A Monadic Account of First-class Synchronous Events

Matthew Fluet
Cornell University
fluet@cs.cornell.edu

January 20, 2006

Abstract

1 Introduction

“A value of type `IO a` is an ‘action’ that, when performed, may do some input/output, before delivering a value of type `a`.” [13, (Peyton Jones, *Tackling the Awkward Squad*)]

“The type τ `event` is the type of a synchronous operation that returns a value of type τ when it is synchronized upon.” [16, (Reppy, *Concurrent Programming in ML*)]

These two quotations represent the key ideas behind two major research agendas. The first captures the essence of *monadic I/O*, which is the fundamental abstraction used to provide input/output and concurrency in the lazy, purely-functional language (Concurrent) Haskell. The second captures the essence of *first-class synchronous events*, which is the fundamental abstraction used to provide concurrency in the strict, mostly-functional language Concurrent ML. While there are many superficial connections between Concurrent Haskell and Concurrent ML, the striking parallels in these descriptions of their fundamental abstractions begs further investigation. The question that naturally arises is the following: “Does the `event` type constructor of Concurrent ML form a *monad*?” Having asked that question, we are immediately led to another, related question: “Since non-deterministic choice is an event combinator, does the `event` type constructor of Concurrent ML form a *monad-with-plus*?”

Unfortunately, it is easy to demonstrate that the most natural means of defining the monad-with-plus operations in terms of CML event combinators fail to satisfy the necessary monad laws.

In this work, we review the monad laws and the reasons that CML events do not naturally form a monad-with-plus. This investigation reveals that the essential missing component is an event combinator that combines a sequence of events into a single event and whose semantics ensure an “all-or-nothing” property – either all of the constituent events synchronize in sequence or none of them synchronize.

We propose a new concurrency calculus that draws inspiration from both Concurrent ML and Concurrent Haskell. Like Concurrent ML, it includes first-class synchronous events and event combinators (including the new sequencing combinator), forming a monad-with-plus. Like Concurrent Haskell, it includes first-class I/O actions and I/O combinators, forming a monad.

The essence of the dynamics is to embrace the interpretation of monad as a notion of computation. Hence, the two monads (Evt and IO), yield two sorts of computations. This is expressed by three levels of evaluation: sequential evaluation of pure terms, synchronous evaluation of synchronizing events (e.g., the Evt monad), and concurrent evaluation of concurrent threads (e.g., the IO monad). The bridge between the Evt and IO monads is synchronization, which moves events and threads from concurrent evaluation to synchronous evaluation and back to concurrent evaluation.

```
type thread_id

val spawn : (unit -> unit) -> thread_id

type 'a chan

val channel : unit -> 'a chan
val recv    : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit

type 'a event

val recvEvt   : 'a chan -> 'a event
val sendEvt   : ('a chan * 'a) -> 'a event

val wrap      : ('a event * ('a -> 'b)) -> 'b event
val guard     : (unit -> 'a event) -> 'a event
val withNack  : (unit event -> 'a event) -> 'a event
val choose    : 'a event list -> 'a event

val never     : 'a event
val alwaysEvt : 'a -> 'a event

val sync   : 'a event -> 'a
```

Figure 1: The CML Interface

This concurrency calculus appears to be quite expressive (perhaps too expressive). We show that general Concurrent ML programs, with arbitrary `wrap` and `guard` and `withNack` computations, may be seen as a stylized use of the new concurrency calculus. This encoding sheds further light on the reasons that CML events do not form a monad-with-plus. We also show that the new concurrency calculus is strictly more expressive than Concurrent ML: one *can* abstractly implement an n-way rendezvous operation (i.e., as an event value), given the 2-way channel rendezvous operations.

We conclude with preliminary implementation considerations.

2 Background

In this section, we review both Concurrent ML and Concurrent Haskell. Elements of each of these languages contribute to the design of the concurrency calculus presented in Section 5.

2.1 Concurrent ML (CML)

Concurrent ML (CML) extends Standard ML with concurrency primitives [15, 16]. The main novelty of CML lies in the design decision to make synchronous operations into first-class values. These values, called *events*, represent *potential* communication and synchronization actions. Event values are themselves quiescent; their communication action is only performed when a thread uses them for synchronization. By providing synchronous events as first-class values, CML acheives a level of abstraction and modularity in concurrent programs that exceeds that found in other concurrent languages.

In this section, we give an informal overview of the central CML constructs. Figure 1 gives the signature of the essential CML primitives.

The type `thread_id` is the (abstract) type of thread IDs; although unused in this interface, thread IDs are in an unspecified total order that may be used to break cyclic dependencies. The type `'a chan` is the (abstract) type of channels on which values of type `'a` may be communicated.

The first four of operations are fairly standard:

`spawn` *f* creates a new thread to evaluate the function *f* and returns the ID of the newly created thread.

`channel ()` creates a new synchronous channel

`recv` *ch* receives a value on the channel *ch*. This operation blocks the calling thread until there is another thread sending a message on *ch*.

`send (`*ch*`,`*msg*`)` sends the value *msg* on the channel *ch*. This operation blocks the calling thread until there is another thread receiving a message on *ch*.

The remaining type and functions support the event abstraction. The type `'a event` is the (abstract) type of synchronous operations. Thinking of `'a event` as an abstract datatype, we see that the interface includes base event functions (`recvEvt`, `sendEvt`, `never`, `alwaysEvt`) as introduction forms, and synchronization functions (`sync`) as elimination forms. There are additional combinators (`wrap`, `guard`, `withNack`, `choose`) for constructing complex events out of constituent events.

`recvEvt` *ch* returns an event value that represents the operation of receiving a value on the channel *ch*. When this event is synchronized on, it receives a value on the channel and returns it as the result of synchronization.

`sendEvt (`*ch*`,`*msg*`)` returns an event value that represents the operation of sending the value *msg* on the channel *ch*. When this event is synchronized on, it sends the value on the channel and returns `()` as the result of synchronization.

`wrap (`*ev*`,`*f*`)` returns an event value that wraps the event value *ev* with the post-synchronous action *f* (a *wrapper function*). When this event value is synchronized on, the function *f* is applied to the synchronization result of *ev*.

`guard` *f* returns a *delayed* event value that creates an event value out of the pre-synchronous action *f* (a *guard function*). When this event is synchronized on, the function *f* is applied to `()` and its result is used in the synchronization.

`withNack` *f* returns a *delayed* event value that creates an event value out of the pre-synchronous action *f*. When this event is synchronized on, the function *f* is applied to a *negative acknowledgement* event and its result is used in the synchronization. The negative acknowledgement event is enabled if some event other than the one produced by *f* is chosen for the synchronization result.

`choose [`ev_1`,`$\ldots$`,`ev_n`]` returns an event value that represents the non-deterministic choice of the event values ev_1,. . .,ev_n. When this event is synchronized on, one of the events ev_i will be chosen and synchronized on, and its result will be the synchronization result.

`never` returns an event value that is never enabled. This event may not be chosen and synchronized on for the synchronization result.

`alwaysEvt` *v* returns an event value that is always enabled. When this event is synchronized on, it returns *v* as the result of synchronization.

`sync` *ev* sychronizes on the event value *ev* and returns the synchronization result.

