigma',T',\mathtt{skip}\rangle$ does, with $u \in [\![T']\!]$. In the case where u is proper the conclusion follows from Lemma 5.1. In the case where u is Done we see from the definition of $[\![T']\!]$ that $\mathrm{Done} \in [\![T']\!]$ iff $T' = \varepsilon$. □

The following *Adequacy Theorem* for pure transition sequences is an immediate consequence of Lemmas 5.8 and 5.9:

Theorem 5.10.

(1) For $n > 0$, $(\sigma_1,\sigma_1')\dots(\sigma_n,\sigma_n') \in [\![T,C]\!]^c$ iff there are T_i, C_i, $(i = 1,n)$ such that $T_1 = T$, $C_1 = C$, and $\langle \sigma_i,T_i,C_i\rangle \longrightarrow_a{}^* \longrightarrow_c \langle \sigma_i',T_{i+1},C_{i+1}\rangle$, for $1 \le i \le n-1$, and $\langle \sigma_n,T_n,C_n\rangle \longrightarrow_a{}^*$ some $\langle \sigma_n',T',\mathtt{skip}\rangle$.
(2) For $n > 0$, $(\sigma_1,\sigma_1')\dots(\sigma_n,\sigma_n')\mathrm{Done} \in [\![T,C]\!]^c$ iff there are T_i, C_i, $(i = 1,n)$ such that $T_1 = T$, $C_1 = C$, and $\langle \sigma_i,T_i,C_i\rangle \longrightarrow_a{}^* \longrightarrow_c \langle \sigma_i',T_{i+1},C_{i+1}\rangle$, for $1 \le i \le n-1$, and $\langle \sigma_n,T_n,C_n\rangle \longrightarrow_a{}^* \langle \sigma_n',\varepsilon,\mathtt{skip}\rangle$.

□

As a corollary we obtain an adequacy theorem for runs:

Corollary 5.11.

(1) For $n \ge 2$, $\sigma_1\dots\sigma_n \in \mathrm{runs}([\![T,C]\!])$ iff there are T_i, C_i, $(i = 1,n-1)$ such that $T_1 = T$, $C_1 = C$, $\langle \sigma_i,T_i,C_i\rangle \longrightarrow_a{}^* \longrightarrow_c \langle \sigma_{i+1},T_{i+1},C_{i+1}\rangle$ $(1 \le i \le n-2)$, and $\langle \sigma_{n-1},T_{n-1},C_{n-1}\rangle \longrightarrow_a{}^*$ some $\langle \sigma_n,T',\mathtt{skip}\rangle$.
(2) For $n \ge 2$, $\sigma_1\dots\sigma_n\mathrm{Done} \in \mathrm{runs}([\![T,C]\!])$ iff there are T_i, C_i, $(i = 1,n-1)$ such that $T_1 = T$, $C_1 = C$, and $\langle \sigma_i,T_i,C_i\rangle \longrightarrow_a{}^* \longrightarrow_c \langle \sigma_{i+1}T_{i+1},C_{i+1}\rangle$ $(1 \le i \le n-2)$, and $\langle \sigma_{n-1},T_{n-1},C_{n-1}\rangle \longrightarrow_a{}^* \langle \sigma_n,\varepsilon,\mathtt{skip}\rangle$.

□

5.3. **Full Abstraction.** The first lemma in the proof of full abstraction bounds the nondeterminism of commands in semantic terms.

Lemma 5.12. For all C, u, and σ, the set $\{\tau \mid u(\sigma,\tau) \in [\![C]\!]\}$ is finite.

Proof. More generally, we prove that for all T, C, $u = (\sigma_1,\tau_1)\dots(\sigma_{n-1},\tau_{n-1})$, and σ_n, the set $\{\tau \mid u(\sigma_n,\tau) \in [\![T,C]\!]\}$ is finite, and similarly that the set $\{\tau \mid u(\sigma_n,\tau) \in [\![T]\!]\}$ is finite. The proof is by induction on n. The proof relies on adequacy; a purely semantic proof might be possible but seems harder.

- If C is `skip`, then Lemma 5.3 implies that τ_1 is σ_1 Ret, and $(\sigma_2,\tau_2)\dots(\sigma_n,\tau) \in [\![T_1]\!]$. In case $n = 1$, we are done, with a unique choice for τ_1. Otherwise, we conclude by induction hypothesis.
- if C is blocked, then $n = 0$, by Lemma 5.2, so this case is vacuous.

- If C is neither `skip` nor blocked, then Lemma 5.8 implies that τ_1 is unique. In case $n = 1$, we are done, with a unique choice for τ_1. Otherwise, Lemma 5.8 also implies that $(\sigma_2, \tau_2) \dots (\sigma_n, \tau) \in [\![T']\!]$ for a unique T'. As in the case of `skip`, the desired conclusion follows by induction hypothesis.
- Finally, having established the claim for sequences of length n for sets of the form $[\![T, C]\!]$, we consider sequences of length n in a set of the form $[\![T]\!]$. Suppose that T consists of $C_1, \dots, C_k$. A transition sequence v in $[\![T]\!]$ is a shuffle of transition sequences in $[\![C_1]\!], \dots, [\![C_k]\!]$, each of length at most n. The finiteness property for $[\![T]\!]$ follows from the fact that there are only finitely many possible ways of decomposing v as a shuffle.

□

Intuitively, Lemma 5.12 is useful because it implies that, at any point, there are certain steps that a command cannot take, and in proofs those steps can be used as unambiguous, visible markers of activity by the context. This lemma is somewhat fragile—it does not hold once one adds to the language either the nondeterministic choice operator considered in Section 6.1 or the parallel composition operator of Section 4.6.2. It follows that neither of these operators is definable in the language. An alternative argument that does not use the lemma relies on fresh variables instead. The fresh variables permit an alternative definition of the desired markers.

Full-abstraction results invariably require some notion of observation. Let us write $\mathrm{obs}(P)$ for the observations that we make on $P \in \mathrm{Proc}$. Equational full abstraction is that $[\![C]\!] = [\![D]\!]$ if and only if, for every context $\mathcal{C}$, we have $\mathrm{obs}([\![\mathcal{C}[C]]\!]) = \mathrm{obs}([\![\mathcal{C}[D]]\!])$. In other words, two commands have the same meaning if and only if they yield the same observations in every context of the language. The stronger inequational full abstraction is that $[\![C]\!] \subseteq [\![D]\!]$ if and only if, for every context $\mathcal{C}$, we have $\mathrm{obs}([\![\mathcal{C}[C]]\!]) \subseteq \mathrm{obs}([\![\mathcal{C}[D]]\!])$. The difficult part of this equivalence is usually the implication from right to left: that if, for every context $\mathcal{C}$, $\mathrm{obs}([\![\mathcal{C}[C]]\!]) \subseteq \mathrm{obs}([\![\mathcal{C}[D]]\!])$, then $[\![C]\!] \subseteq [\![D]\!]$.

One possible candidate for $\mathrm{obs}(P)$ is P^c. This notion of observation can be criticized as too fine-grained. Nevertheless, we find it useful to prove full abstraction for this notion of observation, with the following lemma. We first need some auxiliary definitions for its proof, and the lemma that follows. Given two stores σ and σ', we define:

- a boolean expression $\mathrm{check}(\sigma)$ as the conjunction of the formulas $x = n$ for every variable x, where n is the natural number $\sigma(x)$ (so $\mathrm{check}(\sigma)$ is true in σ and false elsewhere);
- a command $\mathrm{goto}(\sigma)$ as the sequence of assignments $x := n$ for every variable x, where n is the natural number $\sigma(x)$;
- a command $(\sigma \rightsquigarrow \sigma')$ as `if` $\mathrm{check}(\sigma)$ `then` $\mathrm{goto}(\sigma')$ `else block`;
- a command $(\sigma \rightsquigarrow \sigma' \rightsquigarrow \sigma'')$ as $(\sigma \rightsquigarrow \sigma')$; `yield`; $(\sigma' \rightsquigarrow \sigma'')$; `yield`.

These definitions exploit the fact that the set of variables is finite. However, with more care, analogous definitions could be given otherwise, by focusing on the set of variables relevant to the programs under observation.

Lemma 5.13. If $[\![\mathcal{C}[C]]\!]^c \subseteq [\![\mathcal{C}[D]]\!]^c$ for every context $\mathcal{C}$, then $[\![C]\!] \subseteq [\![D]\!]$.

Proof. Letting $P = [\![C]\!]$ and $Q = [\![D]\!]$, we assume that $P \not\subseteq Q$ and prove that there exists $\mathcal{C}$ such that $[\![\mathcal{C}]\!](P)^c \not\subseteq [\![\mathcal{C}]\!](Q)^c$. For this, choose a sequence w in P but not in Q. If $w = w^c$, then we can take $\mathcal{C}$ to be $[\ ]$. Therefore, for the rest of the proof, we consider the case $w \neq w^c$.

If $w \neq w^c$, then w is of the form $u(\sigma, \sigma' \text{ Ret})v$. We let $\mathcal{C} = [\,]; (\sigma' \rightsquigarrow \sigma'')$ where σ'' does not appear in u or v and $u(\sigma, \sigma'') \notin Q$ (so, by prefix-closure, $u(\sigma, \sigma'')v \notin Q$). Such a choice of σ'' is always possible by Lemma 5.12. Thus, $[\![\mathcal{C}]\!](P)$ contains $u(\sigma, \sigma'' \text{ Ret})v$, and $[\![\mathcal{C}]\!](P)^c$ contains $u(\sigma, \sigma'')v$.

Suppose that $u(\sigma, \sigma'')v$ is also in $[\![\mathcal{C}]\!](Q)^c$, and that this is because some sequence w' is in $[\![\mathcal{C}]\!](Q)$ and $w'^c = u(\sigma, \sigma'')v$. By the definition of the semantics of sequential composition, this could arise in one of the following ways:

- $w' = u(\sigma, \sigma'' \text{ Ret})v$, with $w \in Q$. This contradicts $w \notin Q$.
- $w' = u'(\sigma, \sigma'')v'$, and σ'' occurs as the second store of a return transition in either u' or v'. This contradicts the requirement that σ'' does not appear in u or v.
- $w' = u(\sigma, \sigma'')v$, $w' \in Q$, and w' does not have a return transition. This contradicts the requirement that $u(\sigma, \sigma'') \notin Q$.

□

Another possible candidate for $\text{obs}(P)$ is $\text{runs}(P)$. Runs record more than mere input-output behavior, but much less than entire execution histories. We therefore find them attractive for our purposes. The following lemma connects runs to cleaning.

Lemma 5.14. If $\text{runs}([\![\mathcal{C}[C]]\!]) \subseteq \text{runs}([\![\mathcal{C}[D]]\!])$ for every context $\mathcal{C}$, then $[\![C]\!]^c \subseteq [\![D]\!]^c$.

Proof. Letting $P = [\![C]\!]$ and $Q = [\![D]\!]$, we assume that $P^c \not\subseteq Q^c$ and prove that there exists $\mathcal{C}$ such that $\text{runs}([\![\mathcal{C}]\!](P)) \not\subseteq \text{runs}([\![\mathcal{C}]\!](Q))$.

For this, choose a sequence $w \in P^c$ but $w \notin Q^c$, in order to derive a contradiction.

