A Model of Cooperative Threads

Martín Abadi
Microsoft Research, Silicon Valley and
University of California, Santa Cruz
abadi@microsoft.com

Gordon Plotkin
LFCS, University of Edinburgh and
Microsoft Research, Silicon Valley
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.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Languages, Theory

Keywords denotational semantics, monad, operational semantics, transaction

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., [7, 23, 20, 21]; denotational semantics of those languages seem to be much rarer, and to address message passing rather than shared-memory concurrency, e.g., [13, 19]. 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 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., [22]. Cooperative threads also arise in other contexts, such as Internet services and synchronous programming [4, 29, 9, 8, 5]. Most recently, cooperative threads are central in two models for programming with transactions, Automatic Mutual Exclusion (AME) and Transactions with Isolation and Cooperation (TIC) [18, 28]. AME is one of the main starting points for our research. The intended implementations of AME rely on software transactional memory [27] for executing multiple cooperative threads simultaneously. However, concurrent transactions do not appear in the high-level operational semantics of the AME constructs [1]. 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

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

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

Figure 1. Syntax.

of effects and the analysis of the denotational semantics in this monadic setting (Theorem 6.3). Section 7 concludes. The appendix outlines some of the more interesting proofs.

2. The Language

Our language is an extension of a basic imperative language with assignments, sequencing, conditionals, and while loops (IMP [30]). 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:

- 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.

Thus, we can for example write (Figure 2) 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 $x := 0$

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

Figure 2. Example command.

may happen 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 [18, 1]. 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.

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

Figure 4. State space.

We define the syntax of the language in Figure 1. We do not detail the constructs on numerical expressions, nor those for boolean conditions, which are as usual.

Thus, our language is basically a fragment of the AME calculus [1]. 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`. 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 4, 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:

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

3.3 Steps

A transition $\Gamma \longrightarrow \Gamma'$ takes an execution from one state to the next. Figure 3 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.

$$\langle \sigma, T, \mathcal{E}[x := e] \rangle \longrightarrow \langle \sigma[x \mapsto n], T, \mathcal{E}[\texttt{skip}] \rangle \quad \text{if } \sigma(e) = n \qquad \text{(Set)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{skip}; C] \rangle \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle \qquad \text{(Seq)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle \longrightarrow \langle \sigma, T, \mathcal{E}[C] \rangle \quad \text{if } \sigma(b) = \texttt{true} \qquad \text{(Cond True)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } C \texttt{ else } D] \rangle \longrightarrow \langle \sigma, T, \mathcal{E}[D] \rangle \quad \text{if } \sigma(b) = \texttt{false} \qquad \text{(Cond False)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{while } b \texttt{ do } C] \rangle \longrightarrow \langle \sigma, T, \mathcal{E}[\texttt{if } b \texttt{ then } (C; \texttt{while } b \texttt{ do } C) \texttt{ else skip}] \rangle \qquad \text{(While)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{async } C] \rangle \longrightarrow \langle \sigma, T.C, \mathcal{E}[\texttt{skip}] \rangle \qquad \text{(Async)}$$

$$\langle \sigma, T, \mathcal{E}[\texttt{yield}] \rangle \longrightarrow \langle \sigma, T.\mathcal{E}[\texttt{skip}], \texttt{skip} \rangle \qquad \text{(Yield)}$$

$$\langle \sigma, T.C.T', \texttt{skip} \rangle \longrightarrow \langle \sigma, T.T', C \rangle \qquad \text{(Activate)}$$

Figure 3. Transition rules of the abstract machine.

We write $\Gamma \longrightarrow_c \Gamma'$ when $\Gamma \longrightarrow \Gamma'$ via the Activate 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 [3] and appears in various studies of parallel composition [2, 16, 10, 11]. 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 $C_1; \texttt{async } (C_2)$, any occurrence of "$\mathrm{Ret}$" 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

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

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

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

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 $u(\sigma_n, \sigma_n' \ \mathrm{Ret})u'$ 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 $u(\sigma_n, \sigma_n' \ \mathrm{Ret})u'$ 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 $u(\sigma_n, \sigma_n' \ \mathrm{Ret})u'$ and the meaning of D contains $(\sigma_n', \sigma_n'' \ \mathrm{Ret})v$ (a trace that starts with a transition with a marker). Moreover, if

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

Figure 5. Denotational semantics.

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 $(\sigma, \sigma\ \mathrm{Ret})$ plus every sequence of the form $(\sigma, \sigma\ \mathrm{Ret})\mathrm{Done}$. 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 $(\sigma, \sigma\ \mathrm{Ret})$, but not $(\sigma, \sigma\ \mathrm{Ret})\mathrm{Done}$.