As a very simple example, we consider implementations of an event value that represents communication between a client thread and two server threads. If a client wishes to issue requests to both servers and interact with whichever server accepts the request first, then the client constructs the following event:

```
choose [
    wrap (sendEvt (req1, serverCh1), fn () => recv (replyCh1)),
    wrap (sendEvt (req2, serverCh2), fn () => recv (replyCh2))
  ]
```

If, on the other hand, the client wishes to issue requests to both servers and interact with whichever server honors the request first, then the client constructs the following event:

```
choose [
    guard (fn () => (send (req1, serverCh1); recvEvt (replyCh1))),
    guard (fn () => (send (req2, serverCh2); recvEvt (replyCh2)))
  ]
```

Of course, there is no *a priori* requirement that the client make use of different communication patterns with the two servers:

```
choose [
    guard (fn () => (send (req1, serverCh1); recvEvt (replyCh1))),
    wrap (sendEvt (req2, serverCh2), fn () => recv (replyCh2))
  ]
```

The first-class nature of event values is precisely what enables these communication patterns to be exported abstractly by the server interfaces (say, as `protocol1` and `protocol2`); in this case, the client constructs the following event:

```
choose [protocol1 req1, protocol2 req2]
```

In general, one often wants to implement a protocol consisting of a sequence of communications: $c_1 ; c_2 ; \cdots ; c_n$. When this protocol is used in a selective communication, one of the c_i is designated as the *commit point*, the communication by which this protocol is chosen over the others in a select. The entire protocol may be packaged as an event value by using the `guard` constructor to prefix the communications $c_1 ; \cdots ; c_{i-1}$ and using the `wrap` constructor to postfix the communications $c_{i+1} ; \cdots ; c_n$. Note that the semantics of CML require that all of the pre-synchronous communications must synchronize, in order for `guard` to yield the commit point communication; likewise, all of the post-synchronous communications must synchronize, in order for `wrap` to yield the synchronization result.

2.2 Concurrent Haskell

Concurrent Haskell extends Haskell with concurrency primtives [14, 13]. The main novelty of Concurrent Haskell lies in the use of monadic I/O as a means of integrating side-effecting operations into a lazy, purely-functional language. Practically, the essence of monadic I/O is the construction of *I/O actions* that, when performed, may do some I/O before yielding a value. Theoretically, the essence of monadic I/O is based on the algebraic structure of monads [10, 18], which we review in more detail in the next section.

In this section, we give an informal overview of the central Concurrent Haskell constructs. Figure 2 gives the signature of the essential Concurrent Haskell primitives.

The type `IO a` is the (abstract) type of I/O actions. Thinking of `IO a` as an abstract datatype, we see that the interface includes base actions (`getChar`, `putChar`, `newEmptyMVar`, `takeMVar`, `putMVar`) as introduction forms and combinators (`>>=`, `return`) for constructing complex actions out of constituent actions.

`getChar` constructs an I/O action that represents the operation of reading a character from standard input. When this action is performed, it returns the character read.

`putChar` c constructs an I/O action that represents the operation of writing the character c to standard output. When this action is performed, it returns `()`.

a `>>=` f constructs an I/O action that performs the I/O action a (returning e) followed by the I/O action $f\ e$ (returning e') and returns e'.

```
data IO a

getChar :: IO Char
putChar :: Char -> IO ()

(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a

data ThreadId

forkIO :: IO () -> IO ThreadId

data MVar a

newEmptyMVar  :: IO (MVar a)
takeMVar      :: MVar a -> IO a
putMVar       :: MVar a -> a -> IO a
```

Figure 2: The Concurrent Haskell Interface

`return` *e* constructs an I/O action that performs no side-effects and returns *e*.

The type `ThreadId` is the (abstract) type of thread IDs; although unused in this interface, thread IDs may be used to interact with an executing thread.

`forkIO` *a* constructs an I/O action that represents the operation of creating a new therad to perform the I/O action *a*. When this action is performed, it returns the ID of the newly created thread.

The type `MVar a` is the type of a shared, mutable variable, which may be either empty or full.

`newEmptyMVar` creates a new, empty mvar.

`takeMVar` *mv* blocks until the mvar *mv* is full, then reads and returns the value, leaving the mvar empty.

`putMVar` *mv v* blocks until the mvar *mv* is empty, then writes the value *v*, leaving the mvar full.

In a Concurrent Haskell program, there must be a distinguished value `main` with the type `IO ()`. The execution of a Concurrent Haskell program performs the I/O action denoted by `main`, which may include creating new threads to perform their own actions. In a sense, execution of the program represents the elimination form of the `IO` abstraction.

In the sequel, we will not require Concurrent Haskell's `MVar`s, though it is worth noting that their semantics may be encoded in Concurrent ML [16, Section 5.3.1].

3 Monads and Monad Laws

Monads, a particular structure that arises in Category Theory, were introduced into programming languages research as a way of modeling *computation types* [10]. In this setting, we distinguish between objects corresponding to values of type τ and objects corresponding to computations yielding values of type τ. The latter can themselves be considered as values of type $\mathsf{M}\ \tau$, where M is a unary type-constructor that abstracts a particular notion of computation and computational effect. Monads have a rich mathematical structure and can serve as a useful foundation for structuring denotational semantics, but we will focus on their practical application as a mechanism for representing computational effects in a programming language [18].

The representation of monads in a programming language is based on the *Kleisli triple* formulation of monads; essentially, each monad (notion of computation) M is given as a unary type-constructor, accompanied by two operations:

Signature for *Monad*

$$\begin{array}{rcl} \mathsf{M} & :: & \star \to \star \\ \mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\ \mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \end{array}$$

The first operation yields the trivial computation that simply delivers its argument without any effect whatsoever; the second operation composes two computations in sequence, feeding the result of the first into the second. These two operations must obey the *monad laws* [18]:

Laws for *Monad*

$$\begin{array}{rcl} \mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \simeq & f\ a \\ \mathsf{mbind}\ m\ \mathsf{munit} & \simeq & m \\ \mathsf{mbind}\ (\mathsf{mbind}\ m\ f)\ g & \simeq & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mbind}\ (f\ x)\ g) \end{array}$$

We may interpret these laws as requiring mbind to be an associative binary operation with munit as a left and right identity. (The binding in the second argument of mbind distinguishes a *monad* from a *monoid*.)

The monad laws are formulated in terms of an equivalence $\simeq$ between expressions. We will be somewhat informal about the nature of this equivalence, but when considering Concurrent ML and the concurrency calculus of Section 5, we will mainly be interested in observational equivalence, where the notion of observation includes the (infinite) stream of input and output actions of the program.