First, suppose that w is of the form $(\sigma_1, \sigma_1') \ldots (\sigma_n, \sigma_n')$, with $n > 0$. We let $\mathcal{C}$ be `async` $[\,]; \text{mesh}(w)$, where $\text{mesh}(w)$ is the command

$$\texttt{yield}; (\sigma_1' \rightsquigarrow \sigma_1'' \rightsquigarrow \sigma_2); \ldots; (\sigma_{n-1}' \rightsquigarrow \sigma_{n-1}'' \rightsquigarrow \sigma_n); (\sigma_n' \rightsquigarrow \sigma_n'')$$

where the stores σ_i'' are all different from one another and from all other stores in w, and are such that

$$(\sigma_1, \sigma_1') \ldots (\sigma_i, \sigma_i')(\sigma_i', \sigma_i'') \notin Q^c$$

and

$$(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')(\sigma_{i-1}'', \sigma_i)(\sigma_i, \sigma_i')(\sigma_i', \sigma_i'') \notin Q^c$$

Such a choice of stores σ_i'' is always possible by Lemma 5.12. Since $[\![\text{mesh}(w)]\!]$ contains the transition sequence:

$$(\sigma_1, \sigma_1)(\sigma_1', \sigma_1'')(\sigma_1'', \sigma_2) \ldots (\sigma_{n-1}'', \sigma_n)(\sigma_n', \sigma_n'' \text{ Ret})\text{Done}$$

we obtain that $[\![\mathcal{C}]\!](P)$ contains the transition sequence:

$$(\sigma_1, \sigma_1)(\sigma_1, \sigma_1')(\sigma_1', \sigma_1'')(\sigma_1'', \sigma_2)(\sigma_2, \sigma_2') \ldots (\sigma_{n-1}'', \sigma_n)(\sigma_n, \sigma_n')(\sigma_n', \sigma_n'' \text{ Ret})$$

which generates the run:

$$\sigma_1\sigma_1\sigma_1'\sigma_1''\sigma_2\sigma_2' \ldots \sigma_{n-1}''\sigma_n\sigma_n'\sigma_n''$$

Suppose that this run is also in $\text{runs}([\![\mathcal{C}]\!](Q))$. Therefore, there exists $w' \in Q^c$ such that

$$(\sigma_1, \sigma_1')(\sigma_1', \sigma_1'')(\sigma_1'', \sigma_2)(\sigma_2, \sigma_2') \ldots (\sigma_{n-1}'', \sigma_n)(\sigma_n, \sigma_n')(\sigma_n', \sigma_n'')$$

is a shuffle of w' with

$$(\sigma_1', \sigma_1'')(\sigma_1'', \sigma_2) \ldots (\sigma_{n-1}'', \sigma_n)(\sigma_n', \sigma_n'')\text{Done}$$

which we call w'', or with a prefix of w''. We analyze the origin of the transitions in the shuffle:

- The transitions (σ_i, σ'_i) must all come from w', since each of the transitions in w'' contains one of the stores σ''_j and, by choice, these are different from σ_i and σ'_i.
- Suppose that, up to some $i - 1 < n$, w' starts like w, in other words it starts as $(\sigma_1, \sigma'_1) \dots (\sigma_{i-1}, \sigma'_{i-1})$. Suppose further that, in the shuffle up to this point, each transition (σ_j, σ'_j) is followed immediately by the corresponding transitions $(\sigma'_j, \sigma''_j)(\sigma''_j, \sigma_{j+1})$ from w''. We argue that this remains the case up to n.
  - We consider $(\sigma'_{i-1}, \sigma''_{i-1})$, the next possible transition in the shuffle. This transition cannot come from w' because, by the choice of σ''_{i-1}, we have that
    $$(\sigma_1, \sigma'_1) \dots (\sigma_{i-1}, \sigma'_{i-1})(\sigma'_{i-1}, \sigma''_{i-1}) \notin Q^c$$
    So this transition comes from w''.
  - One step further, in order to derive a contradiction, we suppose that the transition $(\sigma''_{i-1}, \sigma_i)$ comes from w'. So w' starts:
    $$(\sigma_1, \sigma'_1) \dots (\sigma_{i-1}, \sigma'_{i-1})(\sigma''_{i-1}, \sigma_i)$$
    and in fact:
    $$(\sigma_1, \sigma'_1) \dots (\sigma_{i-1}, \sigma'_{i-1})(\sigma''_{i-1}, \sigma_i)(\sigma_i, \sigma'_i)$$
    since, as noted above, the last transition here must come from w'. The next transition in the shuffle is (σ'_i, σ''_i). By the choice of σ''_i, we have that
    $$(\sigma_1, \sigma'_1) \dots (\sigma_{i-1}, \sigma'_{i-1})(\sigma''_{i-1}, \sigma_i)(\sigma_i, \sigma'_i)(\sigma'_i, \sigma''_i) \notin Q^c$$
    So the transition (σ'_i, σ''_i) cannot come from w'. Therefore, it must come from w''. However, the next available transition in w'' is $(\sigma''_{i-1}, \sigma_i)$, and (σ'_i, σ''_i) and $(\sigma''_{i-1}, \sigma_i)$ must be different because σ''_{i-1} and σ''_i are different, by choice, from σ'_i and σ_i.
    Thus, the assumption that the transition $(\sigma''_{i-1}, \sigma_i)$ comes from w' leads to a contradiction. This transition must come from w''.
- Finally, suppose that, up to n, w' starts like w, in other words as:
  $$(\sigma_1, \sigma'_1) \dots (\sigma_n, \sigma'_n)$$
  and that, in the shuffle, each transition (σ_j, σ'_j) is followed immediately by the corresponding transitions $(\sigma'_j, \sigma''_j)(\sigma''_j, \sigma_{j+1})$ from w''. By the choice of σ''_n, we have that
  $$(\sigma_1, \sigma'_1) \dots (\sigma_n, \sigma'_n)(\sigma'_n, \sigma''_n) \notin Q^c$$
  so (σ'_n, σ''_n) comes from w'', not from w'.

In sum, $w' = w$, and therefore $w \in Q^c$, contradicting our assumption that $w \notin Q^c$.

Next, suppose that w is of the form $(\sigma_1, \sigma'_1) \dots (\sigma_n, \sigma'_n)$ Done. With the same $\mathcal{C}$, we obtain that $[\![\mathcal{C}]\!](P)$ contains the transition sequence:

$$(\sigma_1, \sigma_1)(\sigma_1, \sigma'_1)(\sigma'_1, \sigma''_1)(\sigma''_1, \sigma_2)(\sigma_2, \sigma'_2) \dots (\sigma''_{n-1}, \sigma_n)(\sigma_n, \sigma'_n)(\sigma'_n, \sigma''_n \text{ Ret})\text{Done}$$

which generates the run:

$$\sigma_1\sigma_1\sigma'_1\sigma''_1\sigma_2\sigma'_2 \dots \sigma''_{n-1}\sigma_n\sigma'_n\sigma''_n\text{Done}$$

Suppose that this run is also in $\text{runs}([\![\mathcal{C}]\!](Q))$. Again, by the choice of $\sigma''_1, \dots, \sigma''_n$, this can be the case only if w is in Q^c. (The argument for the contradiction may actually be simplified in this case, because of the marker Done.) □

We obtain the following *Full-abstraction Theorem*:

Theorem 5.15. $[\![C]\!] \subseteq [\![D]\!]$ iff, for every context $\mathcal{C}$, $\mathrm{runs}([\![\mathcal{C}[C]]\!]) \subseteq \mathrm{runs}([\![\mathcal{C}[D]]\!])$.

Proof. The implication from $[\![C]\!] \subseteq [\![D]\!]$ is an immediate consequence of the compositionality of the semantics (Proposition 4.2). The converse follows from Lemmas 5.13 and 5.14. □

Coarser-grained definitions of $\mathrm{obs}(P)$ may sometimes be appropriate. For those, we expect that full abstraction will typically require additional closure conditions on P, such as closure under suitable forms of stuttering and

mumbling, much as in our work and Brookes's on parallel composition [AP93, Bro96].

6. Algebra

The development of the denotational semantics in Section 4 is ad hoc, in that the semantics is not related to any systematic approach. In this section we show how it fits in with the algebraic theory of effects [PP02, PP03, HPP06, PP08, PP09].

In the functional programming approach to imperative languages, commands have unit type, 1. Then, taking the monadic point of view [BHM02], they are modeled as elements of $T(1)$ for a suitable monad T on, say, the category of ω-cpos and continuous functions. For parallelism one might look for something along the lines of the resumptions monad [HP79, CM93, HPP06].

In the algebraic approach to computational effects [PP02, HPP06], one analyses the monads as free algebra monads for a suitable equational or Lawvere theory L (here meaning in the enriched sense, so that inequations are allowed, as are families of operations continuously parameterized over an ω-cpo). The operations of the theory (for example a binary choice operation in the case of nondeterminism) are thought of as effect constructors in that they create the effects at hand.

As discussed in [HP79], resumptions are generally not fully abstract when their domain equation is solved in a category of cpos. If, instead, it is solved in a category of semilattices, increased abstraction may be obtained. The situation was analyzed from the algebraic point of view in [HPP06]. It was shown there that resumptions arise by combining a theory for stores [PP02] with one for nondeterminism, one for nontermination, and one for a unary operation d thought of as suspending computation. The difference between solving the equation in a category of semilattices or cpos essentially amounts to whether or not one asks that d, and the other operations, commute with nondeterminism.

In [Bro96], Brookes, using an apparently different and mathematically elementary trace-based approach, succeeded in giving a fully abstract semantics for a language of the kind considered in [HP79]. However, in [Jef95], Jeffrey showed that trace-based models of concurrent languages can arise as solutions to domain equations in a category of semilattices, thereby relating the two approaches.

We propose here to identify the suspension operation d with the operation of the same name introduced in Section 4.3; indeed this identification was the origin of the definition of `yield` given there, and it is natural to further identify `yield` as the generic effect [PP03] corresponding to the suspension operation. These identifications are justified by Corollary 6.5, below, and the discussion following it.

In Section 6.1 we carry out an algebraic analysis of resumptions. We show in Theorem 6.1 that, imposing the commutations with nondeterminism just discussed, they do indeed correspond to a traces model, provided one uses the Hoare or lower powerdomain.

(This powerdomain is a natural choice as we consider only "may" semantics in this paper, and elements of such powerdomains are Scott closed, so downwards-closed, a natural generalization of our prefix-closedness condition.) The proof makes the link between domain equations and traces.

The missing ingredient in an algebraic analysis of Proc is then an account of `async`. In the denotational semantics of any command of the form `async` C, all Ret marking is lost from the meaning of C, because of the application of the cleaning function, $-^c$; further all the sequences in $[\![C]\!]^c$ are proper. We propose to treat `async` as a generic effect, parameterized by an element of AProc (which will be $[\![C]\!]^c$).

In order to give the equations for the async operation it will, as one may expect, be useful to first have an algebraic analysis of AProc; we carry out this analysis in Section 6.2. It turns out, as detailed in Theorem 6.2, that AProc is similar to, but not quite, a resumptions ω-cpo. Finally, we analyze processes in Section 6.3, showing, in Theorem 6.4, that a process is a kind of "double-thread"—more precisely, a resumption that returns not only a value but also an element of AProc.

6.1. **Resumptions.** Our theory L_{Res} for resumptions follows [HPP06] but is somewhat modified, as we are interested only in "may" semantics and as we wish to allow infinitely proceeding processes. The theory is a combination of several constituent theories which we now consider successively.

The Lawvere theory L_{S} of stores can be presented via a family of unary operations $\mathrm{update}_{x,n}$ and a family of "$\mathbb{N}$-ary" operations lookup_x ($x \in \mathrm{Vars}$, $n \in \mathbb{N}$). (An $\mathbb{N}$-ary operation is a countably infinitary operation whose arguments are indexed by the natural numbers.) For any computation γ, $\mathrm{update}_{x,n}(\gamma)$ is read as the computation that first updates x to n and then proceeds as γ; for any $\mathbb{N}$-indexed collection $(\gamma_n)_n$ of computations, $\mathrm{lookup}_x((\gamma_n)_n)$ is read as the computation that proceeds as γ_n if x has value n in the current store.

The Lawvere theory L_{H} for nondeterminism is that of the lower (aka Hoare) powerdomain, presented using a binary nondeterministic choice operation $\cup$; the Lawvere theory L_Ω for nontermination is the theory of a least element, presented using a constant Ω; and the Lawvere theory L_{d} for suspension is that of a unary operation d, with no equations. See [PP02, HPP06] for more details of these theories, including an account of the equations for stores and for Hoare powerdomains.