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

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

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 $(\sigma, \sigma'\ \mathrm{Ret})$ is a prefix of $(\sigma, \sigma'\ \mathrm{Ret})\mathrm{Done}$, but (σ, σ') is not a prefix of $(\sigma, \sigma'\ \mathrm{Ret})$. Such differences are essential.

4.2 Transition Sequences

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

A *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 in the sense above. It is *proper* if it is not equal to Done. We write PSeq for the set of pure transition sequences.

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 sequence, pure or not).
- For a non-empty sequence of transitions t, we write $\mathrm{fst}(t)$ 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 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, where 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. 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 cpo; below it also serves for the semantics of thread pools.

We define a continuous *clean* function

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

by:

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

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

We define the set of *shuffles* of a pure transition sequence u with a sequence v 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 $v\,\mathrm{Done}$ is a shuffle of u and v. Similarly, if v does not finish with Done, then a shuffle of $u\,\mathrm{Done}$ and v is a shuffle of u and v.
- A shuffle of $u\,\mathrm{Done}$ and $v\,\mathrm{Done}$ is a shuffle of u and v followed by Done.

We write $u \bowtie v$ for the set of shuffles of u and v.

We define a continuous *composition* function

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

by:

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

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

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

We also define a continuous *delay* function

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

by:

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

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

$$\mathrm{async} : \mathrm{Pool} \to \mathrm{Proc}$$

by:

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

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

Interpretation The denotational semantics

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

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:

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

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

We straightforwardly extend the semantics to contexts, so that

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

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 4.1. $[\![\mathcal{C}[C]]\!] = [\![\mathcal{C}]\!]([\![C]\!])$. *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.

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 $I =_{\mathrm{def}} \{\varepsilon, \mathrm{Done}\}$.

We define the *(left) action* $u \cdot v$ of a pure transition sequence u on a transition sequence v, by setting

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

and

$$u \cdot \varepsilon = \{\varepsilon\}$$

We then define

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

by:

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

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 $[\![\texttt{yield}]\!]$).

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 4.2. *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. $\mathrm{async}(I,R) = R$

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:
$$\begin{array}{lcl}[\![\texttt{yield};D]\!]^c & = & [\![\texttt{async}\ (D)]\!]^c \\ & = & \{(\sigma,\sigma)u^c \mid \sigma \in \mathrm{Store}, u \in [\![D]\!]\}\downarrow;\end{array}$$
- 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). 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` are viable, however. We briefly discuss those variants and others in Section 7.

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, 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 also applies to runs, which are essentially traces that the command can produce on its own, i.e., with an empty context.

The full-abstraction theorem implies that, if two commands C and D have the same set of traces denotationally, then they produce the same runs in combination with every context. In other words, observing runs, we cannot distinguish C and D in any context. 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 $\sigma' = \sigma''$. For such a pure transition sequence $w = (\sigma_1,\sigma_2)\dots(\sigma_{n-1},\sigma_n)$, 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 $\mathrm{runs}(P)$ for the set of runs generated by (pure) transition sequences in P.

5.2 Adequacy

LEMMA 5.1. *The following equalities hold:*

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

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

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

LEMMA 5.3. $[\![T, \mathtt{skip}]\!] = \{(\sigma,\sigma\ \mathrm{Ret})v \mid v \in [\![T]\!]\}{\downarrow}$.

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 $(\sigma',\sigma'')v \in [\![T', C']\!]^c$.*

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

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:

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

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 $C \trianglelefteq D$ 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 $i \geq 0$:

$$\mathtt{block} \trianglelefteq D \qquad \frac{C \trianglelefteq D}{(\mathtt{while}\ b\ \mathtt{do}\ C)_i \trianglelefteq (\mathtt{while}\ b\ \mathtt{do}\ D)}$$

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

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

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:

$$(\mathtt{async}\ C)^{(i)} = \mathtt{async}\ C^{(i)}$$
$$(\mathtt{if}\ b\ \mathtt{then}\ C\ \mathtt{else}\ D)^{(i)} = \mathtt{if}\ b\ \mathtt{then}\ C^{(i)}\ \mathtt{else}\ D^{(i)}$$
$$(C; D)^{(i)} = C^{(i)}; D^{(i)}$$
$$(\mathtt{while}\ b\ \mathtt{do}\ C)^{(i)} = (\mathtt{while}\ b\ \mathtt{do}\ C^{(i)})_i$$

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

LEMMA 5.7. *1. If $C \trianglelefteq D$ then $[\![C]\!] \subseteq [\![D]\!]$.*
2. For any command D:

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

We can now establish the converse of Lemma 5.5.