Implementing a monad in a programming language requires choosing a particular representation for M and implementing the operations munit and mbind. Additionally, each monad can be accompanied by operations specific to the particular computational effect under consideration. For example, the monad of mutable integer state has operations to read and write the state and could be implemented thusly:

$$\begin{array}{rcl} \mathsf{M}\ \alpha & = & \mathsf{Int} \to (\mathsf{Int} \times \alpha) \\ \\ \mathsf{munit} & = & \lambda x.\ \lambda s.\ (s, x) \\ \mathsf{mbind} & = & \lambda m.\ \lambda f.\ \lambda s.\ \mathsf{let}\ (s',\ a)\ =\ m\ s\ \mathsf{in} \\ & & \qquad\qquad\ \ \mathsf{let}\ m'\ =\ f\ a\ \mathsf{in} \\ & & \qquad\qquad\ \ m'\ s' \\ \\ \mathsf{readState} & :: & \mathsf{M}\ \mathsf{Int} \\ \mathsf{readState} & = & \lambda s.\ (s, s) \\ \mathsf{writeState} & :: & \mathsf{Int} \to \mathsf{M}\ () \\ \mathsf{writeState} & = & \lambda i.\ \lambda s.\ (i, ()) \end{array}$$

A programming language may also provide a monad as a primitive abstraction; this is the case with the `IO` monad in Haskell. In the absence of concurrency, we may consider the `IO` monad as a special state monad, where the manipulated state corresponds to the "state" of the real world. By making the "state of the real world" abstract and ensuring that each of the special operations treat the state in a proper single-threaded manner, a Haskell implementation can implement the `IO` monad with destructive updates. The monad discipline ensures that this implementation preserves the semantics of the state-passing style.

This analogy breaks down when extended to concurrency (since the "state of the real world" must be passed among the concurrently executing threads), so Concurrent Haskell defines the behavior of the `IO` monad by adopting an operational semantics. This is the approach we take in Section 5.

Note that for each monad, we can define the following operations:

Derived Functions for *Monad*

$$\begin{array}{rcl}
\mathsf{mmap} & :: & (\alpha \to \beta) \to (\mathsf{M}\ \alpha \to \mathsf{M}\ \beta) \\
\mathsf{mmap}\ f\ m & = & \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{munit}\ (f\ x)) \\
\mathsf{mjoin} & :: & \mathsf{M}\ (\mathsf{M}\ \alpha) \to \mathsf{M}\ \alpha \\
\mathsf{mjoin}\ m & = & \mathsf{mbind}\ m\ \mathsf{id}
\end{array}$$

The first operation simply applies a function to the result of a computation. The second operation flattens a "double" computation (i.e., a computation yielding a computation) into a "simple" computation.

In fact, monads are often defined in terms of munit, mmap, and mjoin, rather than in terms of munit and mbind. This leads to an alternative signature:

Alternative Signature for *Monad*

$$\begin{array}{rcl}
\mathsf{M} & :: & \star \to \star \\
\mathsf{munit} & :: & \alpha \to \mathsf{M}\ \alpha \\
\mathsf{mmap} & :: & (\alpha \to \beta) \to (\mathsf{M}\ \alpha \to \mathsf{M}\ \beta) \\
\mathsf{mjoin} & :: & \mathsf{M}\ (\mathsf{M}\ \alpha) \to \mathsf{M}\ \alpha
\end{array}$$

alternative laws:

Alternative Laws for *Monad*

$$\begin{array}{rcl}
\mathsf{mmap}\ \mathsf{id} & \simeq & \mathsf{id} \\
\mathsf{mmap}\ (f \circ g) & \simeq & (\mathsf{mmap}\ f) \circ (\mathsf{mmap}\ g) \\
\mathsf{mmap}\ (f \circ \mathsf{munit}) & \simeq & \mathsf{munit} \circ f \\
\mathsf{mmap}\ (f \circ \mathsf{mjoin}) & \simeq & \mathsf{mjoin} \circ (\mathsf{mmap}\ (\mathsf{mmap}\ f)) \\
\mathsf{mjoin} \circ \mathsf{munit} & \simeq & \mathsf{id} \\
\mathsf{mjoin} \circ (\mathsf{mmap}\ \mathsf{munit}) & \simeq & \mathsf{id} \\
\mathsf{mjoin} \circ (\mathsf{mmap}\ \mathsf{mjoin}) & \simeq & \mathsf{mjoin} \circ \mathsf{mjoin}
\end{array}$$

and alternative defined functions:

Alternative Derived Functions for *Monad*

$$\begin{array}{rcl}
\mathsf{mbind} & :: & \mathsf{M}\ \alpha \to (\alpha \to \mathsf{M}\ \beta) \to \mathsf{M}\ \beta \\
\mathsf{mbind}\ m\ f & = & \mathsf{mjoin}\ (\mathsf{mmap}\ f\ m)
\end{array}$$

The two definitions are equivalent; we provide both to facilitate comparison.

We may readily see that mbind (and mjoin) correspond to a notion of sequencing computation. Another fundamental notion of computation is that of alternation or choice. A monad-with-plus augments the monad operations with two additional operations:

Signature for *MonadPlus*

$$\begin{array}{rcl}
\mathsf{mzero} & :: & \mathsf{M}\ \alpha \\
\mathsf{mplus} & :: & \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha \to \mathsf{M}\ \alpha
\end{array}$$

The first operation is the trivial computation that always fails (without delivering any result); the second operation introduces a choice between alternatives.

There is some debate regarding the precise set of laws that a monad-with-plus should satisfy [11], but there is reasonable agreement on the following *monad-with-plus laws* [18, 3, 7]:

Laws for *MonadPlus*

$$\begin{array}{rcl}
\mathsf{mplus}\ m\ \mathsf{mzero} & \simeq & m \\
\mathsf{mplus}\ \mathsf{mzero}\ m & \simeq & m \\
\mathsf{mplus}\ m_1\ (\mathsf{mplus}\ m_2\ m_3) & \simeq & \mathsf{mplus}\ (\mathsf{mplus}\ m_1\ m_2)\ m_3 \\
\mathsf{mbind}\ \mathsf{mzero}\ f & \simeq & \mathsf{mzero} \\
\mathsf{mbind}\ (\mathsf{mplus}\ m_1\ m_2)\ f & \simeq & \mathsf{mplus}\ (\mathsf{mbind}\ m_1\ f)\ (\mathsf{mbind}\ m_2\ f)
\end{array}$$

The first three laws require that mplus is an associative binary operation with mzero as a left and right identity. The last two laws require that mzero is a left zero for mbind and that mbind distributes over mplus (to the left).

There are additional monad-with-plus laws that will prove useful in the sequel [11]:

Additional Laws for *MonadPlus*

$$\begin{array}{rcl}
\mathsf{mplus}\ m_1\ m_2 & \simeq & \mathsf{mplus}\ m_2\ m_1 \\
\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mzero}) & \simeq & \mathsf{mzero} \\
\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mplus}\ (f\ x)\ (g\ x)) & \simeq & \mathsf{mplus}\ (\mathsf{mbind}\ m\ f)\ (\mathsf{mbind}\ m\ g)
\end{array}$$

The first law requires that mplus is a commutative binary operation, which emphasizes mplus as a *non-determinisitic* choice. The last two laws require that mzero is a right zero for mbind and that mbind distribute over mplus (to the right).