For resumptions, continuing to follow [HPP06], we wish the operations of L_{S} to commute with those of L_{H} and L_Ω (which automatically commute with each other) and it is also natural to have d commute with nondeterministic choice, but not with the operations of L_{S}, as we wish to model interruption points, and not with Ω, as we want to be able to model infinitely proceeding processes. We therefore define:

$$L_{\mathrm{Res}} = L_{\mathrm{H}} \otimes ((L_{\mathrm{S}} \otimes L_\Omega) + L_{\mathrm{d}})$$

and let T_{Res} be the associated monad. (For any two theories L and L' presented using disjoint signatures, the theories $L + L'$ and $L \otimes L'$ can be presented using the union of the signatures of L and L' and, in the former case, by the union of their equations and, in the latter case, by the union of their equations together with additional equations that say that each operation of each theory commutes with each operation of the other.)

We now give an elementary trace-based picture of $T_{\mathrm{Res}}(P)$ for sufficiently general ω-cpos P. Let Q be a partial order. A *Q-transition* is a pair of states $(\sigma, \sigma'\ x)$ in which the

second is marked with an element x of Q; we let τ range over stores and stores marked with an element of Q. A *basic Q-transition sequence* is a non-empty sequence consisting of plain transitions optionally followed by a Q-transition. Let $\leq_Q$ be the least preorder on the set of basic Q-transition sequences which contains the prefix relation $\leq_p$ and is such that, for any x, y in Q, if $x \leq y$ then $u(\sigma, \sigma' x) \leq_Q u(\sigma, \sigma' y)$. One has that $\leq_Q$ is a partial order and that $u \leq_Q v$ holds iff:

$$\begin{array}{ll} \text{either} & u \leq_p v \\ \text{or else} & \exists w, x \leq y.\, u \leq_p w(\sigma, \sigma' x) \wedge v = w(\sigma, \sigma' y) \end{array}$$

We need a few notions concerning ideals in partial orders. An *ideal* I in a partial order Q is a downwards-closed subset of Q; for any subset X of Q we write $X{\downarrow}$ for the least ideal including X, viz $\{x \in Q \mid \exists y \in X.\, x \leq y\}$; and for any $x \in Q$ we write $x{\downarrow}$ for $\{x\}{\downarrow}$. Downwards-closed sets, i.e., ideals, provide a suitable generalization of prefix-closed sets when passing from sequences to general partial orders.

An ideal I is *directed* if it is nonempty and any two elements of the ideal have an upper bound in the ideal. An ideal is *denumerably generated* if $I = X{\downarrow}$ for some denumerable $X \subseteq I$. We write $\mathcal{I}^{\uparrow}_{\omega}(Q)$, respectively $\mathcal{I}_{\omega}(Q)$, for the collection of all denumerably generated directed ideals of Q, respectively all denumerably generated ideals of Q, and we partially order them by subset; $\mathcal{I}^{\uparrow}_{\omega}(Q)$ is an ω-cpo, indeed it is the free such over Q; and $\mathcal{I}^{\omega}(Q)$ is the free ω-cpo with all finite sups over Q: it follows that it is also the free such ω-cpo over $\mathcal{I}^{\uparrow}_{\omega}(Q)$.

Let Q-BTrans be the set of basic Q-transition sequences, partially ordered as above. One can view $\mathcal{I}_{\omega}(Q\text{-BTrans})$ as an L_{Res}-model with the following definitions of the operations, where now we use l to range over Vars:

$$\begin{array}{rcl} (\mathrm{update}_{l,n})_{\mathcal{I}_{\omega}(Q\text{-BTrans})}(I) & = & \{(\sigma,\tau)u \mid (\sigma[l \mapsto n], \tau)u \in I\} \\ (\mathrm{lookup}_{l})_{\mathcal{I}_{\omega}(Q\text{-BTrans})}((I_n)_n) & = & \bigcup_n \{(\sigma,\tau)u \in I_n \mid \sigma(l) = n\} \\ I \cup_{\mathcal{I}_{\omega}(Q\text{-BTrans})} J & = & I \cup J \\ \Omega_{\mathcal{I}_{\omega}(Q\text{-BTrans})} & = & \emptyset \\ \mathrm{d}_{\mathcal{I}_{\omega}(Q\text{-BTrans})}(I) & = & \{(\sigma,\sigma)u \mid \sigma \in \mathrm{Store}, u \in I\} \cup \ \{(\sigma,\sigma) | \sigma \in \mathrm{Store}\} \end{array}$$

(We skip over the small difference between the notion of an L_{Res}-model and of an algebra satisfying equations.)

We write ωCpo and ωSL for, respectively, the category of ω-cpos and the category of ω-cpos with all finite sups. For any poset P, its lifting $P_{\perp}$ is the poset obtained from P by freely adjoining a least element $\perp$; its elements are $(0, x)$, for $x \in P$, and $\perp$, and they are ordered in the evident way. If P has all sups of increasing ω-chains, i.e., is an ω-cpo (respectively has finite sups), so does $P_{\perp}$. For any object a of any given category, and any set X, we write $X \otimes a$ and a^X for, respectively, the X-fold sum and product of a with itself, assuming they exist. The category ωSL has countable biproducts, given by the usual cartesian product of posets, and it is convenient to identify $X \otimes L$ with L^X, for countable sets X.

The next theorem shows that the algebraic notion of resumptions can indeed be characterized in trace-based terms, specifically as ideals of basic Q-transition sequences.

Theorem 6.1. Viewed as an L_{Res}-model, $\mathcal{I}_{\omega}(Q\text{-BTrans})$ is $T_{\mathrm{Res}}(\mathcal{I}^{\uparrow}_{\omega}(Q))$. The unit

$$(\eta_{T_{\mathrm{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)} : \mathcal{I}^{\uparrow}_{\omega}(Q) \to \mathcal{I}_{\omega}(Q\text{-BTrans})$$

is given by:

$$(\eta_{T_{\mathrm{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)}(I) = \{(\sigma, \sigma\ x) \mid \sigma \in \mathrm{Store}, x \in I\}$$

and, for any continuous $f : \mathcal{I}^{\uparrow}_{\omega}(Q) \to \mathcal{I}_{\omega}(R\text{-BTrans})$, its Kleisli extension

$$f^{\dagger} : \mathcal{I}_{\omega}(Q\text{-BTrans}) \to \mathcal{I}_{\omega}(R\text{-BTrans})$$

is given by:

$$\begin{array}{rcl} f^{\dagger}(I) & = & \{u(\sigma,\tau)v \mid \exists \sigma', x.\ u(\sigma,\sigma'\ x) \in I, (\sigma',\tau)v \in f(x{\downarrow})\} \\ & & \cup\ \{u \in I \mid u \text{ has no } Q\text{-transition}\} \end{array}$$

Proof. Models of L_{Res} in ωCpo correspond to models of L_{S} in ωSL together with a morphism $\mathrm{d}' : L_{\perp} \to L$, where L is the carrier of the model. (Such morphisms are equivalent to ω-continuous maps on L which preserve binary sups, but not necessarily $\perp$.) The carrier L of the model of L_{Res} is that of the model of L_{S} in ωSL; it is necessarily an ω-cpo with all finite lubs. The L_S operations on L become those of the model of L_{S} in ωSL, and the map $\mathrm{d} : L \to L$ extends uniquely to a morphism on $L_{\perp}$, obtaining the required map d'. This correspondence extends straightforwardly to an equivalence of categories.

So, as $\mathcal{I}_{\omega}(Q)$ is the free ω-cpo with finite sups over the ω-cpo $\mathcal{I}^{\uparrow}_{\omega}(Q)$, we seek the free structure

$$(L, (\mathrm{update}_{l,n})_L, (\mathrm{lookup}_l)_L, d_L)$$

over $\mathcal{I}_{\omega}(Q)$, consisting of a model $(L, (\mathrm{update}_{l,n})_L, (\mathrm{lookup}_l)_L)$ of L_{S} in ωSL and a morphism $\mathrm{d}' : L_{\perp} \to L$.

By Theorem 1 of [PP02] the free algebra monad for L_S over ωSL is $T_S = (S \otimes -)^S$, where we abbreviate Store to S (the theorem depends on the set of variables being finite). The definitions of the operations $(\mathrm{update}_{l,n})_{T_S(L)}$ and $(\mathrm{lookup}_l)_{T_S(L)}$ of an algebra $T_S(L)$ are given by Proposition 1 of [PP02]; the unit $(\eta_{T_{\mathrm{S}}})_L$ at L is the canonical map $L \longrightarrow (S \otimes L)^S$.

So, by Corollary 2 of [HPP06], for any poset Q, L is the solution of the following "domain equation" in ωSL:

$$L \cong (S \otimes (L_{\perp} + \mathcal{I}_{\omega}(Q)))^S \tag{6.1}$$

by which we mean the initial ω-cpo with finite sups L and map

$$\alpha : (S \otimes (L_{\perp} + \mathcal{I}_{\omega}(Q)))^S \to L$$

(Such a map is necessarily an isomorphism.)

The morphism $(\mathrm{update}_{l,n})_L : L \to L$ is

$$L \xrightarrow{\alpha^{-1}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q)) \xrightarrow{(\mathrm{update}_{l,n})_{T_S(L_{\perp} + \mathcal{I}_{\omega}(Q))}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q)) \xrightarrow{\alpha} L$$

the morphism $(\mathrm{lookup}_l)_L : L^{\mathbb{N}} \to L$ is

$$L^{\mathbb{N}} \xrightarrow{(\alpha^{-1})^{\mathbb{N}}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q))^{\mathbb{N}} \xrightarrow{(\mathrm{lookup}_l)_{T_S(L_{\perp} + \mathcal{I}_{\omega}(Q))}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q)) \xrightarrow{\alpha} L$$

the morphism $\mathrm{d}'_L : L_{\perp} \to L$ is

$$L_{\perp} \xrightarrow{\mathrm{inl}} L_{\perp} + \mathcal{I}_{\omega}(Q) \xrightarrow{(\eta_{T_{\mathrm{S}}})_{(L_{\perp} + \mathcal{I}_{\omega}(Q))}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q)) \xrightarrow{\alpha} L$$

and at $\mathcal{I}^{\uparrow}(Q)$ the unit $\eta_{T_{\mathrm{Res}}}$ is

$$\mathcal{I}^{\uparrow}_{\omega}(Q) \hookrightarrow \mathcal{I}_{\omega}(Q) \xrightarrow{\mathrm{inr}} L_{\perp} + \mathcal{I}_{\omega}(Q) \xrightarrow{(\eta_{T_{\mathrm{S}}})_{(L_{\perp} + \mathcal{I}_{\omega}(Q))}} T_{\mathrm{S}}(L_{\perp} + \mathcal{I}_{\omega}(Q)) \xrightarrow{\alpha} L$$

Now, since countable copowers and powers coincide in ωSL, Equation (6.1) can be rewritten as:

$$L \cong S \otimes (S \otimes (L_\perp + \mathcal{I}_\omega(Q))) \tag{6.2}$$

As $\mathcal{I}_\omega : \mathrm{Pos} \to \omega\mathrm{SL}$ is a left adjoint, where Pos is the category of posets, it preserves all colimits; $\mathcal{I}_\omega$ also commutes with lifting. So there is an isomorphism:

$$\beta : \mathcal{I}_\omega(S \times (S \times (R_\perp + Q))) \cong S \otimes (S \otimes (\mathcal{I}_\omega(R)_\perp + \mathcal{I}_\omega(Q)))$$

for any poset R. So, again using that $\mathcal{I}_\omega$ preserves all colimits, we can solve Equation (6.2) by first solving the equation:

$$R \cong S \times (S \times (R_\perp + Q))$$

in the category Pos, and then applying $\mathcal{I}_\omega$. To do that, one takes R to be the least set such that

$$R = S \times (S \times (R_\perp + Q))$$

and then imposes the evident inductively defined partial order on it. The solution of Equation (6.2) is then given by taking $L = \mathcal{I}_\omega(R)$ and $\alpha = \beta^{-1}$.