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

LEMMA 5.9. *1. For any proper non-empty pure transition sequence u, $(\sigma,\sigma')u \in [\![T, C]\!]^c$ holds 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, $(\sigma,\sigma')\mathrm{Done} \in [\![T, C]\!]^c$ holds iff $\langle\sigma, T, C\rangle \longrightarrow_a{}^* \langle\sigma', \varepsilon, \mathtt{skip}\rangle$.

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

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 certain nondeterministic choice operators to 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 $\mathcal{C}$, $\mathrm{obs}([\![\mathcal{C}[C]]\!]) = \mathrm{obs}([\![\mathcal{C}[D]]\!])$. In other words, two commands have the same meaning if and only if they lead to 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 $\mathcal{C}$, $\mathrm{obs}([\![\mathcal{C}[C]]\!]) \subseteq \mathrm{obs}([\![\mathcal{C}[D]]\!])$. 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.

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

Another possible candidate for $\mathrm{obs}(P)$ is $\mathrm{runs}(P)$. 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 $\mathrm{runs}([\![\mathcal{C}[C]]\!]) \subseteq \mathrm{runs}([\![\mathcal{C}[D]]\!])$ for every context $\mathcal{C}$, then $[\![C]\!]^c \subseteq [\![D]\!]^c$.*

We obtain the following *Full-abstraction Theorem*:

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

One direction of the theorem follows from Lemmas 5.13 and 5.14. The other is an immediate consequence of the compositionality of the semantics (Proposition 4.1).

Coarser-grained definitions of $\mathrm{obs}(P)$ may sometimes be appropriate. For those, full abstraction will typically require additional closure conditions on P, such as closure under stuttering and closure under mumbling, much as in our work and Brookes's on parallel composition [2, 10].

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 the functional programming approach to imperative languages, commands have unit type, 1. Then, taking the monadic point of view [6], 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 [15, 12, 17]. In the algebraic approach to computational effects [24, 17], one analyses the monads as free algebra monads T_L for a suitable equational or Lawvere theory L (meaning in the enriched sense, so that inequations are allowed, as are families of operations continuously parameterized over a cpo).

As discussed in [15], 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 [17]. It was shown there that resumptions arise by combining a theory for stores [24] 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 [10], 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 [15]. However, in [19], 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 [25] corresponding to the suspension operation. These identifications are justified by Theorem 6.3, 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.) In [10] Brookes imposed further closure conditions on his sets of traces, viz under stuttering and under mumbling, but these are not needed for our full-abstraction result.

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 clean 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 we define to be the sub-ω-cpo of Pool of all nonempty 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.

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 cpo. Finally, we analyze processes in Section 6.3, showing, in Theorem 6.3, 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 [17] 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 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 "Nat-ary" operations lookup_x ($x \in \mathrm{Vars}$, $n \in \mathrm{Nat}$). (A Nat-ary operation is a countably infinitary operation whose arguments are indexed by

elements of Nat.) 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 Nat-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.

The Lawvere theory L_H for nondeterminism is that of the lower (aka Hoare) powerdomain, presented using a binary 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 [24, 17] for more details of these theories, including an account of the equations for stores and for Hoare powerdomains.

For resumptions, continuing to follow [17], 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:

$$L_{\mathrm{Res}} = L_H \otimes ((L_S \otimes L_\Omega) + L_d)$$

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

We now give an elementary trace-based picture of $T_{\mathrm{Res}}(P)$ for moderately general 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 can show 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* in a partial order Q is a downwards-closed subset of Q; for any subset X of Q we write $X\!\downarrow$ for $\{x \in Q \mid \exists y \in X.\, x \leq y\}$, the least ideal including X; 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.

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:

$$\begin{array}{rcl} (\mathrm{update}_{\mathrm{Res}})_{x,n}(I) & = & \{(\sigma,\tau)u \mid (\sigma[x \mapsto n], \tau)u \in I\} \\ (\mathrm{lookup}_{\mathrm{Res}})_x(I_n)_n & = & \bigcup_n \{(\sigma,\sigma')u \in I_n \mid \sigma(x) = n\} \\ I \cup_{\mathrm{Res}} J & = & I \cup J \\ \Omega_{\mathrm{Res}} & = & \emptyset \\ \mathrm{d}_{\mathrm{Res}}(I) & = & \{(\sigma,\sigma)u \mid \sigma \in \mathrm{Store}, u \in I\} \\ & & \cup\, \{(\sigma,\sigma) | \sigma \in \mathrm{Store}\} \end{array}$$