4 CML Revisited

We are now in a position to investigate our initial question: “Does the `event` type constructor of Concurrent ML form a *monad-with-plus*?” To answer this question, we must first instantiate the monad-with-plus operations with the Concurrent ML event combinators. In doing so, we may be guided by the types of the operations and the combinators (being somewhat liberal with currying and the order of arguments). This yields the following instantiations:

MonadPlus	CML
M	`event`
munit	`alwaysEvt`
mbind	???
mmap	`fn f => fn ev => wrap (ev, f)`
mjoin	???
mzero	`never`
mplus	`fn ev1 => fn ev2 => choose [ev1, ev2]`

As we have an obvious instantiation for mmap, we are motivated to look for an instantiation for mjoin, to complete the alternative signature for monad. Considering the remaining Concurrent ML combinators, we see that sync may be instantiated to have the same type as mjoin; this further yields an instantiation for the derived function mbind:

MonadPlus	CML
mbind	`fn ev => fn f => sync (wrap (ev, f))`
mjoin	`sync`

We may now check the monad laws. We will make use of the following equivalences:

$$\begin{array}{rcl}
\texttt{sync (alwaysEvt v)} & \simeq & \texttt{v} \\
\texttt{sync (wrap (ev, f))} & \simeq & \texttt{f (sync ev)}
\end{array}$$

The first monad law asserts that munit is a left identity for mbind:

$$\begin{array}{rcl}
\mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \simeq & f\ a \\
\texttt{(fn ev => fn f => sync (wrap (ev, f))) (alwaysEvt v) f} & \simeq^? & \texttt{f v} \\
\texttt{sync (wrap (alwaysEvt v, f))} & \simeq^? & \texttt{f v} \\
\texttt{f (sync (alwaysEvt v))} & \simeq^? & \texttt{f v} \\
\texttt{f v} & \simeq^? & \texttt{f v}
\end{array}$$

The second monad law asserts that munit is a right idenity for mbind:

$$
\begin{array}{rcl}
\mathsf{mbind}\ m\ \mathsf{munit} & \simeq & m \\
\texttt{(fn ev => fn f => sync (wrap (ev, f))) ev alwaysEvt} & \simeq^{?} & \texttt{ev} \\
\texttt{sync (wrap (ev, alwaysEvt))} & \simeq^{?} & \texttt{ev} \\
\texttt{alwaysEvt (sync ev)} & \simeq^{?} & \texttt{ev}
\end{array}
$$

This equivalence does not hold, since the left hand side forces the event to be synchronized on, while the right hand side denotes a quiescent event value.

The fact that this instantiation fails to satisfy the second monad law because it synchronizes on events "too early," suggests an alternative instantiation of mjoin:

MonadPlus	CML
mbind	`fn ev => fn f => wrap (wrap (ev, f), sync)`
mjoin	`fn ev => wrap (ev, sync)`

Returning to the first monad law, we have the following:

$$
\begin{array}{rcl}
\mathsf{mbind}\ (\mathsf{munit}\ a)\ f & \simeq & f\ a \\
\texttt{(fn ev => fn f => wrap (wrap (ev, f), sync)) (alwaysEvt v) f} & \simeq^{?} & \texttt{f v} \\
\texttt{wrap (wrap (alwaysEvt v, f), sync)} & \simeq^{?} & \texttt{f v}
\end{array}
$$

It is not entirely clear whether or not this equivalence holds. In Concurrent ML (as in Standard ML), a function may have arbitrary side-effects, including I/O. Hence, in this case, the left hand side denotes a quiescent event value, while the right hand side denotes an expression that evaluates to an event value. Should the function `f` perform I/O, then the equivalence does not hold.

This is a general issue with strict, mostly-functional languages with imperative I/O. While it would be convenient to conclude that the `event` type constructor does not form a monad, doing so for this reason will not illuminate some of the deeper issues that need to be resolved in order to design a calculus with first-class synchronous events forming a monad-with-plus. Therefore, let us assume that `f` is a pure, terminating function. In fact, let us consider the case where `f` is the function `fn x => never` and `v` is the value `0`. Under this instantiation, we are interested in the equivalence of:

$$\texttt{wrap (wrap (alwaysEvt 0, fn x => never), sync)} \quad \simeq^{?} \quad \texttt{never}$$

Although both sides denote quiescent event values, they are not equivalent, as may be seen by placing both events in the context `sync (choose [alwaysEvt 0, [ ]])`. The right hand side reduces as follows:

```
sync (choose [alwaysEvt 0, never])
↦ 0
```

since `choose` will never select `never`. On the other hand, the left hand side reduces as follows:

```
sync (choose [alwaysEvt 0, wrap (wrap (alwaysEvt 0, fn x => never), sync)])
↦ {0, sync ((fn x => never) 0)}
↦ {0, sync never}
↦ {0, ⊥}
```

where we have a set of possible outcomes and where $\perp$ denotes deadlock. The deadlocking alternative arises because `choose` may select either of the `alwaysEvt` events.

One might well question whether or not deadlock itself should be considered an observable behavior. Nonetheless, we may easily construct contexts in which the input/output behavior of the program (or, more properly, the set of possible input/output behaviors) depends upon whether or not an executing thread deadlocks. Hence, we will consider deadlock to be an (indirectly) observable value.

Since a context that distinguishes these events is expressed in terms of `choose`, we turn our attention to the monad-with-plus laws. It should be clear from Section 2.1 that `choose` and `never` satisfy the first three monad-with-plus laws and the first additional monad-with-plus law:

$$\begin{array}{rcl} \mathsf{mplus}\ m\ \mathsf{mzero} & \simeq & m \\ \mathsf{mplus}\ \mathsf{mzero}\ m & \simeq & m \\ \mathsf{mplus}\ m_1\ (\mathsf{mplus}\ m_2\ m_3) & \simeq & \mathsf{mplus}\ (\mathsf{mplus}\ m_1\ m_2)\ m_3 \\ \mathsf{mplus}\ m_1\ m_2 & \simeq & \mathsf{mplus}\ m_2\ m_1 \end{array}$$

asserting that mplus is a commutative and associative binary operation with mzero as a left and right identity.

The fourth monad-with-plus law asserts that mzero is a left zero for mbind:

$$\begin{array}{rcl} \mathsf{mbind}\ \mathsf{mzero}\ f & \simeq & \mathsf{mzero} \\ \texttt{wrap (wrap (never, f), sync)} & \simeq^? & \texttt{never} \end{array}$$

Both sides denote quiescent event values. Applying `sync` to both events yields the same observable behavior: deadlock. Furthermore, since `never` is the commit point for the left hand side, it should be clear that the two events are equivalent.