We now have an expression of L as $\mathcal{I}_\omega(R)$, as well as definitions of $(\mathrm{update}_{l,n})_L$, $(\mathrm{lookup}_l)_L$, d_L, and the unit. So, given the initial discussion above, we see that L forms the free model of L_{Res} over $\mathcal{I}_\omega^\uparrow(R)$ with unit:

$$(\eta_{\mathrm{Res}})_{\mathcal{I}_\omega^\uparrow(R)}(I) = \{(\sigma, (\sigma, \mathrm{inr}(x))) \mid x \in I\}$$

and with operations:

$$\begin{array}{lcl}
(\mathrm{update}_{l,n})_L(I) & = & \{(\sigma, (\sigma', u)) \mid (\sigma[l \mapsto n], (\sigma', u)) \in I\} \\
(\mathrm{lookup}_l)_L((I_n)_n) & = & \{(\sigma, (\sigma', u)) \in I_n \mid n \in \mathbb{N}, \sigma(l) = n\} \\
I \cup_L J & = & I \cup J \\
\Omega_L & = & \emptyset \\
\mathrm{d}_L(I) & = & \{(\sigma, (\sigma, \mathrm{inl}(0, u))) \mid \sigma \in S, u \in I\} \ \cup \ \{(\sigma, (\sigma, \perp)) \mid \sigma \in S\})
\end{array}$$

There is an evident isomorphism of partial orders $\theta_{\mathrm{Res}} : R \cong Q\text{-BTrans}$, given recursively by:

$$\begin{array}{lcl}
\theta_{\mathrm{Res}}((\sigma, (\sigma', \mathrm{inl}((0, u))))) & = & (\sigma, \sigma')\theta_{\mathrm{Res}}(u) \\
\theta_{\mathrm{Res}}((\sigma, (\sigma', \mathrm{inl}(\perp)))) & = & (\sigma, \sigma') \\
\theta_{\mathrm{Res}}((\sigma, (\sigma', \mathrm{inr}(x)))) & = & (\sigma, \sigma' x)
\end{array}$$

This induces an isomorphism $\mathcal{I}_\omega(R) \cong \mathcal{I}_\omega(Q\text{-BTrans})$ of ω-cpos, and so the free such model is also carried by $\mathcal{I}_\omega(Q\text{-BTrans})$. Using this, and the above definitions of the operations and unit for $\mathcal{I}_\omega(R)$, one then verifies that the operations and unit for $\mathcal{I}_\omega(Q\text{-BTrans})$ are as required.

As regards the formula for the Kleisli extension, that $f^\dagger(\eta_{T_{\mathrm{Res}}})_{\mathcal{I}_\omega^\uparrow(Q)} = f$ is evident and that the purported extension is a morphism of models of L_{Res} is a calculation. □

One can go further and obtain a closely related, if less elementary, picture of $T_{\mathrm{Res}}(P)$ for an arbitrary ω-cpo P: one needs a notion of ideal that takes the ω-sups of P into account.

6.2. **Asynchronous Processes.** One might hope that AProc can be understood as an ω-cpo of resumptions, and, indeed, basic {Done}-transition sequences and proper pure non-empty transition sequences are very similar. Define a map $\theta_{\mathrm{AProc}} : \{\mathrm{Done}\}\text{-BTrans} \to \mathrm{PPSeq}$ by:

$$\begin{array}{lcll} \theta_{\mathrm{AProc}}(u(\sigma,\sigma'\ \mathrm{Done})) & = & u(\sigma,\sigma')\mathrm{Done} & \\ \theta_{\mathrm{AProc}}(u) & = & u & (\text{if } u \text{ does not contain Done}) \end{array}$$

Unfortunately, while θ_{AProc} is a monotonic bijection, it is not an isomorphism of partial orders, as $u(\sigma,\sigma') \leq_p u(\sigma,\sigma')\mathrm{Done}$ but $u(\sigma,\sigma') \not\leq_{\{\mathrm{Done}\}} u(\sigma,\sigma'\ \mathrm{Done})$.

There is a related programming language phenomenon. Denotationally, we have the inclusion:

$$[\![(\texttt{async}\ (\texttt{yield};\texttt{block}));C]\!] \subseteq [\![(\texttt{async}\ \texttt{skip});C]\!]$$

but not the inclusion:

$$[\![\texttt{yield};\texttt{block}]\!] \subseteq [\![\texttt{skip}]\!]$$

As in the proof of the full-abstraction theorem, one can distinguish $[\![\texttt{yield};\texttt{block}]\!]$ from $[\![\texttt{skip}]\!]$ using a sequential context; however, this context is not available when the command is within an `async`.

To solve this difficulty we take the theory of asynchronous threads L_{AProc} to be L_{Res} extended by a new constant halt and the equation:

$$\mathrm{d}(\Omega) \leq \mathrm{halt}$$

We can turn AProc into a model of L_{AProc} by defining operations as follows:

$$\begin{array}{rl} (\mathrm{update}_{l,n})_{\mathrm{AProc}}(P) & = \{(\sigma,\sigma')u \mid (\sigma[l \mapsto n],\sigma')u \in P\} \cup \{\varepsilon\} \\ (\mathrm{lookup}_l)_{\mathrm{AProc}}((P_n)_n) & = \bigcup_n \{(\sigma,\sigma')u \in P_n \mid \sigma(l) = n\} \cup \{\varepsilon\} \\ P \cup_{\mathrm{AProc}} Q & = P \cup Q \\ \Omega_{\mathrm{AProc}} & = \{\varepsilon\} \\ \mathrm{d}_{\mathrm{AProc}}(P) & = \{(\sigma,\sigma)u \mid \sigma \in \mathrm{Store}, u \in P\} \cup \{\varepsilon\} \\ \mathrm{halt}_{\mathrm{AProc}} & = \{(\sigma,\sigma)\mathrm{Done} \mid \sigma \in \mathrm{Store}\}\downarrow \end{array}$$

Note that $\mathrm{halt}_{\mathrm{AProc}} = [\![\texttt{skip}]\!]^c$.

We write T_{AProc} for the monad associated to the theory AProc. The next theorem shows that the variant theory L_{AProc} indeed captures AProc. First we need some notation.

- We define a unary derived operation $a_{l,m,k}$, for $l \in \mathrm{Vars}$ and $m, n \in \mathrm{Value}$ by:

  $$a_{l,m,k}(x) \equiv_{\mathrm{def}} \mathrm{lookup}_l((t_{m'})_{m'})$$

  where:

  $$t_{m'} \equiv_{\mathrm{def}} \begin{cases} \mathrm{update}_{l,k}(x) & (\text{if } m' = m) \\ \Omega & (\text{otherwise}) \end{cases}$$

- We define a unary derived operation $a_{\sigma,\sigma'}$, for $\sigma, \sigma' \in \mathrm{Store}$ by:

  $$a_{\sigma,\sigma'}(x) \equiv_{\mathrm{def}} a_{l_1,\sigma(l_1),\sigma'(l_1)}(\ldots a_{l_n,\sigma(l_n),\sigma'(l_n)}(x)\ldots)$$

  where $l_1, \ldots, l_n$ is an enumeration of Vars.
- For every sequence of plain transitions $u = (\sigma_1,\sigma'_1)\ldots(\sigma_n,\sigma'_n)$ we define a unary derived operation a_u by:

  $$a_u(x) \equiv_{\mathrm{def}} a_{\sigma_1,\sigma'_1}(d(\ldots a_{\sigma_n,\sigma'_n}(d(x))\ldots))$$

- For every sequence of plain transitions u and $\sigma, \sigma' \in \mathrm{Store}$, we define two constants $\overline{u}$ and $\overline{u(\sigma,\sigma')\mathrm{Done}}$ by:

$$\overline{u} \equiv_{\mathrm{def}} a_u(\Omega) \qquad \text{and} \qquad \overline{u(\sigma,\sigma')\mathrm{Done}} \equiv_{\mathrm{def}} a_u(\mathrm{halt})$$

Note that $\overline{u}_{\mathrm{APROC}} = \overline{u}_{\mathcal{I}_\omega(Q)} = u\downarrow$, where, for example, $\overline{u}_{\mathrm{APROC}}$ is the interpretation of $\overline{u}$ in APROC; further $\overline{u(\sigma,\sigma')\mathrm{Done}}_{\mathrm{APROC}} = u(\sigma,\sigma')\mathrm{Done}\downarrow$. Below we may confuse a constant or operation with its interpretation in a specific algebra A, e.g., writing $\overline{u}$ or a_u rather than $\overline{u}_A$ or $(a_u)_A$, provided that the intended algebra can be understood from the context.

Theorem 6.2. APROC is the initial L_{APROC}-model, i.e., it is $T_{\mathrm{APROC}}(0)$.

Proof. We begin by examining the connection between $\mathcal{I}_\omega(\{\mathrm{Done}\})$ and APROC. By Theorem 6.1, $\mathcal{I}_\omega(\{\mathrm{Done}\})$ is the free model of L_{Res} over $\{\mathrm{Done}\}$. So $f : \{\mathrm{Done}\} \to \mathrm{APROC}$ has a unique extension to a morphism $f^\dagger : \mathcal{I}_\omega(\{\mathrm{Done}\}) \to \mathrm{APROC}$ of L_{Res}-models, where $f(\mathrm{Done}) =_{\mathrm{def}} \mathrm{halt}_{\mathrm{APROC}}$. We now show that:

$$f^\dagger(I) = \{\theta_{\mathrm{APROC}}(u) \mid u \in I\}\downarrow$$

from which it follows that $f^\dagger$ is onto. It is enough to show that $f^\dagger(u\downarrow) = \theta_{\mathrm{APROC}}(u)\downarrow$, which holds as, for any u not containing Done, we calculate that

$$f^\dagger(u\downarrow) = f^\dagger(\overline{u})_{\mathcal{I}_\omega(\{\mathrm{Done}\})}) = (\overline{u})_{\mathrm{APROC}} = u\downarrow = \theta_{\mathrm{APROC}}(u)\downarrow$$

and that

$$\begin{array}{rclcl} f^\dagger(u(\sigma,\sigma'\ \mathrm{Done})\downarrow) & = & f^\dagger((a_u)_{\mathcal{I}_\omega(\{\mathrm{Done}\})}(\eta(\mathrm{Done}))) & = & (a_u)_{\mathrm{APROC}}(f^\dagger(\eta(\mathrm{Done}))) \\ & = & (a_u)_{\mathrm{APROC}}(\mathrm{halt}_{\mathrm{APROC}}) & = & u(\sigma,\sigma')\mathrm{Done}\downarrow \\ & = & \theta_{\mathrm{APROC}}(u(\sigma,\sigma'\ \mathrm{Done}))\downarrow & & \end{array}$$

where, in both cases, the second equality holds as $f^\dagger$ is a morphism of L_{Res}-models.

Let L be a model of L_{APROC}. We have to show there is a unique morphism $h : \mathrm{APROC} \to L$. For uniqueness, let h, h' be such morphisms. Then both $f^\dagger \circ h$ and $f^\dagger \circ h'$ are morphisms of L_{Res} models from $\mathcal{I}_\omega(\{\mathrm{Done}\})$ to L, extending the map $\mathrm{Done} \mapsto \mathrm{halt}_L$. So, as there is only one such map, $f^\dagger \circ h = f^\dagger \circ h'$, and therefore, as $f^\dagger$ is onto, $h = h'$, as required.

For existence, define the map $\theta : \mathrm{PPSeq} \to L$ by: $\theta(u) = (a_u)_L$. Using the fact that L is a model of APROC, particularly the axiom $\mathrm{d}(\Omega) \leq \mathrm{halt}$, one has that θ is monotonic. One can then define a continuous map $h : \mathrm{APROC} \to L$ by:

$$h(I) = \bigvee_{u \in I} \theta(u)$$

with the sup on the right existing as I is denumerable. Let g be the unique morphism of L_{Res} models from $\mathcal{I}_\omega(\{\mathrm{Done}\})$ to L, extending the map $\mathrm{Done} \mapsto \mathrm{halt}_L$.