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

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 $\eta : \mathcal{I}^{\uparrow}_{\omega}(Q) \to \mathcal{I}_{\omega}(Q\text{-BTrans})$ is given by:*

$$\eta(I) = \{(\sigma, \sigma\ x) \mid x \in I\}$$

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:

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

One can go further and obtain a closely related, if less elementary, picture of $T_{\mathrm{Res}}(P)$ for arbitrary 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 a cpo of resumptions, and, indeed, proper pure non-empty transition sequences and basic $\{\mathrm{Done}\}$-transition sequences are very similar. One can associate to every pure transition sequence $u(\sigma,\sigma')\mathrm{Done}$ (respectively non-empty pure transition sequence u not containing Done) the basic $\{\mathrm{Done}\}$-transition sequence $u(\sigma,\sigma'\ \mathrm{Done})$ (respectively u). Unfortunately, while the association is a bijection between proper pure non-empty transition sequences and basic $\{\mathrm{Done}\}$-transition sequences, it does not respect the order, since $u(\sigma,\sigma') \leq_p u(\sigma,\sigma')\mathrm{Done}$ 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:

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

Operationally, as in the proof of the full-abstraction theorem, one can distinguish $[\![\texttt{yield}; \texttt{block}]\!]$ from $[\![\texttt{skip}]\!]$ using a sequential context which, however, 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 with an equation:

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

accounting for the additional inequalities discussed above.

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

$$\begin{array}{rcl} (\mathrm{update}_{\mathrm{AProc}})_{x,n}(X) & = & \{(\sigma,\sigma')u \mid (\sigma[x \mapsto n], \sigma')u \in X\}\!\downarrow \\ (\mathrm{lookup}_{\mathrm{AProc}})_x(X)_n & = & (\bigcup_n \{(\sigma,\sigma')u \in X_n \mid \sigma(x) = n\})\!\downarrow \\ X \cup_{\mathrm{AProc}} Y & = & X \cup Y \\ \Omega_{\mathrm{AProc}} & = & \{\varepsilon\} \\ \mathrm{d}_{\mathrm{AProc}}(X) & = & \{(\sigma,\sigma)u \mid \sigma \in \mathrm{Store}, u \in X\}\!\downarrow \\ \mathrm{halt}_{\mathrm{AProc}} & = & \{(\sigma,\sigma)\mathrm{Done}\}\!\downarrow \end{array}$$

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

The next theorem shows that the variant theory L_{AProc} indeed captures AProc.

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

Here T_{AProc} is the free algebra functor associated to the theory AProc. It is not hard to 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.

6.3 Processes

We turn to our algebraic account of Proc. The signature is that for L_{Res} together with two families of unary operation symbols async_P and $\mathrm{yield\_to}_P$, where $P \in \mathrm{AProc}$. 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, $P \cdot t$ and $t \cdot P$.

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 $\cup$:

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

The second group of equations is for async:

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

The first three state that $P \cdot -$ commutes with another operation; the next concerns the interaction of async with suspension and brings in yield_to; the last reduces two async's to one. The third, and last, group of equations is for yield_to:

$$\begin{array}{rcl}x \cdot (\mathrm{update}_{\mathrm{AProc}})_{l,v}(P) & = & \mathrm{update}_{l,v}(x \cdot P) \\ x \cdot (\mathrm{lookup}_{\mathrm{AProc}})_l(P_n)_n & = & \mathrm{lookup}_l(x \cdot P_n)_n \\ x \cdot \Omega_{\mathrm{AProc}} & = & \Omega \\ x \cdot \mathrm{d}_{\mathrm{AProc}}(P) & = & \mathrm{d}(x \cdot P) \cup \mathrm{d}(P \cdot x) \\ x \cdot \mathrm{halt}_{\mathrm{AProc}} & = & \mathrm{d}(x)\end{array}$$

The first three assert that $x \cdot -$ acts homomorphically with respect to an operation; the next concerns the interaction with suspension; and the last concerns what happens when asynchronous threads halt. We take L_{Proc} to be $L_{\mathrm{Res}} + L_{\mathrm{Spawn}}$, i.e., the equations are the ones just given for async and yield_to, together with those for L_{Res}.

One might have expected to see an equation with left-hand side $P \cdot (x \cdot P')$; indeed, we could have added the equation:

$$P \cdot (x \cdot P') = (P \cdot x) \cdot P' \cup (x \cdot P) \cdot P'$$

However this equation is redundant, and can be proved from the others using the algebraic induction principle of "Computational Induction" described in [26]. (One proceeds by such an induction on P', with a subinduction on P.)