The fifth monad-with-plus law asserts that mbind distributes over mplus (to the left):

$$\mathsf{mbind}\ (\mathsf{mplus}\ m_1\ m_2)\ f \quad \simeq \quad \mathsf{mplus}\ (\mathsf{mbind}\ m_1\ f)\ (\mathsf{mbind}\ m_2\ f)$$

Suppose we instantiate m_1 with munit true, m_2 with munit false, and f with $\lambda b.$ if b then munit true else mzero. Then, in a monad-with-plus, we may reason as follows:

$$\begin{array}{lll} \multicolumn{2}{l}{\mathsf{mbind}\ (\mathsf{mplus}\ m_1\ m_2)\ f} & \\ \equiv & \mathsf{mbind}\ (\mathsf{mplus}\ (\mathsf{munit\ true})\ (\mathsf{munit\ false}))\ f & \text{definitions of } m_1 \text{ and } m_2 \\ \simeq & \mathsf{mplus}\ (\mathsf{mbind}\ (\mathsf{munit\ true})\ f)\ (\mathsf{mbind}\ (\mathsf{munit\ false})\ f) & \text{fifth monad-with-plus law} \\ \simeq & \mathsf{mplus}\ (f\ \mathsf{true})\ (f\ \mathsf{false}) & \text{first monad law} \\ \equiv & \mathsf{mplus}\ (\mathsf{munit\ true})\ \mathsf{mzero} & \text{definition of } f \\ \simeq & \mathsf{munit\ true} & \text{second monad-with-plus law} \end{array}$$

This equivalence does not hold in Concurrent ML, as may be seen by considering the context `sync [ ]`. The right hand side reduces as follows:

```
    sync (alwaysEvt true)
↦  true
```

On the other hand, the left hand side reduces as follows:

```
    sync (wrap (wrap (choose [alwaysEvt true, alwaysEvt false], f), sync))
↦  {sync (f true), sync (f false)}
↦  {sync (alwaysEvt true), sync never}
↦  {true, ⊥}
```

As this example shows, the `never` event is not revealed until it is too late to select a different, non-deadlocking alternative. We may see that this is related to the second additional monad-with-plus law, asserting that mzero is a right zero for mbind:

$$\begin{array}{rcl} \mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mzero}) & \simeq & \mathsf{mzero} \\ \texttt{wrap (wrap (ev, fn x => never), sync)} & \simeq^? & \texttt{never} \end{array}$$

This equivalence does not hold in Concurrent ML, as may be seen by considering the context `sync [ ]`. The right hand side immediately deadlocks, while the left hand side performs a synchronization and then deadlocks. If the event denoted by `ev` includes synchronous communication, then the left hand side may enable another thread (blocked waiting for communication) to proceed, while the right hand side will not enable the other thread.

Finally, consider the third additional monad-with-plus law, asserting that mbind distributes over mplus (to the right):

$$\mathsf{mbind}\ m\ (\lambda x.\ \mathsf{mplus}\ (f\ x)\ (g\ x)) \quad \simeq \quad \mathsf{mplus}\ (\mathsf{mbind}\ m\ f)\ (\mathsf{mbind}\ m\ g)$$

```
wrap (wrap (ev, fn x => choose [f x, g x]), sync)
   ≃? choose [wrap (wrap (ev, f), sync), wrap (wrap (ev, g), sync)]
```

Once again, we may see that the equivalence does not hold in Concurrent ML.

As we noted earlier, our original instantiation of the monad operations with Concurrent ML event combinators fails to satisfy the monad laws because it synchronizes on events "too early." Our second instantiation may be seen to have the opposite problem: it synchronizes on events "too late," whereby an event is not revealed until it is too late to select a different, non-deadlocking alternative.

We have also been fairly liberal in our analysis, choosing to assume that functions in the monad laws correspond to pure functions, which need not be the case in Concurrent ML, as functions may have abritrary side-effects, including I/O. In order to better separate concerns, we could posit a language in the spirit of Concurrent ML that made use of monadic I/O. In such a setting, we would not expect `sync` to have the type `'a event -> 'a`, as synchronization is not a pure function, but rather depends on the "state of the real world" (in the sense that the real world contains the state of concurrently executing threads). Hence, we would expect `sync` to have the type `'a event -> 'a io`. Under this scenario, we may not instatiate mjoin with `sync` or `fn ev => wrap (ev, sync)`, as these expressions have the type `'a event event -> 'a io event`.

Taken together, all of this suggests that taking sync as the basis of the monadic bind operator is fraught with peril. We solve these issues in the next section by presenting a new event combinator that directly corresponds to the monadic bind operator.

5 CFP: Concurrent Functional Pidgin

In this section, we present a novel concurrency language, dubbed Concurrent Functional Pidgin (CFP). Our language draws inspiration from both Concurrent ML and Concurrent Haskell. Like Concurrent ML, it includes first-class synchronous events and event combinators, but, unlike Concurrent ML, these events form a monad-with-plus. Like Concurrent Haskell, it includes first-class I/O actions and I/O combinators, and, like Concurrent Haskell, these actions form a monad.

We have purposefully kept our language as simple as possible. We will rapidly move through the syntax and static semantics. The most interesting aspect of the language is its operational semantics.

5.1 Syntax

Figure 3 gives the syntax of the CFP language. Values in the language naturally divide into four categories: standard λ-calculus terms, thread identifiers and channel names, event monad combinators, and I/O monad combinators. Most of the event combinators should be familar from the description of Concurrent ML: alwaysEvt v, neverEvt, and chooseEvt v_1 v_2. The exception is the thenEvt v_1 v_2 combinator, which has no direct analogue in Concurrent ML. In CFP, thenEvt v_1 v_2 is the mbind operator for the event monad-with-plus.

Likewise, most of the I/O combinators should be familiar from the description of Concurrent Haskell: unitIO v approximates `return v`, bindIO v_1 v_2 approximates `v1 >>= v2`, and getChar and putChar v approximate `getChar` and `putChar v`.

The remaining I/O combinators include process creation (spawn v), querying the thread identifier (getTId), channel creation (newChan), and event synchronization (sync v).

At the expression level, we add polymorphic let-binding and recursive functions.

Values	v	$::=$	x	variables
		$\mid$	$\mathfrak{b}$	base constants (chars, ints, etc.)
		$\mid$	$\lambda x.\ e$	function abstraction
		$\mid$	$(v_1, \ldots, v_n)$	tuples
		$\mid$	θ	thread identifiers
		$\mid$	κ	channel names
		$\mid$	$\mathsf{alwaysEvt}\ v$	always event (munit for Evt)
		$\mid$	$\mathsf{thenEvt}\ v_1\ v_2$	event sequence (mbind for Evt)
		$\mid$	$\mathsf{neverEvt}$	never event (mzero for Evt)
		$\mid$	$\mathsf{chooseEvt}\ v_1\ v_2$	event choice (mplus for Evt)
		$\mid$	$\mathsf{recvEvt}\ v$	receive event
		$\mid$	$\mathsf{sendEvt}\ v_1\ v_2$	send event
		$\mid$	$\mathsf{unitIO}\ v$	return action (munit for IO)
		$\mid$	$\mathsf{bindIO}\ v_1\ v_2$	action sequence (mbind for IO)
		$\mid$	$\mathsf{getChar}$	read action
		$\mid$	$\mathsf{putChar}\ v$	write action
		$\mid$	$\mathsf{spawn}\ v$	thread creation action
		$\mid$	getTId	thread id query action
		$\mid$	$\mathsf{newChan}$	channel creation action
		$\mid$	$\mathsf{sync}\ v$	event synchronization action
Expressions	e	$::=$	v	values
		$\mid$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let binding
		$\mid$	$\mu f.x.\ e$	recursive function abstraction
		$\mid$	$e_1\ e_2$	function application
		$\mid$	$\pi_i\ e$	tuple projection