We have that $h \circ f^\dagger = g$, as, for any u not containing Done, we may calculate that:

$$h(f^\dagger(u\downarrow)) = h(u\downarrow) = \theta(u) = a_u = g(a_u) = g(u)$$

and that

$$\begin{array}{rclcl} h(f^\dagger(u(\sigma,\sigma'\ \mathrm{Done})\downarrow)) & = & h(a_u(\mathrm{halt})) & = & h(u(\sigma,\sigma')\mathrm{Done}\downarrow) \\ & = & \theta(u(\sigma,\sigma')\mathrm{Done}) & = & a_{u(\sigma,\sigma')\mathrm{Done}} \\ & = & a_{u(\sigma,\sigma')}(\mathrm{halt}) & = & g(a_{u(\sigma,\sigma')}(\eta(\mathrm{Done}))) \\ & = & g(u(\sigma,\sigma'\,\mathrm{Done})) & & \end{array}$$

As $h \circ f^\dagger = g$, and $f^\dagger$ and g are morphisms of L_{Res} models, and $f^\dagger$ is onto, h is automatically a morphism of L_{Res} models. For example, for the preservation of d, given $I \in \mathrm{AProc}$, choose $J \in \mathcal{I}_\omega(\{\mathrm{Done}\})$ such that $f^\dagger(J) = I$ and calculate that:

$$\begin{array}{rclcl} h(\mathrm{d}_{\mathrm{AProc}}(I)) & = & h(\mathrm{d}_{\mathrm{AProc}}(f^\dagger(J))) & & \\ & = & h(f^\dagger(\mathrm{d}_{\mathcal{I}_\omega(\{\mathrm{Done}\})}(J))) & = & g(\mathrm{d}_{\mathcal{I}_\omega(\{\mathrm{Done}\})}(J)) \\ & = & \mathrm{d}_L(g(J)) & = & \mathrm{d}_L(h(f^\dagger(J))) \\ & = & \mathrm{d}_L(h(I)) & & \end{array}$$

Further, h preserves halt as $h(\mathrm{halt}_{\mathrm{AProc}}) = \theta(\mathrm{halt}_{\mathrm{AProc}}) = \mathrm{halt}_L$. We therefore have that h is a morphism of L_{AProc}-models, which concludes the proof. □

One can go on and obtain a general view of the monad T_{AProc} using a suitable notion of (proper) pure Q-transition sequences. However we omit the details as they are not needed for an account of processes.

There is another possible proof of Theorem 6.2 along the lines of that of Theorem 6.1. First one notes that to have a model of L_{AProc} in ωCpo is to have a model of L_{S} in ωSL, with carrier L, say, together with a morphism $\mathrm{d} : L_\perp \to L$ and an element $\mathrm{halt} \in L$ such that $\mathrm{d}(\Omega) \leq \mathrm{halt}$. It is not hard to see that to have such a morphism and element is to have a morphism $(L + \mathcal{I}_\omega(\mathbb{1}))_\perp \to L$, where $\mathbb{1}$ is the one-point partial order.

One then sees that the carrier of the initial such model is given by the solution of the domain equation:

$$L \cong (S \otimes (L + \mathcal{I}_\omega(\mathbb{1}))_\perp)^S$$

and that that can be solved by first solving the corresponding equation

$$R \cong S \times (S \times (R + \mathbb{1})_\perp)$$

in Pos and then setting $L = \mathcal{I}_\omega(R)$. The rest of the proof proceeds as expected.

Equally, there should be an elementary proof of Theorem 6.1, which, like that of Theorem 6.2, makes use of definability. The more conceptual proofs have the advantage of showing, via domain equations, the origins of the two kinds of transition sequences and their ordering.

6.3. **Processes.** We turn to our algebraic account of Proc. The signature of our theory of processes, L_{Proc}, is that for L_{Res} together with two families of unary operation symbols async_P and $\mathrm{yield_to}_P$, where P is in AProc. The first of these corresponds to the function of the same name defined above, but restricted to asynchronous threads. The second corresponds to a slightly different version of async in which the first action is that of the thread spun off, rather than that of the active command. We often find it convenient to write $\mathrm{async}_P t$ and $\mathrm{yield_to}_P t$ as, respectively, $P \rhd t$ and $P \lhd t$, thinking of them as right and left shuffles.

We begin with a theory L_{Spawn} for async and yield_to which involves the other operations. The first group of equations for L_{Spawn} concerns commutation with $\cup$:

$$\begin{array}{rcl} (P \cup_{\mathrm{AProc}} P') \rhd x & = & (P \rhd x) \cup (P' \rhd x) \\ P \rhd (x \cup y) & = & (P \rhd x) \cup (P \rhd y) \\ (P \cup_{\mathrm{AProc}} P') \lhd x & = & (P \lhd x) \cup (P' \lhd x) \\ P \lhd (x \cup y) & = & (P \lhd x) \cup (P \lhd y) \end{array}$$

The second group of equations concerns the interaction of async with the other operations of L_{Proc} (except for $\lhd$):

$$\begin{array}{rcl} P \rhd \mathrm{update}_{l,n}(x) & = & \mathrm{update}_{l,n}(P \rhd x) \\ P \rhd \mathrm{lookup}_l((x_n)_n) & = & \mathrm{lookup}_l((P \rhd x_n)_n) \\ P \rhd \Omega & = & \Omega \\ P \rhd \mathrm{d}(x) & = & \mathrm{d}(P \bowtie x) \\ P \rhd (P' \rhd x) & = & (P \bowtie P') \rhd x \end{array}$$

where we write $P \bowtie x$ for the "left action" $(P \rhd x) \cup (P \lhd x)$. The first three state that $P \rhd -$ commutes with another operation; the next concerns the interaction of async with suspension and brings in yield_to; the last reduces two occurrences of async to one. The third, and last, group of equations is for the interaction of yield_to with the other operations of L_{APproc}:

$$\begin{array}{rcl} (\mathrm{update}_{l,n})_{\mathrm{APproc}}(P) \lhd x & = & \mathrm{update}_{l,n}(P \lhd x) \\ (\mathrm{lookup}_l)_{\mathrm{APproc}}((P_n)_n) \lhd x & = & \mathrm{lookup}_l((P_n \lhd x)_n) \\ \Omega_{\mathrm{APproc}} \lhd x & = & \Omega \\ \mathrm{d}_{\mathrm{APproc}}(P) \lhd x & = & \mathrm{d}(P \bowtie x) \\ \mathrm{halt}_{\mathrm{APproc}} \lhd x & = & \mathrm{d}(x) \end{array}$$

The first three assert that $- \lhd x$ acts homomorphically with respect to an operation; the next concerns the interaction with suspension; and the last concerns what happens when asynchronous threads halt. Finally we add an inequation:

$$\Omega_{\mathrm{APproc}} \rhd x \leq x$$

We take the equations of L_{Proc} to be those of L_{Spawn}, i.e., the equations are the ones just given for async and yield_to, together with those of L_{Res}. One would naturally have expected L_{Proc} also to have an equation with left-hand side $P \rhd (P' \lhd x)$; indeed, we could have added the equation:

$$P \rhd (P' \lhd x) = P' \lhd (P \bowtie x)$$

However this equation is redundant as it can be proved from the others using the algebraic induction principle of "Computational Induction" described in [PP08]. (One proceeds by such an induction on P', with a subinduction on P.) The inequation is somewhat inelegant: a possible improvement would be to use Pool instead rather than restricting to asynchronous threads. This would give the possibility of a version of halt, to denote Done $\downarrow$, such that the equations

$$\mathrm{halt} \rhd x = \mathrm{halt} \lhd x = x$$

held, making the inequation redundant.

Let T_{Proc} be the monad associated to the theory Proc. We now aim to give a picture of $T_{\mathrm{Proc}}(\mathcal{I}^{\uparrow}_{\omega}(Q))$ like that we gave of $T_{\mathrm{Res}}(\mathcal{I}^{\uparrow}_{\omega}(Q))$. Take the partial order Q-Trans of the *Q-transition sequences* to be that of the basic $(Q \times \mathrm{PSeq})$-transition sequences. Note that one can regard Q-transition sequences as elements of a kind of "double thread" in which the first thread returns a value together with a second (asynchronous) thread.

We show that $Q\text{-Proc} =_{\mathrm{def}} \mathcal{I}_{\omega}(Q\text{-Trans})$ carries the free model of L_{Proc} on $\mathcal{I}^{\uparrow}_{\omega}(Q)$. We view Q-Proc as a L_{Res}-model as in Section 6.1. In order to give async and yield_to, we first mutually recursively define the incomplete right and left shuffles $u \rhd v$ and $u \lhd v$ in Q-Proc

of a proper pure transition sequence u with a Q-transition sequence v, by:

$$\begin{array}{lcll} u \rhd (\sigma, \sigma') & = & \{(\sigma, \sigma')u^-\}\downarrow & \\ u \rhd (\sigma, \sigma'(x, u')) & = & \{(\sigma, \sigma'(x, w)) \mid w \in u \bowtie u'\} \downarrow & \\ u \rhd (\sigma, \sigma')v & = & \{(\sigma, \sigma')w \mid w \in u \bowtie v\} \downarrow & (v \neq \varepsilon) \end{array}$$

where, for any pure transition sequence w, w^- is w less any occurrence of Done, and writing $u \bowtie v$ for the incomplete shuffles $(u \lhd v) \cup (u \rhd v)$ of u and v, and:

$$\begin{array}{lcl} \varepsilon \lhd v & = & \emptyset \\ (\sigma, \sigma')\mathrm{Done} \lhd v & = & \{(\sigma, \sigma')v\} \downarrow \\ (\sigma, \sigma')u \lhd v & = & \{(\sigma, \sigma')w \mid w \in u \bowtie v\} \downarrow \end{array}$$

where, in the last line, u is required to be proper. (Recall that an incomplete shuffle of two sequences is a shuffle of two of their prefixes, equivalently a prefix of a shuffle of them.) Both $\rhd$ and $\lhd$ are monotonic operations.

Then, for $P \in \mathrm{AProc}$ and $I \in Q\text{-Proc}$, we put:

$$(\mathrm{async}_{\mathrm{Proc}})_P(I) = \bigcup_{u \in P,\, v \in I} u \rhd v$$

$$(\mathrm{yield_to}_{\mathrm{Proc}})_P(I) = \bigcup_{u \in P,\, v \in I} u \lhd v \;\cup\; \{u^- \mid u \in P, u \neq \varepsilon\}$$

If I is not empty we have:

$$(\mathrm{yield_to}_{\mathrm{Proc}})_P(I) = \bigcup_{u \in P,\, v \in I} u \lhd v$$

With these additional operations, Q-Proc is a model of L_{Proc}.

In the following we make use of the notation introduced in Section 6.2.

Lemma 6.3. For any proper pure transition sequence u, the equation $u\downarrow \lhd \Omega = \overline{u^-}$ is provable in L_{Proc}.

Proof. The proof is by induction on the length of u. In the case where $u = \varepsilon$, we have $u\downarrow = \Omega_{\mathrm{AProc}}$, and in the equational theory we have $\Omega_{\mathrm{AProc}} \lhd \Omega = \Omega$, as required.

In the case where $u = (\sigma, \sigma')$, we have $u\downarrow = a_{\sigma,\sigma'}(\mathrm{d}\Omega_{\mathrm{AProc}})$, and in the equational theory, we have:

$$\begin{array}{rcl} a_{\sigma,\sigma'}(\mathrm{d}\Omega_{\mathrm{AProc}}) \lhd \Omega & = & a_{\sigma,\sigma'}(\mathrm{d}\Omega_{\mathrm{AProc}} \lhd \Omega) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(\Omega_{\mathrm{AProc}} \lhd \Omega) \cup \mathrm{d}(\Omega_{\mathrm{AProc}} \rhd \Omega)) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}\Omega) \end{array}$$

In the case where $u = (\sigma, \sigma')\mathrm{Done}$, we have $u\downarrow = a_{\sigma,\sigma'}(\mathrm{halt})$, and in the equational theory, we have:

$$\begin{array}{rcl} a_{\sigma,\sigma'}(\mathrm{halt}) \lhd \Omega & = & a_{\sigma,\sigma'}(\mathrm{halt} \lhd \Omega) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}\Omega) \end{array}$$

Finally, in the case where $u = (\sigma, \sigma')v$, with v a proper pure transition sequence, we have $u\downarrow = a_{\sigma,\sigma'}(\mathrm{d}(v\downarrow))$, and in the equational theory, we have:

$$\begin{array}{rcl} a_{\sigma,\sigma'}(\mathrm{d}(v\downarrow)) \lhd \Omega & = & a_{\sigma,\sigma'}(\mathrm{d}(v\downarrow) \lhd \Omega) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}((v\downarrow) \lhd \Omega) \cup \mathrm{d}((v\downarrow) \rhd \Omega)) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}((v\downarrow) \lhd \Omega)) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(\overline{v^-})) \\ & = & \overline{u^-} \end{array}$$

using the induction hypothesis in the next-to-last step. □

Our main algebraic theorem characterizes free models of a natural equational theory for resumptions with thread-spawning in terms of a kind of double-thread.