We now aim to give a picture of $T_{\mathrm{Proc}}(\mathcal{I}_\omega(Q))$ like that of $T_{\mathrm{Res}}(\mathcal{I}_\omega(Q))$. 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 $\mathcal{I}_\omega(Q)$. We view Q-Proc as a L_{Res}-model as in Section 6.1. In order to give async and yield_to we define, abusing notation, *left* and *right actions* of PSeq on Q-Trans:

$$\cdot : \mathrm{PSeq} \times Q\text{-Trans} \to Q\text{-Proc} \quad \cdot : Q\text{-Trans} \times \mathrm{PSeq} \to Q\text{-Proc}$$

The definitions are by mutual induction on the length of the sequences involved. For all $u \in \mathrm{PSeq}$ and $v \in Q$-Trans, we put:

$$\begin{array}{rcl}u[\mathrm{Done}] \cdot (\sigma, \sigma') & = & \{(\sigma, \sigma')u\}\downarrow \\ u \cdot (\sigma, \sigma'(x, v)) & = & \{(\sigma, \sigma'(x, w)) \mid w \in u \bowtie v\} \downarrow \\ u \cdot (\sigma, \sigma')v & = & \{(\sigma, \sigma')w \mid w \in u \cdot v \cup v \cdot u\} \downarrow\end{array}$$

where [Done] indicates an optional occurrence of Done, and:

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

where, in the last line, u is required to be proper. For $P \in \mathrm{AProc}$ and $I \in Q$-Proc, we put:

$$\begin{array}{rcl}(\mathrm{async}_{\mathrm{Proc}})_P(I) & = & \bigcup_{u \in P, v \in I} u \cdot v \\ (\mathrm{yield\_to}_{\mathrm{Proc}})_P(I) & = & \bigcup_{u \in P, v \in I} v \cdot u\end{array}$$

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

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. The cpo of processes Proc is the free model over 1; this places cooperative threads within the monadic approach to effects.

THEOREM 6.3. *Viewed as an L_{Proc}-model, $\mathcal{I}_\omega(Q\text{-Trans})$ is the free model over $\mathcal{I}^{\uparrow}_\omega(Q)$. The unit $\eta : \mathcal{I}^{\uparrow}_\omega(Q) \to \mathcal{I}_\omega(Q\text{-Trans})$ is given by:*

$$\eta(I) = \{(\sigma, \sigma\ (x, \mathrm{Done})) \mid x \in I\}$$

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

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

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.

We are finally in a position to give our algebraic account of Proc. There is an isomorphism $\theta : Q\text{-Trans} \to \mathrm{TSeq} \setminus \{\varepsilon\}$, where $Q = \{\mathrm{Ret}\}$, sending $u = (\sigma_1, \sigma_1') \ldots (\sigma_n, \sigma_n')$ to itself and $u(\sigma, \sigma'\ (\mathrm{Ret}, v))$ to $u(\sigma, \sigma'\ \mathrm{Ret})v$. One then has an isomorphism $\mathcal{I}_\omega(Q\text{-Trans}) \cong \mathrm{Proc}$, given by: $I \mapsto \theta(I) \cup \{\varepsilon\}$. So Proc can be seen as the free model of L_{Proc} on $\{\mathrm{Ret}\}$.

This algebra determines the semantics of our language, and, in that sense, justifies the previous, more ad hoc, account. First, we have $[\![\texttt{skip}]\!] = \eta(\{\mathrm{Ret}\})$ and $I \circ J = (\{\mathrm{Ret}\} \mapsto J)^\dagger(I)$, so the Kleisli structure determines the semantics of `skip` and composition, 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 assignment and conditionals; the d operation is that of the algebra; and `block` is modeled by Ω. The semantics of spawning is determined by async together with the function $-^c : \mathrm{Proc} \to \mathrm{AProc}$, and it turns out that the latter is also determined by algebraic means. Specifically, one can regard AProc as a model of L_{Proc}, setting:

$$\mathrm{async}_P(Q) = \{(\sigma, \sigma')w \mid \exists u \in P, (\sigma, \sigma')v \in Q.w \in u \bowtie v\} \downarrow$$

and $\mathrm{yield\_to}_P(Q) = \mathrm{async}_Q(P)$, and then $-^c$ is the extension of $\mathrm{Ret} \mapsto \mathrm{halt}_{\mathrm{AProc}}$ 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:

$$\langle \sigma, T, \mathcal{E}[\texttt{yield\_to}\ C] \rangle \quad \longrightarrow \quad \langle \sigma, T.\mathcal{E}[\texttt{skip}], C \rangle$$

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 $P \cdot \mathrm{d}(x) = \mathrm{d}(P \cdot x) \cup \mathrm{d}(x \cdot P)$ becomes:

$$(\texttt{async}\ C); \texttt{yield}; D$$
$$=$$
$$(\texttt{yield}; (\texttt{async}\ C); D)\ \texttt{or}\ (\texttt{yield}; (\texttt{yield\_to}\ C); D)$$

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. [19]. 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.