Figure 3: CFP Syntax

$$
\begin{array}{llcll}
\textit{Types} & \tau & ::= & \mathsf{b} & \text{base type constants (int, char, etc.)} \\
 & & \mid & \alpha & \text{type variables} \\
 & & \mid & \tau_2 \to \tau_2 & \text{function type} \\
 & & \mid & \tau_1 \times \cdots \times \tau_n & \text{tuple type} \\
 & & \mid & \mathsf{TId} & \text{thread id type} \\
 & & \mid & \mathsf{Chan}\ \tau & \text{channel type} \\
 & & \mid & \mathsf{Evt}\ \tau & \text{event type (monadic)} \\
 & & \mid & \mathsf{IO}\ \tau & \text{IO type (monadic)}
\end{array}
$$

$$
\begin{array}{llcl}
\textit{Type Schemes} & \sigma & ::= & \tau \\
 & & \mid & \forall \alpha_1 \cdots \alpha_n.\ \tau
\end{array}
$$

$$
\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau}
\qquad
\frac{\mathrm{TypeOf}(\mathfrak{b}) \succ \tau}{\Gamma \vdash \mathfrak{b} : \tau}
\qquad
\frac{\Gamma, x{:}\tau_x \vdash e : \tau}{\Gamma \vdash \lambda x.\ e : \tau_x \to \tau}
\qquad
\frac{\Gamma \vdash v_1 : \tau_1 \quad \cdots \quad \Gamma \vdash v_n : \tau_n}{\Gamma \vdash (v_1, \ldots, v_n) : \tau_1 \times \cdots \times \tau_n}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_x \qquad \Gamma, x{:}\mathrm{Clos}_\Gamma(\tau_x) \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}
\qquad
\frac{\Gamma, f{:}\tau_x \to \tau, x{:}\tau_x \vdash e : \tau}{\Gamma \vdash \mu f.x.\ e : \tau_x \to \tau}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_x \to \tau \qquad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau}
\qquad
\frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n}{\Gamma \vdash \pi_i\ e : \tau_i}
$$

$$
\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathsf{alwaysEvt}\ v : \mathsf{Evt}\ \tau}
\qquad
\frac{\Gamma \vdash v_1 : \mathsf{Evt}\ \tau_x \qquad \Gamma \vdash v_2 : \tau_x \to \mathsf{Evt}\ \tau}{\Gamma \vdash \mathsf{thenEvt}\ v_1\ v_2 : \mathsf{Evt}\ \tau}
$$

$$
\frac{}{\Gamma \vdash \mathsf{neverEvt} : \mathsf{Evt}\ \tau}
\qquad
\frac{\Gamma \vdash v_1 : \mathsf{Evt}\ \tau \qquad \Gamma \vdash v_2 : \mathsf{Evt}\ \tau}{\Gamma \vdash \mathsf{chooseEvt}\ v_1\ v_2 : \mathsf{Evt}\ \tau}
$$

$$
\frac{\Gamma \vdash v : \mathsf{Chan}\ \tau}{\Gamma \vdash \mathsf{recvEvt}\ v : \mathsf{Evt}\ \tau}
\qquad
\frac{\Gamma \vdash v_1 : \mathsf{Chan}\ \tau \qquad \Gamma \vdash v_2 : \tau}{\Gamma \vdash \mathsf{sendEvt}\ v_1\ v_2 : \mathsf{Evt}\ ()}
$$

$$
\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \mathsf{unitIO}\ v : \mathsf{IO}\ \tau}
\qquad
\frac{\Gamma \vdash v_1 : \mathsf{IO}\ \tau_x \qquad \Gamma \vdash v_2 : \tau_x \to \mathsf{IO}\ \tau}{\Gamma \vdash \mathsf{bindIO}\ v_1\ v_2 : \mathsf{IO}\ \tau}
$$

$$
\frac{}{\Gamma \vdash \mathsf{getChar} : \mathsf{IO\ Char}}
\qquad
\frac{\Gamma \vdash v : \mathsf{Char}}{\Gamma \vdash \mathsf{putChar}\ v : \mathsf{IO}\ ()}
$$

$$
\frac{\Gamma \vdash v : \mathsf{IO}\ ()}{\Gamma \vdash \mathsf{spawn}\ v : \mathsf{IO\ TId}}
\qquad
\frac{}{\Gamma \vdash \mathsf{getTId} : \mathsf{IO\ TId}}
\qquad
\frac{}{\Gamma \vdash \mathsf{newChan} : \mathsf{IO}\ (\mathsf{Chan}\ \tau)}
\qquad
\frac{\Gamma \vdash v : \mathsf{Evt}\ \tau}{\Gamma \vdash \mathsf{sync}\ v : \mathsf{IO}\ \tau}
$$

Figure 4: CFP Static Semantics

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [] \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ e_2 \mid E_1\ e_2 \mid v_1\ E_2 \mid \pi_i\ E$$

$$\frac{}{E[\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2] \hookrightarrow E[e_2[v_1/x]]}$$

$$\frac{}{E[\mu f.x.e] \hookrightarrow E[\lambda x.e[\mu f.x.e/f]]}$$

$$\frac{\delta(\mathsf{b}, v) = v'}{E[\mathsf{b}\ v] \hookrightarrow E[v']}$$

$$\frac{}{E[(\lambda x.e)\ v] \hookrightarrow E[e[v/x]]}$$

$$\frac{}{E[\pi_i\ (v_1, \ldots, v_n)] \hookrightarrow E[v_i]}$$

Figure 5: CFP Dynamic Semantics: Sequential Evaluation

5.2 Static Semantics

For the static semantics, we adopt a simple Hindley-Milner [8] type system, summarized in Figure 4. These rules suffice for type-checking surface programs, in which thread identifiers θ and channel names κ may not be explicitly written by a programmer. If we wished to prove type soundness for the language (via progress and preservation), we would have an additional channel typing environment to assign types to allocated channels.

Note that our adoption of monadic I/O and events means that there is a pure, functional core language. Therefore, there is no need for a value restriction on the typing rule for polymorphic let-bindings.

5.3 Dynamic Semantics

The essence of the dynamics is to embrace the interpretation of monads as notions of computation. Hence, the two monads (Evt and IO), yield two sorts of computations. This is expressed by three levels of evaluation: pure evaluation of sequential terms, synchronous evaluation of synchronizing threads (e.g., the Evt monad), and concurrent evaluation of concurrent threads (e.g., the IO monad). The bridge between the Evt and IO monads is synchronization, which moves threads from concurrent evaluation to synchronous evaluation and back to concurrent evaluation.

5.3.1 Sequential Evaluation ($e \hookrightarrow e'$)

The "lowest" level of evaluation is the sequential evaluation of pure terms. We adopt a call-by-value evaluation strategy. Unsurprisingly, our sequential evaluation relation is entirely standard; see Figure 5.