Theorem 6.4. Viewed as an L_{Proc}-model, $\mathcal{I}_\omega(Q\text{-Trans})$ is the free model over $\mathcal{I}_\omega^\uparrow(Q)$. The unit $(\eta_{T_{\mathrm{Proc}}})_{\mathcal{I}_\omega^\uparrow(Q)} : \mathcal{I}_\omega^\uparrow(Q) \to \mathcal{I}_\omega(Q\text{-Trans})$ is given by:

$$(\eta_{T_{\mathrm{Proc}}})_{\mathcal{I}_\omega^\uparrow(Q)}(I) = \{(\sigma, \sigma\ (x, \mathrm{Done})) \mid x \in I\}\downarrow$$

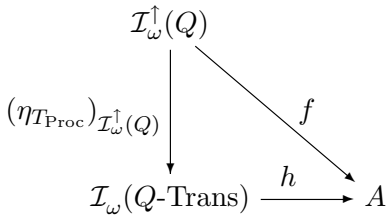
and, for any continuous $f : \mathcal{I}_\omega^\uparrow(Q) \to \mathcal{I}_\omega(R\text{-Trans})$, its Kleisli extension

$$f^\dagger : \mathcal{I}_\omega(Q\text{-Trans}) \to \mathcal{I}_\omega(R\text{-Trans})$$

is given by:

$$\begin{array}{rcl} f^\dagger(I) & = & \{u(\sigma,\tau)v \mid \exists \sigma', x.\ u(\sigma, \sigma'\ (x, \mathrm{Done})) \in I, \\ & & \qquad (\sigma', \tau)v \in f(x\downarrow)\} \\ & & \cup \{u(\sigma,\tau)v \mid \exists \sigma', x, w \neq \mathrm{Done}.\ u(\sigma, \sigma'\ (x, w)) \in I, \\ & & \qquad (\sigma', \tau)v \in w\downarrow \rhd f(x\downarrow)\} \\ & & \cup\ \{u \in I \mid u \text{ has no } (Q \times \mathrm{PSeq}) \text{ transition}\} \end{array}$$

Proof. To show that $\mathcal{I}_\omega(Q\text{-Trans})$ is the free algebra over $\mathcal{I}_\omega^\uparrow(Q)$ with unit as above, we must show that for any L_{Proc}-model A and any continuous function $f : \mathcal{I}_\omega^\uparrow(Q) \to A$ there is a unique morphism $h : \mathcal{I}_\omega(Q\text{-Trans}) \to A$ of models of L_{Proc} such that the following diagram commutes:

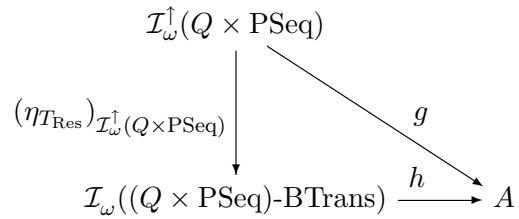


We begin by showing uniqueness. To that end, fix A and f, and let h be a morphism such that the diagram commutes. Define $g : \mathcal{I}_\omega^\uparrow(Q \times \mathrm{PSeq}) \longrightarrow A$ by putting:

$$g((x,u)\downarrow) = \begin{cases} f(x\downarrow) & (\text{if u} = \mathrm{Done}) \\ u\downarrow \rhd_A f(x\downarrow) & (\text{otherwise}) \end{cases}$$

This is a good definition, with monotonicity being established using the inequation for $\rhd$. We have $f = g\alpha$ and $(\eta_{T_{\mathrm{Proc}}})_{\mathcal{I}_\omega^\uparrow(Q)} = (\eta_{T_{\mathrm{Res}}})_{\mathcal{I}_\omega^\uparrow(Q\times\mathrm{PSeq})}\alpha$ where $\alpha : \mathcal{I}_\omega^\uparrow(Q) \to \mathcal{I}_\omega^\uparrow(Q \times \mathrm{PSeq})$ is defined by setting $\alpha(x\downarrow) = (x, \mathrm{Done})\downarrow$.

We then have that the following diagram commutes:



as we may we calculate, for $u = \text{Done}$, that:

$$\begin{array}{rcl} h((\eta_{T_{\text{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q\times \text{PSeq})}((x,u)\downarrow)) & = & h(\eta(x\downarrow)) \\ & = & f(x\downarrow) \\ & = & g((x,u)\downarrow) \end{array}$$

and, for $u \neq \text{Done}$, that:

$$\begin{array}{rcl} h((\eta_{T_{\text{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q\times \text{PSeq})}((x,u)\downarrow)) & = & h(\{(\sigma,\sigma(x,u)) \mid \sigma \in \text{Store}\}\downarrow) \\ & = & h(u\downarrow \rhd \{(\sigma,\sigma(x,\text{Done})) \mid \sigma \in \text{Store}\}\downarrow) \\ & = & h(u\downarrow \rhd (\eta_{T_{\text{Proc}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)}(x\downarrow)) \\ & = & u\downarrow \rhd_A h((\eta_{T_{\text{Proc}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)}(x\downarrow)) \\ & = & u\downarrow \rhd_A f(x\downarrow) \\ & = & g((x,u)\downarrow) \end{array}$$

This is enough to show uniqueness, as if $h(\eta_{T_{\text{Proc}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)} = h'(\eta_{T_{\text{Proc}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)} = f$, for two such morphisms h and h', then $h(\eta_{T_{\text{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q\times \text{PSeq})} = h'(\eta_{T_{\text{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q\times \text{PSeq})} = g$, and so $h = h'$, as h and h' are morphisms of models of L_{Res} (being morphisms of models of L_{Proc}).

For existence we are again given A and f and wish to construct a suitable h. To that end, with g and α as before, take h to be the T_{Res}-extension of g. Then we have $h(\eta_{T_{\text{Proc}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q)} = h(\eta_{T_{\text{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q\times \text{PSeq})}\alpha = g\alpha = f$ and so it remains to prove that h preserves async and yield_to.

As regards the preservation of async, since it is continuous, preserves $\cup$ in each argument, and is strict in its second argument, it suffices to establish preservation for individual transition sequences. That is, it suffices to show, for all proper pure transition sequences u and all v in Q-Trans, that:

$$h(u \rhd v) = u \rhd_A h(v)$$

where here, and below, we omit $\downarrow$'s, writing, e.g., u and v rather than $u\downarrow$ and $v\downarrow$.

As regards the preservation of yield_to, since it is continuous and preserves $\cup$ in each argument, it suffices to show, for all proper pure transition sequences u and all v in Q-Trans that:

$$h(u \lhd v) = u \lhd_A h(v)$$

and:

$$h(u \lhd \Omega) = u \lhd_A h(\Omega)$$

For the last of these three equations, as $h(\Omega) = \Omega$, using Lemma 6.3, we see that is enough to show that $h(\overline{u^-}) = \overline{u^-}$, and this holds as h is a homomorphism of models of L_{Res}.

The proof of the first two equations is a simultaneous induction on the sum of the lengths of u and v, invoking L_{Proc} equations on A as necessary. We begin with the first equation. In the first case, we consider $v = (\sigma,\sigma')$. Here, on the one hand, we have:

$$h(u \rhd (\sigma,\sigma')) = h((\sigma,\sigma')u^-) = h(\overline{(\sigma,\sigma')u^-}) = \overline{(\sigma,\sigma')u^-}$$

using the fact that h is a homomorphism for the last equality, and, on the other, we have:

$$\begin{array}{rcll} u \rhd_A h((\sigma,\sigma')) & = & u \rhd_A h(a_{\sigma,\sigma'}(\mathrm{d}\Omega)) & \\ & = & u \rhd_A (a_{\sigma,\sigma'}(\mathrm{d}\Omega)) & \\ & = & a_{\sigma,\sigma'}(u \rhd_A \mathrm{d}\Omega) & \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(u \rhd_A \Omega) \cup \mathrm{d}(u \lhd_A \Omega)) & \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(\overline{u^-})) & \text{(by Lemma 6.3)} \\ & = & \overline{(\sigma,\sigma')u^-} & \end{array}$$

For the next case we consider $v = (\sigma, \sigma'(x, u'))$. Here, on the one hand we have:

$$\begin{array}{rcl} h(u \rhd v) & = & h(\{(\sigma, \sigma'(x, u'')) \mid u'' \in u \bowtie u'\}) \\ & = & \bigcup_{u'' \in u \bowtie u'} h((\sigma, \sigma'(x, u''))) \\ & = & \bigcup_{u'' \in u \bowtie u'} a_{\sigma,\sigma'}(h((\eta_{T_{\mathrm{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q \times \mathrm{PSeq})}(x, u''))) \\ & = & \bigcup_{u'' \in u \bowtie u'} a_{\sigma,\sigma'}(u'' \rhd_A f(x)) \\ & = & a_{\sigma,\sigma'}(\bigcup_{u'' \in u \bowtie u'}(u'' \rhd_A f(x))) \\ & = & a_{\sigma,\sigma'}((u \bowtie u') \rhd_A f(x)) \end{array}$$

and, on the other hand, we have:

$$\begin{array}{rcl} u \rhd_A h(v) & = & u \rhd_A h(a_{\sigma,\sigma'}((\eta_{T_{\mathrm{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q \times \mathrm{PSeq})}(x, u'))) \\ & = & a_{\sigma,\sigma'}(u \rhd_A h((\eta_{T_{\mathrm{Res}}})_{\mathcal{I}^{\uparrow}_{\omega}(Q \times \mathrm{PSeq})}(x, u'))) \\ & = & a_{\sigma,\sigma'}(u \rhd_A (u' \rhd_A f(x))) \end{array}$$

For the last case for the first equation we have $v = (\sigma, \sigma')v'$, with v' in Q-Trans, and we calculate:

$$\begin{array}{rcllcl} h(u \rhd (\sigma, \sigma')v') & = & a_{\sigma,\sigma'}(h(u \rhd \mathrm{d}(v'))) & & = & a_{\sigma,\sigma'}(h(\mathrm{d}(u \rhd v' \cup u \lhd v'))) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(h(u \rhd v') \cup h(u \lhd v'))) & & = & a_{\sigma,\sigma'}(\mathrm{d}(u \rhd_A h(v') \cup u \lhd_A h(v'))) \\ & = & a_{\sigma,\sigma'}(u \rhd_A (\mathrm{d}(h(v')))) & & = & u \rhd_A a_{\sigma,\sigma'}(\mathrm{d}(h(v'))) \\ & = & u \rhd_A h((\sigma, \sigma')v') & & & \end{array}$$

applying the induction hypothesis in the second line.