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 stuttering and under mumbling. 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` [14] and, in the semantics, treat blocking as a kind of exception. Finally, we could revisit lower-level semantics with explicit optimistic concurrency and rollbacks, of the kind employed in the implementation of AME.

Acknowledgments

We are grateful to Martín Escardó and Martin Hyland for comments on this work.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–74, 2008.

[2] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, June 1993.

[3] Karl Abrahamson. Modal logic of concurrent nondeterministic programs. In Gilles Kahn, editor, *International Symposium on Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 1979.

[4] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track: the 2002 USENIX Annual Technical Conference*, pages 289–302, 2002.

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

[6] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Advanced Lectures from International Summer School on Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 42–122. Springer, 2002.

[7] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–129, 1992.

[8] Gérard Boudol. Fair cooperative multithreading. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Concurrency Theory, 18th International Conference*, volume 4703 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.

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

[10] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, June 1996.

[11] Stephen Brookes. The essence of parallel Algol. *Information and Computation*, 179(1):118–149, 2002.

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

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

[14] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[15] Matthew Hennessy and Gordon D. Plotkin. Full abstraction for a simple programming language. In J. Bečvář, editor, 8^{th} *Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer, 1979.

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

[17] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1–3):70–99, 2006.

[18] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX workshop on Hot Topics in Operating Systems*, pages 1–6, 2007.

[19] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 255–264, 1995.

[20] Alan Jeffrey. Semantics for core Concurrent ML using computation types. In *Higher order operational techniques in semantics*, pages 55–90. Cambridge University Press, 1997.

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

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

[23] Prakash Panangaden and John H. Reppy. The essence of Concurrent ML. In Flemming Nielson, editor, *ML with Concurrency*, chapter 1, pages 5–29. Springer, 1997.

[24] Gordon Plotkin and John Power. Notions of computation determine monads. *Lecture Notes in Computer Science*, 2303:373–393, 2002.

[25] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[26] Gordon D. Plotkin and Matija Pretnar. A logic for algebraic effects. In *Proceedings, Twenty-Third Annual IEEE Symposium on Logic in Computer Science*, pages 118–129, 2008.

[27] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[28] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 191–210, 2007.

[29] J. Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric A. Brewer. Capriccio: scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, 2003.

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

Appendix: Proofs

This appendix gives some proofs of interest, to the extent that space permits.

We begin with the proof of two main lemmas used in the proof of full abstraction. 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). We also define commands:

- $\mathrm{goto}(\sigma)$ as the sequence of assignments $x := n$ for every variable x, where n is the natural number $\sigma(x)$;
- $(\sigma \rightsquigarrow \sigma')$ as `if` $\mathrm{check}(\sigma)$ `then` $\mathrm{goto}(\sigma')$ `else block`;
- $(\sigma \rightsquigarrow \sigma' \rightsquigarrow \sigma'')$ as $(\sigma \rightsquigarrow \sigma')$; `yield`; $(\sigma' \rightsquigarrow \sigma'')$; `yield`.

Proof of Lemma 5.13: 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 $u(\sigma, \sigma'\ \mathrm{Ret})v$. 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 $u(\sigma, \sigma''\ \mathrm{Ret})v$, 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:

- $w' = u(\sigma, \sigma''\ \mathrm{Ret})v$, with $w \in Q$. This contradicts $w \notin Q$.
- $w' = u'(\sigma, \sigma'')v'$, where u' and v' are of the same length as u and v, respectively, and σ'' occurs marked with Ret 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 Ret marker. This contradicts the requirement that $u(\sigma, \sigma'') \notin Q$. ∎

Proof of Lemma 5.14: Letting $P = [\![C]\!]$ and $Q = [\![D]\!]$, we assume that $P^c \not\subseteq Q^c$ and prove that there exists $\mathcal{C}$ such that $\mathrm{runs}([\![\mathcal{C}]\!](P)) \not\subseteq \mathrm{runs}([\![\mathcal{C}]\!](Q))$. For this, choose a sequence w in P^c but not in Q^c.