5.3.2 Synchronous Evaluation ($(e_1, \ldots, e_k) \leadsto_k (e'_1, \ldots, e'_k)$)

The "middle" level of evaluation is synchronous evaluation of events. This evaluation relation is closely related to the event matching relation in the semantics of CML [16]. Intuitively, the relation:

$$(e_1, \ldots, e_k) \leadsto_k (e'_1, \ldots, e'_k)$$

means that the k events e_1,. . .,e_k make one step towards synchronization by transforming into the events e'_1,. . .,e'_k. The complete set of rules is given in Figure 6.

$$\textit{Synchronous Evaluation Contexts} \quad M^{Evt} \quad ::= \quad [] \mid \textsf{thenEvt}\ M_1^{Evt}\ v_2$$

$$\textsc{EvtEval}\quad \frac{E[e] \hookrightarrow E[e']}{(M^{Evt}[E[e]]) \rightsquigarrow_1 (M^{Evt}[E[e']])}$$

$$\textsc{EvtBind}\quad \frac{}{(M^{Evt}[\textsf{thenEvt}\ (\textsf{alwaysEvt}\ v_1)\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2\ v_1])}$$

$$\textsc{EvtPlus1}\quad \frac{}{(M^{Evt}[\textsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_1])} \qquad \textsc{EvtPlus2}\quad \frac{}{(M^{Evt}[\textsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2])}$$

$$\textsc{EvtSendRecv}\quad \frac{}{(M_1^{Evt}[\textsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\textsf{recvEvt}\ \kappa]) \rightsquigarrow_2 (M_1^{Evt}[\textsf{alwaysEvt}\ ()], M_2^{Evt}[\textsf{alwaysEvt}\ v])}$$

$$\textsc{Permutation}\quad \frac{p \in \mathrm{Perm}_k \qquad (e_{p(1)}, \ldots, e_{p(k)}) \rightsquigarrow_k (e'_1, \ldots, e'_k)}{(e_1, \ldots, e_k) \rightsquigarrow_k (e'_{\overline{p}(1)}, \ldots, e'_{\overline{p}(k)})}$$

$$\textsc{Subset}\quad \frac{1 \leq j \leq k \qquad (e_1, \ldots, e_j) \rightsquigarrow_j (e'_1, \ldots, e'_j)}{(e_1, \ldots, e_j, e_{j+1} \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_j, e_{j+1} \ldots, e_k)}$$

Figure 6: CFP Dynamic Semantics: Synchronous Evaluation

The rule EvtEval implements the sequential evaluation of an expression in the active position. The rule EvtBind implements sequential composition in the Evt monad. The rules EvtPlus1 and EvtPlus2 implements a non-deterministic choice between events. The rule EvtSendRecv implements the two-way rendezvous of communication along a channel; note that the transition replaces the sendEvt and recvEvt events with A events. The final two rules are structural rules that admit transitioning on a permutation of the set of events and transitioning on a subset of the set of events.

It is worth considering the possible terminal configurations for a set of events. The "good" terminal configuration is one in which all events are reduced to always events: (alwaysEvt $v_1, \ldots,$ alwaysEvt v_k). The "bad" terminal configurations are ones with a never event neverEvt or with unmatched send/receive events.

5.3.3 Concurrent Evaluation ($\mathcal{T} \xrightarrow{a} \mathcal{T}'$) (Angelic)

The "highest" level of evaluation is concurrent evaluation of threads. We organize the executing group of threads as a set of pairs of thread IDs and expressions:

$$\begin{array}{lcll} \textit{Concurrent Threads} & T & ::= & \langle \theta, e \rangle \\ \textit{Concurrent Thread Groups} & \mathcal{T} & ::= & \{T, \ldots\} \end{array}$$

To model the input/output behavior of the program, transitions are labelled with an optional action:

$$\textit{Actions} \quad a \quad ::= \quad ?c \mid !c \mid \epsilon$$

The actions allow reading a character c from standard input ($?c$) or writing a character c to standard output ($!c$). The silent action ϵ indicates no observable input/output behavior.

The complete set of rules is given in Figure 7.

All of the rules include non-deterministically choosing one or more threads for a step of evaluation. The rule IOEval implements the sequential evaluation of an expression in the active position. The rule IOBind implements sequential composition in the IO monad. The two rules IOGetChar and IOPutChar perform the appropriate labelled transition, yielding an observable action.

The rule Spawn creates a new thread by selecting a fresh thread ID. The rule GetTID returns the thread's identifier. The rule NewChan creates a new channel by selecting a fresh channel name.

The most interesting rule is Sync. The rule selects some collection of threads that are prepared to synchronize on event values. This set of event values is passed to the synchronous evaluation relation, which takes *multiple transitions* to yield a terminal configuration in which all events are reduced to always events. That is, the set of events successfully synchronize to final results. The results of synchronization are moved from the Evt monad to the IO monad.

There are two interesting facets of the Sync rule. The first is that the concurrent transition has a silent action. Hence, synchronization itself is not observable, though it may unblock a thread so that subsequent i/o actions are observed. Likewise, individual synchronous evaluation transitions do not yield observable actions.

The second is the fact that multiple synchronous evaluation steps correspond to a single concurrent evaluation step. Transitions from different threads may be interleaved, but Sync prevents transitions from different sets of synchronizing events from being interleaved. Hence, synchronization executes "atomically," although the synchronization of a single event is not "isolated" from the synchronizations of other events. (Indeed, it is imperative that multiple events synchronize simultaneously in order to enable synchronous communication along channels.) Note that the Sync rule conveys an *all-or-nothing* property on event synchronization and on the thenEvt v_1 v_2 event combinator. When a synchronizing on event of the form thenEvt v_1 v_2, the synchronous evaluation must first synchronize on v_1, yielding alwaysEvt v_1', and then synchronize on v_2 v_1'. Should the event v_2 v_1' reduce to a bad terminal configuration, then the semantics may not choose this evaluation.

$$\textit{Concurrent Evaluation Contexts} \quad M^{IO} \quad ::= \quad [] \mid \mathsf{bindIO}\ M_1^{IO}\ v_2$$

$$\textsc{IOEval}\quad \frac{E[e] \hookrightarrow E[e']}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e]] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e']] \rangle\}}$$

$$\textsc{IOBind}\quad \frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{bindIO}\ (\mathsf{unitIO}\ v_1)\ v_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1] \rangle\}}$$

$$\textsc{IOGetChar}\quad \frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}] \rangle\} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c] \rangle\}}$$

$$\textsc{IOPutChar}\quad \frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c] \rangle\} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()] \rangle\}}$$

$$\textsc{Spawn}\quad \frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{spawn}(v)] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta'] \rangle, \langle \theta', v \rangle\}}$$

$$\textsc{GetTId}\quad \frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getTId}] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta] \rangle\}}$$

$$\textsc{NewChan}\quad \frac{\kappa' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{newChan}] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \kappa'] \rangle\}}$$

$$\textsc{Sync}\quad \frac{(v_1, \ldots, v_k) \rightsquigarrow_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1'] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k'] \rangle\}}$$

Figure 7: CFP Dynamic Semantics: Concurrent Evaluation (Angelic)

5.3.4 Concurrent Evaluation ($\mathcal{T}; \mathcal{S} \xrightarrow{a} \mathcal{T}'; \mathcal{S}'$) (Djinnish)

The subtleties of this *all-or-nothing* property may be more apparent by considering an alternative concurrent evaluation relation. The previous concurrent evaluation relation may be termed "angelic," in the sense that the Sync rule omniciently and benevolently chooses a good set of synchronizing threads and a good sequence of synchronous evaluation transitions.

In our alternative semantics, we only allow single steps of the synchronous evaluation within a single step of the concurrent evaluation. To accomodate this, we introduce synchronous thread groups:

Synchronization IDs	ϕ	$\in$	*SId*
Synchronizing Threads	S	$::=$	$\langle \phi, \langle \langle \theta_1, M_1^{IO}, v_1, e_1' \rangle, \ldots, \langle \theta_k, M_k^{IO}, v_k, e_k' \rangle \rangle \rangle$
Synchronizing Thread Groups	$\mathcal{S}$	$::=$	$\{S, \ldots\}$

Now, our concurrent evaluation relation consists of both concurrent threads and synchronizing threads. Figure 8 gives the complete set of rules.