Turning to the second equation, the first case we consider is where $u = \varepsilon$, and we have:

$$h(\varepsilon \lhd v) \;=\; h(\Omega \lhd v) \;=\; h(\Omega) \;=\; \Omega \;=\; \Omega \lhd_A h(v) \;=\; \varepsilon \lhd_A h(v)$$

The second case is where $u = (\sigma, \sigma')\mathrm{Done}$ and we have:

$$\begin{array}{rcllcl} h((\sigma, \sigma')\mathrm{Done} \lhd v) & = & h((\sigma, \sigma')v) & & = & a_{\sigma,\sigma'}(\mathrm{d}(hv)) \\ & = & a_{\sigma,\sigma'}(\mathrm{halt} \lhd_A h(v)) & & = & a_{\sigma,\sigma'}(\mathrm{halt}) \lhd_A h(v) \\ & = & (\sigma, \sigma')\mathrm{Done} \lhd_A h(v) & & & \end{array}$$

The last case is where $u = (\sigma, \sigma')u'$, with u' a proper pure transition sequence, and we have:

$$\begin{array}{rcllcl} h((\sigma, \sigma')u' \lhd v) & = & h(a_{\sigma,\sigma'}(\mathrm{d}(u')) \lhd v) & & = & h(a_{\sigma,\sigma'}(\mathrm{d}(u' \bowtie v))) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(h(u' \bowtie v))) & & = & a_{\sigma,\sigma'}(\mathrm{d}(u' \bowtie_A h(v))) \\ & = & a_{\sigma,\sigma'}(\mathrm{d}(u')) \lhd_A h(v) & & = & (\sigma, \sigma')u' \lhd_A h(v) \end{array}$$

applying the induction hypothesis to obtain the fourth equality.

Finally, the formula for the Kleisli extension follows from the construction of h, using the Kleisli formula of Theorem 6.1.

□

As in the case of resumptions, one can go further and obtain a closely related, if less elementary, picture of $T_{\mathrm{Proc}}(P)$ for arbitrary P.

Note that the proof of Theorem 6.4 is elementary, making use of definability in a similar way to the proof of Theorem 6.2. However, unlike in the cases of Theorems 6.1 and 6.2, we do not know any conceptual proof of Theorem 6.4. The difficulty is that the theory of processes L_{Proc}, particularly the part concerning $\lhd$ and $\rhd$, seems somewhat ad hoc, and is not built up in a standard way from simpler theories. There is surely more to be understood here.

Nonetheless, with Theorem 6.4 available, we are in a position to give our algebraic account of Proc. There is an isomorphism $\theta_{\mathrm{Proc}} : Q\text{-Trans} \to \mathrm{TSeq} \setminus \{\varepsilon\}$, where $Q = \{\mathrm{Ret}\}$,

sending $u = (\sigma_1, \sigma_1') \dots (\sigma_n, \sigma_n')$ to itself and $u(\sigma, \sigma'\ (\mathrm{Ret}, v))$ to $u(\sigma, \sigma'\ \mathrm{Ret})v$. One then has an isomorphism of ω-cpos $\tilde{\theta}_{\mathrm{Proc}} : \mathcal{I}_\omega(Q\text{-Trans}) \cong \mathrm{Proc}$ given by: $\tilde{\theta}_{\mathrm{Proc}}(I) = \theta_{\mathrm{Proc}}(I) \cup \{\varepsilon\}$. It follows that Proc can be seen as the free model of L_{Proc} over the terminal ω-cpo $\{\mathrm{Ret}\}$, as we now spell out. First, define the set of left shuffles $u \lhd v$ of a pure transition sequence u with a transition sequence v by setting

$$\varepsilon \lhd v = \{\varepsilon\}$$

and

$$(\sigma, \sigma')u \lhd v = \{(\sigma, \sigma')w \mid w \in u \bowtie v\}$$

Then, we have:

Corollary 6.5. Equip Proc with the following operations:

$$\begin{array}{rcl}
(\mathrm{update}_{x,n})_{\mathrm{Proc}}(P) & = & \{(\sigma, \tau)u \mid (\sigma[x \mapsto n], \tau)u \in P\} \cup \{\varepsilon\} \\
(\mathrm{lookup}_x)_{\mathrm{Proc}}((P_n)_n) & = & \bigcup_n \{(\sigma, \tau)u \in P_n \mid \sigma(x) = n\} \cup \{\varepsilon\} \\
P \cup_{\mathrm{Proc}} Q & = & P \cup Q \\
\Omega_{\mathrm{Proc}} & = & \{\varepsilon\} \\
\mathrm{d}_{\mathrm{Proc}}(P) & = & \{(\sigma, \sigma)u \mid \sigma \in \mathrm{Store}, u \in P\} \cup \{\varepsilon\} \\
P \rhd_{\mathrm{Proc}} Q & = & \mathrm{async}(P, Q) \\
P \lhd_{\mathrm{Proc}} Q & = & \bigcup_{u \in P, v \in Q} u \lhd v
\end{array}$$

(where x ranges over Vars).

Then $\tilde{\theta}_{\mathrm{Proc}} : \mathcal{I}_\omega(Q\text{-Trans}) \cong \mathrm{Proc}$ is an isomorphism of L_{Proc}-models, and Proc is the free model of L_{Proc} over $\{\mathrm{Ret}\}$, with unit $(\eta_{\mathrm{Proc}})_{\{\mathrm{Ret}\}} : \{\mathrm{Ret}\} \to \mathrm{Proc}$ given by:

$$(\eta_{\mathrm{Proc}})_{\{\mathrm{Ret}\}}(\mathrm{Ret}) = \{(\sigma, \sigma\,\mathrm{Ret})\mathrm{Done} \mid \sigma \in \mathrm{Store}\}\!\downarrow$$

The Kleisli extension of a map $f : \{\mathrm{Ret}\} \to \mathrm{Proc}$ is given by:

$$f^\dagger(P) = P \circ f(\mathrm{Ret})$$

Proof. The proof is a calculation using Theorem 6.4. The following equations are useful:

$$\tilde{\theta}_{\mathrm{Proc}}(u \rhd v) = (u \rhd \theta_{\mathrm{Proc}}(v))\!\downarrow$$

$$\tilde{\theta}_{\mathrm{Proc}}(u \lhd v) = (u \lhd \theta_{\mathrm{Proc}}(v))\!\downarrow$$

where u is a proper pure transition sequence and v is a $\{\mathrm{Ret}\}$-transition sequence. □

As we now see, the algebraic view also determines the semantics of our language. This achieves our aim of placing cooperative threads within the algebraic approach to effects, thereby justifying the previous, more ad hoc, account.

First, we have that $[\![\texttt{skip}]\!] = (\eta_{\mathrm{Proc}})_{\{\mathrm{Ret}\}}(\mathrm{Ret})$ and that $P \circ Q = (\mathrm{Ret} \mapsto Q)^\dagger(P)$, so the Kleisli structure determines the semantics of `skip` and composition, just as one would expect from the monadic point of view.

Next, the update and lookup operations, together with the assumed primitive natural number and boolean functions, determine the semantics of assignments, conditionals, and while loops. The operations are equivalent to two generic effects, of assignment and reading:

$$:= : \mathrm{Vars} \times \mathbb{N} \to \mathrm{Proc} \qquad ! : \mathrm{Vars} \to T_{\mathrm{Proc}}(\mathbb{N})$$

One can use the reading generic effect to give the semantics of numerical expressions as elements of $T_{\mathrm{Proc}}(\mathbb{N})$; with that, one can give the semantics of assignments, using the assignment generic effect, standard monadic means, and $\tilde{\theta}_{\mathrm{Proc}}$. Similarly, one can use the

reading generic effect to give the semantics of boolean expressions as elements of $T_{\mathrm{Proc}}(\mathbb{B})$, where $\mathbb{B} =_{\mathrm{def}} \{\mathtt{true}, \mathtt{false}\}$; with that one can give the semantics of conditionals and while loops, again using standard monadic means and $\tilde{\theta}_{\mathrm{Proc}}$ (as well as least fixed-points for while loops).

Continuing, the d operation is that of the algebra; and `block` is modeled by Ω_{Proc}. Finally, the semantics of spawning is determined by async together with the cleaning function

$$-^c : \mathrm{Proc} \to \mathrm{AProc}$$

It turns out that the latter is also determined by algebraic means. Specifically, one can regard AProc as a model of L_{Res} as in Section 6.2 (so we ignore halt) and then extend it to a model of L_{Proc} as follows. First for any proper pure transition sequences u and v we define $u \rhd v \in \mathrm{AProc}$ inductively on v by:

$$\begin{array}{lcl} u \rhd \varepsilon & = & \{\varepsilon\} \\ u \rhd (\sigma, \sigma')\mathrm{Done} & = & \{(\sigma, \sigma')u\}\downarrow \\ u \rhd (\sigma, \sigma')v & = & \{(\sigma, \sigma')w \mid w \in u \bowtie v\}\downarrow \end{array}$$

where, in the last line, v is required to be proper. Then we put:

$$(\mathrm{async}_{\mathrm{AProc}})_P(Q) = \bigcup_{u \in P, v \in Q} u \rhd v$$

and $(\mathrm{yield_to}_{\mathrm{AProc}})_P(Q) = (\mathrm{async}_{\mathrm{AProc}})_Q(P)$. With these definitions, $-^c$ is the extension of the map $\mathrm{Ret} \mapsto \mathrm{halt}_{\mathrm{AProc}}$ to Proc.

In the converse direction one can consider adding missing algebraic operations to the language, for example adding $\cup$ and yield_to via constructs C `or` D and `yield_to` C. The latter construct is to the binary yield_to as `async` is to the binary async. It generalizes `yield`, which is equivalent to `yield_to skip`. Its operational semantics is given by the rule:

$$\langle \sigma, T, \mathcal{E}[\mathtt{yield_to}\ C] \rangle \quad \longrightarrow \quad \langle \sigma, T.\mathcal{E}[\mathtt{skip}], C \rangle$$

One may debate the programming usefulness of such additional constructs, but they do allow one to express the equations used for the algebraic characterizations at the level of commands. For example, the equation $P \rhd \mathrm{d}(x) = \mathrm{d}(P \bowtie x)$ becomes:

$$\begin{array}{l} (\mathtt{async}\ C); \mathtt{yield}; D \\ \qquad = \\ \mathtt{yield}; ((\mathtt{async}\ C); D\ \mathtt{or}\ (\mathtt{yield_to}\ C); D) \end{array}$$

6.4. **Dendriform Algebras and Modules.** We have found it useful to employ various forms of shuffle: sometimes we shuffle two things of the same kind with each other, e.g., two pure transition sequences with each other; and sometimes we shuffle two things of different kinds with each other, e.g., a pure transition sequence with a transition sequence.

We have further found it useful to break down such shuffles into left and right shuffles, e.g., in the case of the left and right shuffles of asynchronous processes with processes; indeed we employ a uniform notation, writing $\lhd, \rhd$, and $\bowtie$ for left shuffles, right shuffles, and (ordinary) shuffles, respectively. Our algebraic account of threads has further involved a number of equations concerning the interaction of these shuffle operations with each other and with other operations.

Shuffle operations and their algebra have been studied in a variety of settings. In particular, Loday's dendriform algebras [Lod01, FG08] provide a wide-ranging general notion of

left and right shuffling of two things of the same kind with each other. Foissy's dendriform A-modules [Foi07] provide the corresponding notion of action: left or right shuffling a thing of one kind with a thing of another kind. We next relate our treatment to these general concepts, thereby placing our various shuffle operations and our equations for them in a standard algebraic context.

Let R be a given commutative semiring (with no requirement for a 0 or a 1). Then a *dendriform dialgebra* is an R-module A equipped with two binary bilinear operations $\lhd$ and $\rhd$ such that, for all $x, y, z \in A$:

$$\begin{array}{rcl}(x \lhd y) \lhd z & = & x \lhd (y \bowtie z) \\ x \rhd (y \rhd z) & = & (x \bowtie y) \rhd z \\ (x \rhd y) \lhd z & = & x \rhd (y \lhd z)\end{array}$$

where $x \bowtie y =_{\mathrm{def}} x \lhd y + y \rhd x$; it is *commutative* if $x \lhd y = y \rhd x$ always holds. Then $(A, \bowtie)$ is a semigroup in the category of R-modules, equivalently $\bowtie$ is an associative bilinear operation; it is commutative if the dialgebra is.

Given a dendriform algebra A, a *dendriform A-module* is an R-module M equipped with two binary bilinear operations $\lhd, \rhd : A \times M \to M$ such that, for all $a, b \in A$ and $x \in M$:

$$\begin{array}{rcl}(a \lhd b) \lhd x & = & a \lhd (b \bowtie x) \\ a \rhd (b \rhd x) & = & (a \bowtie b) \rhd x \\ (a \rhd b) \lhd x & = & a \rhd (b \lhd x)\end{array}$$

where $\bowtie\colon A \times M \to M$ is given by: $a \bowtie x = a \lhd x + a \rhd x$. Then $\bowtie\colon A \times M \to M$ is a bilinear action of $(A, \bowtie)$ on M.