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` $[\ ]; \mathrm{mesh}(w)$, where $\mathrm{mesh}(w)$ is the command

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

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 $[\![\mathrm{mesh}(w)]\!]$ contains the transition sequence:

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

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

$$\begin{array}{l}(\sigma_1, \sigma_1)(\sigma_1, \sigma_1')(\sigma_1', \sigma_1'')(\sigma_1'', \sigma_2)(\sigma_2, \sigma_2') \ldots (\sigma_{n-1}'', \sigma_n)\\ (\sigma_n, \sigma_n')(\sigma_n', \sigma_n''\ \mathrm{Ret})\end{array}$$

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 $\mathrm{runs}([\![\mathcal{C}]\!](Q))$. Therefore, there exists $w' \in Q^c$ such that

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

is a shuffle of w' with

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

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 as $(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')$. 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 the next possible transition in the shuffle, namely $(\sigma_{i-1}', \sigma_{i-1}'')$. This transition cannot come from w' because, by the choice of σ_{i-1}'', we have that
    $$(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')(\sigma_{i-1}', \sigma_{i-1}'') \notin Q^c$$
    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 $(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')(\sigma_{i-1}'', \sigma_i)$, and in fact $(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')(\sigma_{i-1}'', \sigma_i)(\sigma_i, \sigma_i')$, 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
    $$(\sigma_1, \sigma_1') \ldots (\sigma_{i-1}, \sigma_{i-1}')(\sigma_{i-1}'', \sigma_i)(\sigma_i, \sigma_i')(\sigma_i', \sigma_i'') \notin Q^c$$
    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 $(\sigma_1, \sigma'_1) \dots (\sigma_n, \sigma'_n)\mathrm{Done}$. With the same $\mathcal{C}$, we obtain that $[\![\mathcal{C}]\!](P)$ contains:

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

which generates the run $\sigma_1\sigma_1\sigma'_1\sigma''_1\sigma_2\sigma'_2 \dots \sigma''_{n-1}\sigma_n\sigma'_n\sigma''_n\mathrm{Done}$. Suppose that this run is also in $\mathrm{runs}([\![\mathcal{C}]\!](Q))$. Again, by the choice of σ''_1, …, σ''_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 continue with proofs of algebraic characterization theorems. We write $\omega\mathrm{Cpo}$, $\omega\mathrm{SL}$, and $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ for, respectively, the categories of ω-cpos, ω-cpos with all finite sups, and the Kleisli category of $\mathcal{I}_\omega$. The latter two have countable biproducts, given by cartesian product in the first case, and the sum of posets in the second; in particular, the copower $\mathrm{Store} \times P$ is the usual cartesian product of posets and we identify P^{Store} with $\mathrm{Store} \times P$.

LEMMA 7.1. *Suppose that R carries the free structure over P in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ which is a model of L_S in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ and which has a map $d_R : R_\perp \to R$. Then $\mathcal{I}_\omega(R)$ carries the free structure over $\mathcal{I}_\omega(P)$ in $\omega\mathrm{SL}$ which is a model of L_S in $\omega\mathrm{SL}$, and which has a map $\mathrm{d} : M_\perp \to M$ and:*

$$\begin{aligned}(\mathrm{update}_{\mathcal{I}_\omega(R)})_{x,n}(I) &= \textstyle\bigcup_{u\in I}(\mathrm{update}_R)_{x,n}(u)\\ (\mathrm{lookup}_{\mathcal{I}_\omega(R)})_l(f) &= \textstyle\bigcup_{x\in\mathrm{Nat}, u\in f(x)}(\mathrm{lookup}_R)_l(v,x)\\ \mathrm{d}_{\mathcal{I}_\omega(R)}(I) &= \textstyle\bigcup_{u\in I} d_R(u) \cup \mathrm{d}_R(\perp)\end{aligned}$$

Proof of Theorem 6.1: By Corollary 2 of [17], the free model

$$(R, \mathrm{update}_R, \mathrm{lookup}_R)$$

of L_S in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ together with a morphism $R_\perp \to R$ over a poset Q has carrier the solution of the following domain equation in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$:

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

by which we mean the initial such R, and where we abbreviate Stores to S. Since countable copowers and powers coincide in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$, this can be rewritten as:

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

As the left adjoint $\mathrm{Pos} \to \omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ preserves all colimits we can solve this domain equation by solving it instead in Pos and that can be done by taking R to be the least set such that