In all our examples we take R to be the natural two-element semiring over $\mathbb{B}$; join semilattices with a zero form $\mathbb{B}$-modules (setting $\mathtt{true}\, x = x$ and $\mathtt{false}\, x = 0$). As a first example, consider the $\mathbb{B}$-module of the collection of all languages, i.e., all sets of strings over a given alphabet, not containing ε. This is a commutative dialgebra, taking $\lhd$ to be the left shuffle operation, and $\rhd$ to be the right one; $\bowtie$ is then the usual shuffle operation.

The semilattice of asynchronous processes AProc forms a commutative dendriform $\mathbb{B}$-algebra, setting:

$$P \lhd_{\mathrm{AProc}} Q = (\mathrm{yield_to}_{\mathrm{AProc}})_P(Q) \qquad P \rhd_{\mathrm{AProc}} Q = (\mathrm{async}_{\mathrm{AProc}})_P(Q)$$

One then has that Q-Proc forms a dendriform AProc-module, setting:

$$P \lhd_{Q\text{-}\mathrm{Proc}} I = (\mathrm{yield_to}_{\mathrm{Proc}})_P(I) \qquad P \rhd_{Q\text{-}\mathrm{Proc}} I = (\mathrm{async}_{\mathrm{Proc}})_P(I)$$

It follows that Proc also forms a dendriform AProc-module, using the definitions of the left and right shuffling given in Corollary 6.5.

Algebraically, the first group of equations for L_{Spawn} state the bilinearity of the two module operations. The second group contains the second of the three module equations. The equation

$$a \rhd (b \lhd x) = b \lhd (a \bowtie x)$$

generalizing one considered above, holds in any module over a commutative dendriform algebra. To account for the other two module equations algebraically one would need an algebraic treatment of the dendriform algebra operations on AProc. These operations are effect deconstructors rather than effect constructors. An account of unary deconstructors has been given in [PP09], but a satisfactory treatment of binary ones remains to be found; we therefore leave further algebraic treatment to future work.

7. Conclusion

A priori, the properties and the semantics of threads in general, and of cooperative threads in particular, may not appear obvious. In our opinion, a huge body of incorrect multithreaded software and a relatively small literature both support this point of view. With the belief that mathematical foundations could prove beneficial, the main technical goal of our work is to define and elucidate the semantics of threads. For instance, semantics can serve for validating reasoning principles; our work is only a preliminary, but encouraging, step in this respect.

Our initial motivation was partly practical—we wanted to understand and further the AME programming model and similar ones. We also saw an opportunity to leverage developments in trace-based denotational semantics and in the algebraic theory of effects, and to extend their applicability to threads. As our results demonstrate, the convergence of these three lines of work proved interesting and fruitful.

We focus on a particular small language with constructs for threads. Several possible extensions may be considered. These include constructs for parallel composition, nondeterministic choice, higher-order functions, and thread-joining. More speculatively, they also include generalized yields, of the kind that arise in the algebraic theory of effects, as discussed in Section 6. Importantly, our monadic treatment of threads indicates how to add higher-order functions to the semantics.

Our results mostly carry over to these extensions. In some cases, small changes or restrictions are required. In particular, the full-abstraction proof with nondeterministic choice would use fresh variables; the one for higher-order functions might require standard limitations on the order of functions, cf. [Jef95]. Thus, our approach seems to be robust, and indeed—as in the case of higher-order functions—helpful in accounting for a range of language features. Further, our algebraic analysis of the thread monad links it to the broader theme of the algebraic treatment of effects. In that regard, as the discussion after Theorem 6.4 indicates, there is clearly still further understanding to be gained.

Another possible direction for further work is the exploration of alternative semantics. For instance, we could switch from the "may" semantics that we study to "must" semantics. We could also define alternative notions of observation. As suggested in Section 5.3, some of the coarser notions of observation might require closure conditions, such as closure under suitable forms of stuttering and under mumbling. These may correspond to suitable axioms on the suspension operator d, as alluded to in [Plo06]: we conjecture that stuttering corresponds to $\mathrm{d}(\mathrm{d}(x)) \leq \mathrm{d}(x)$ and that mumbling corresponds to $\mathrm{d}(x) \geq x$.

It would also be interesting to consider finer notions of observation that distinguish blocking from divergence. To this end we could add constructs such as `orElse` [HMP05] and, in the semantics, treat blocking as a kind of exception. Finally, we could revisit lower-level semantics with explicit optimistic concurrency and roll-backs, of the kind employed in the implementation of AME.

Acknowledgements

We are grateful to Martín Escardó and Martin Hyland for their helpful comments and suggestions.

References

[ABH08] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *Proc. 35th. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (eds. George C. Necula and Philip Wadler), 63–74, ACM Press, 2008.

[AP93] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theor. Comput. Science*, 114(1):3–30, 1993.

[AP09] Martín Abadi and Gordon D. Plotkin. A model of cooperative threads. *Proc. 36th. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (eds. Zhong Shao and Benjamin C. Pierce), 29–40, ACM Press, 2009.

[Abr79] Karl Abrahamson. Modal logic of concurrent nondeterministic programs. *Proc. Int. Symp. on Semantics of Concurrent Computation* (ed. Gilles Kahn), Lect. Notes Comput. Sci., 70:21–33, Springer, 1979.

[AI07] Luca Aceto and Anna Ingólfsdóttir. The saga of the axiomatization of parallel composition. *Proc. 16th. Int. Conf. on Concurrency Theory* (eds. Luís Caires and Vasco Thudichum Vasconcelos), Lect. Notes Comput. Sc., 4703:2–16, Springer, 2007.

[AHT02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. *USENIX Annual Technical Conf., General Track* (ed. Carla Schlatter Ellis), 289–302, 2002.

[AZ06] Roberto Amadio and Silvano Dal Zilio. Resource control for synchronous cooperative threads. *Theor. Comput. Science*, 358:229–254, 2006.

[BHM02] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. *Advanced Lectures from Int. Summer School on Applied Semantics* (eds. Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva), Lect. Notes Comput. Sci., 2395:42–122, Springer, 2002.

[BMT92] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. *Proc. 19th. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 119–129, ACM Press, 1992.

[Bou07] Gérard Boudol. Fair cooperative multithreading. *Proc. 18th. Int. Conf. on Concurrency Theory* (eds. Luís Caires and Vasco Thudichum Vasconcelos), Lect. Notes Comput. Sci., 4703:272–286, Springer, 2007.

[Bou06] Fréderic Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, 2006.

[Bro96] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Inform. Comput.*, 127(2):145–163, 1996.

[Bro02] Stephen Brookes. The essence of parallel Algol. *Inform. Comput.*, 179(1):118–149, 2002.

[CGA05] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Proc. 20th. Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (eds. Ralph E. Johnson and Richard P. Gabriel), 519–538, ACM Press, 2005.

[CM93] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. *Proc. 5th. Biennial Meeting on Category Theory and Computer Science*, 1993.

[FG08] Kurusch Ebrahimi-Fard and Li Guo. Rota-Baxter Algebras and Dendriform Algebras. *J. Pure Appl. Algebra*, 212(2), 320-339, 2008.

[FH99] William Ferreira and Matthew Hennessy. A behavioural theory of first-order CML. *Theor. Comput. Science*, 216(1-2):55–107, 1999.

[Foi07] Loïc Foissy. Bidendriform bialgebras, trees, and free quasi-symmetric functions. *J. Pure Appl. Algebra*, 209(2):439–459, 2007.

[GMR09] Pierre Ganty, Rupak Majumdar, and Andrey Rybalchenko. Verifying liveness for asynchronous programs. *Proc. 36th. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (eds. Zhong Shao and Benjamin C. Pierce), 102–113, ACM Press, 2009.

[GMP06] Dan Grossman, Jeremy Manson and William Pugh, What do high-level memory models mean for transactions? Proc. 2006 Workshop on Memory System Performance and Correctness (eds. Antony L. Hosking and Ali-Reza Adl-Tabatabai), pp. 62–69, ACM Press, 2006.

[HMP05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. *Proc. 10th. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (eds. Keshav Pingali, Katherine A. Yelick and Andrew S. Grimshaw), 48–60, ACM Press, 2005.

[HP79] Matthew Hennessy and Gordon D. Plotkin. Full abstraction for a simple programming language. *Proc. 8th. Symp. on Mathematical Foundations of Computer Science* (ed. J. Bečvář), Lect. Notes Comput. Sci., 74:108–120, Springer, 1979.

[HdeBR94] E. Horita, J. W. de Bakker, and J. J. M. M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Inform. Comput.*, 115(1):125–178, 1994.

[HPP06] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: sum and tensor. *Theor. Comput. Science*, 357(1–3):70–99, 2006.

[IB07] Michael Isard and Andrew Birrell. Automatic mutual exclusion. *Proc. 11th. USENIX Workshop on Hot Topics in Operating Systems*, 1–6, 2007.

[Jef95] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. *Proc. 10th. Symp. on Logic in Computer Science*, 255–264, IEEE Press, 1995.

[Jef97] Alan Jeffrey. Semantics for core Concurrent ML using computation types. *Higher Order Operational Techniques in Semantics* (eds. Andrew D. Gordon and Andrew M. Pitts), 55–90, Cambridge University Press, 1997.

[JR05] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Science*, 338(1–3):17–63, 2005.

[JM07] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. *Proc. 34th. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (eds. Martin Hofmann and Matthias Felleisen), 339–350, ACM Press, 2007.

[Lod01] Jean-Louis Loday. Dialgebras. *Dialgebras and related operads*, Lect. Notes Math., 1763:7–66, Springer, 2001.

[SQL07] Microsoft. SQL Server 2005 books online. CLR Hosted Environment, at `http://msdn.microsoft.com/en-us/library/ms131047.aspx`, 2007.

[PR97] Prakash Panangaden and John H. Reppy. The essence of Concurrent ML. *ML with Concurrency* (ed. Flemming Nielson), 5–29, Springer, 1997.

[Plo06] Gordon D. Plotkin. Hennessy-Plotkin-Brookes Revisited. *Proc. 26th. Foundations of Software Technology and Theoretical Computer Science* (eds. S. Arun-Kumar & Naveen Garg), Lect. Notes Comput. Sci., 4337:4, Springer, 2006

[PP02] Gordon Plotkin and John Power. Notions of computation determine monads. *Proc. 5th. Int. Conf. on Foundations of Software Science and Computation Structures* (eds. Mogens Nielsen and Uffe Engberg), Lect. Notes Comput. Sci., 2303:373–393, Springer, 2002.

[PP03] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Appl. Categor. Struct.*, 11(1):69–94, 2003.

[PP08] Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. *Proc. 23rd. Symp. on Logic in Computer Science*, 118–129, IEEE Press, 2008.

[PP09] Gordon D. Plotkin and Matija Pretnar. Handlers of Algebraic Effects. *Proc. 18th. European Symp. on Programming* (ed. Giuseppe Castagna), Lect. Notes Comput. Sci., 5502:80–94, Springer, 2009.

[SJ05] Vijay A. Saraswat and Radha Jagadeesan. Concurrent clustered programming. *Proc. 16th. Int. Conf. on Concurrency Theory* (eds. Martín Abadi and Luca de Alfaro), Lect. Notes Comput. Sci., 3653:353–367, Springer, 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. *Proc. 14th. Annual ACM Symp. on Principles of Distributed Computing*, 204–213, ACM Press, 1995.

[SKB07] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. *Proc. 22nd. Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (eds. Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes and Guy L. Steele Jr.), 191–210, ACM Press, 2007.

[BCZ03] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. Capriccio: scalable threads for Internet services. *Proc. 19th. ACM Symp. on Operating Systems Principles* (eds. Michael L. Scott and Larry L. Peterson), 268–281, ACM Press, 2003.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.