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

and then imposing the evident inductively defined partial order. It is not hard to see that $R_\perp$ is Q-BTrans. The map $\mathrm{d}_R : R_\perp \to R$ in $\omega\mathrm{Cpo}_{\mathcal{I}_\omega}$ is:

$$R_\perp \xrightarrow{\mathrm{inl}} R_\perp + Q \xrightarrow{\eta_{R_\perp + Q}} (S \times (R_\perp + Q))^S = R$$

and is given by:

$$\mathrm{d}_R(x) = \{((\sigma,\sigma), \mathrm{inl}(x)) \mid \sigma \in S\} \downarrow$$

The map $(\mathrm{update}_R)_{l,v} : R \to R$ is:

$$R = T_S(R_\perp + Q) \xrightarrow{\mathrm{update}_{R_\perp + Q}} T_S(R_\perp + Q) = R$$

and is given by:

$$(\mathrm{update}_R)_{l,v}((\sigma,\sigma), x) = ((\sigma, \sigma[l \mapsto v]), x) \downarrow$$

Similarly $(\mathrm{lookup}_R)_l : R^V \to R$ is given by:

$$(\mathrm{lookup}_R)_l(v, ((\sigma,\sigma), x)) = \{((\sigma,\sigma), x) \mid \sigma(l) = v\} \downarrow$$

By Lemma 7.1 $\mathcal{I}_\omega(R)$ then carries the free structure M over $\mathcal{I}_\omega(Q)$ in $\omega\mathrm{SL}$ which is a model of L_S in $\omega\mathrm{SL}$ and which has a map $M_\perp \to M$. By Theorem 8 of [17], to have a model of L_{Res} is to have such a structure, so $\mathcal{I}_\omega(R)$ carries the free model of L_{Res} over $\mathcal{I}_\omega(Q)$.

There is an evident isomorphism Q-$\mathrm{BTrans} \cong R$, so the free such model is also carried by $\mathcal{I}_\omega(Q\text{-}\mathrm{BTrans})$. Using the above and Lemma 7.1 one then verifies that the operations are as required. For the formula for the Kleisli extension, that $f^\dagger \eta = f$ is evident and that the purported extension is a morphism of models of L_{Res} is a calculation. ∎

Proof of Theorem 6.2: To have a model of L_{APROC} in $\omega\mathrm{Cpo}$ is to have a model M of L_S in $\omega\mathrm{SL}$ together with a map $\mathrm{d} : M_\perp \to M$ and an element $\mathrm{halt} \in M$ such that $\mathrm{d}(\Omega) \leq \mathrm{halt}$. Further, to have such a map and element is to have a map $(M+1)_\perp \to M$. With these observations the proof proceeds analogously to that of Theorem 6.1. ∎

Proof of Theorem 6.3: To show that $\mathcal{I}_\omega(Q\text{-}\mathrm{Trans})$ is the free algebra over $\mathcal{I}^\dagger_\omega(Q)$ with unit η as above, we must show that for any L_{Proc}-model A and any continuous function $f : \mathcal{I}^\dagger_\omega(Q) \to A$ there is a unique morphism $h : \mathcal{I}_\omega(Q\text{-}\mathrm{Trans}) \to A$ of models of L_{Proc} such that $h\eta = f$.

We begin by showing uniqueness. To this end, suppose we are given such an A, f, and morphism h such that $h\eta = f$. Then for uniqueness it suffices to prove that $h\eta_{T_{\mathrm{Res}}} = g$ where $g((x,u)\downarrow) = u \cdot_A f(x)$, as h is a morphism of models of L_{Proc}. We calculate:

$$\begin{aligned}h(\eta_{T_{\mathrm{Res}}}((q,u)\downarrow)) &= h(\{(\sigma, \sigma(q,u)) \mid \sigma \in \mathrm{Stores}\})\\ &= h(u\downarrow \cdot \{(\sigma, \sigma(q, \mathrm{Done})) \mid \sigma \in \mathrm{Stores}\})\\ &= h(u\downarrow \cdot \eta(q\downarrow))\\ &= u\downarrow \cdot h(\eta(q\downarrow))\\ &= u\downarrow \cdot_A f(q\downarrow)\end{aligned}$$

For existence we are again given A and f and wish to construct a suitable h. To this end, with g as before, take h to be the T_{Res}-extension of g. Then $h\eta = f$ and it remains to prove that h preserves async and yield_to. It suffices to prove this for individual transition sequences, by induction invoking L_{Proc} equations on A as necessary. The formula for the Kleisli extension follows using the Kleisli formula of Theorem 6.1. ∎