We have replaced the single Sync rule with four separate rules: SyncInit, SyncStep, SyncCommit, and SyncAbort. The SyncInit rule selects some collection of threads that are prepared to synchronize on event values. Rather than immediately synchronizing on the event values, the rule simply gathers the threads into a collection of synchronizing threads. The SyncStep rule takes a single step in the synchronous evaluation relation. After taking sufficient steps in the synchronous evaluation relation, the collection of events may reach either a good or a bad terminal configuration. If the collection of events reaches a good terminal configuration, then we have acheived a successful synchronization, and the SyncCommit rule moves the results of synchronization from the Evt monad to the IO monad and dissolves the collection of synchronizing threads. If, on the other hand, the collection of events reaches a bad terminal configuration, then the collection of synchronizing threads may be dissolved by the SyncAbort rule, which restores all the threads to their initial synchronizing state.

This concurrent evaluation relation may be termed "djinnish," in the sense that the Sync* rules are still quite powerful (acting on entire sets of threads at once), but need not act entirely in a benevolent manner. However, it is useful to note that the two semantics ultimately have the same observable behavior (at the quiescent points where there are no synchronizing threads):

Theorem 1

$\mathcal{T} \xrightarrow{a_1; \cdots ; a_n}{}^{*} \mathcal{T}'$ in the angelic semantics if and only if $\mathcal{T}; \{\} \xrightarrow{a_1'; \cdots ; a_m'}{}^{*} \mathcal{T}'; \{\}$ in the djinnish semantics, where $a_1; \cdots ; a_n$ equals $a_1'; \cdots ; a_m'$ modulo the insertion and deletion of ϵ actions.

This theorem relies upon the fact that synchronous evaluation transitions do not lead to observable concurrent evaluation transitions. Hence, an any "angelic" synchronization may be expanded into a corresponding committing "djinnish" synchronization (adding multiple ϵ actions). In the other direction, we note that any committing "djinnish" synchronization may be collapsed into a corresponding "angelic" synchronization (deleting multipe ϵ actions), while any aborting "djinnish" synchronization may be elided completely (deleting multiple ϵ actions).

5.4 Discussion

The "djinnish" semantics given in the previous section sheds some light on the nature of the Evt monad.

We may interpret the synchronous evaluation of events as a non-deterministic search with backtracking. That is, the synchronous evaluation of events is searching for a successful synchronization. Furthermore, the search backtracks at synchronization failures (e.g, neverEvt events and unmatched send/receive events).

We may also interpret the synchronous evaluation of events as an abortable transaction. That is, the synchronization of events must happen atomically with respect to other synchronizations and I/O actions. Furthermore, the transaction aborts (with no observable effects) at synchronization failures.

Both of these interpretations clarify the nature of the *all-or-nothing* property of the Evt monad-with-plus. While the "angelic" semantics specifies our intended meaning, it provides no insight into how such

Concurrent Evaluation Contexts $M^{IO} ::= [] \mid \textsf{thenEvt}\ M_1^{IO}\ v_2$

IOEval
$$\frac{E[e] \hookrightarrow E[e']}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e]]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e']]\rangle\}; \mathcal{S}}$$

IOBind
$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{bindIO}\ (\textsf{unitIO}\ v_1)\ v_2]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1]\rangle\}; \mathcal{S}}$$

IOGetChar
$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{getChar}]\rangle\}; \mathcal{S} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{unitIO}\ c]\rangle\}; \mathcal{S}}$$

IOPutChar
$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{putChar}\ c]\rangle\}; \mathcal{S} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{unitIO}\ ()]\rangle\}; \mathcal{S}}$$

Spawn
$$\frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{spawn}\ v]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{unitIO}\ \theta']\rangle, \langle \theta', v\rangle\}; \mathcal{S}}$$

GetTId
$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{getTId}]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{unitIO}\ \theta]\rangle\}; \mathcal{S}}$$

NewChan
$$\frac{\kappa' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{newChan}]\rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\textsf{unitIO}\ \kappa']\rangle\}; \mathcal{S}}$$

SyncInit
$$\frac{\phi' \text{ fresh}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\textsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\textsf{sync}\ v_k]\rangle\}; \mathcal{S} \\ \quad \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \phi', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}\end{array}}$$

SyncStep
$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}\end{array}}$$

SyncCommit
$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, \textsf{alwaysEvt}\ v_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, \textsf{alwaysEvt}\ v_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\textsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\textsf{unitIO}\ v_k']\rangle\}; \mathcal{S}\end{array}}$$

SyncAbort
$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\textsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\textsf{sync}\ v_k]\rangle\}; \mathcal{S}\end{array}}$$

Figure 8: CFP Dynamic Semantics: Concurrent Evaluation (Djinnish)

$$\begin{aligned}\mathsf{mapEvt}\ f\ e &= \mathsf{thenEvt}\ e\ (\lambda x.\ \mathsf{alwaysEvt}\ (f\ x))\\ \mathsf{joinEvt}\ ee &= \mathsf{thenEvt}\ ee\ \mathsf{id}\end{aligned}$$

$$\begin{aligned}\mathsf{mapIO}\ f\ i &= \mathsf{bindIO}\ i\ (\lambda x.\ \mathsf{unitIO}\ (f\ x))\\ \mathsf{joinIO}\ ii &= \mathsf{bindIO}\ ii\ \mathsf{id}\end{aligned}$$

Figure 9: Monadic map and join operations

an execution may be realized in practice. The "djinnish" semantics (and its equivalence to the "angelic" semantics) demonstrates that we may tentatively evaluate synchronizations, while retaining the ability to freely abandon the evaluation. We have used the IO monad to ensure that truly irrevocable (i.e., observable) actions cannot take place during the evaluation of a synchronization.

5.4.1 Monad Laws

It should be mostly clear that the IO type constructor of CFP forms a monad and the Evt type constructor of CFP forms a monad-with-plus.

The former follows directly from the fact that the IOBind rule captures the fact that unitIO is a left identity for bindIO and the evaluation context M^{IO}.

Likewise, the fact that the Evt type constructor forms a monad follows from the EvtBind rule and the evaluation context M^{Evt}. In order to see that Evt forms a monad-with-plus, we note that the commutativity and associativity of chooseEvt follows from the rules EvtPlus1 and EvtPlus2.

The rules relating to neverEvt all follow from the absence of any rules for reducing an event with neverEvt, which in turn prohibits any observable IO actions from following the synchronization on a neverEvt event. From this, we may see that neverEvt is a left and right identity for chooseEvt and that neverEvt is a left and right zero for thenEvt.

The distribution laws also follow from the *all-or-nothing* property of event synchronization. This ensures that the relative ordering of event sequence and event choice does not affect the outcome of the event synchronization.

5.4.2 Spirit of Concurrent ML

Finally, we may see that CFP preserves the "spirit" of CML. Recall from Section 2.1 that one often wants to implement a protocol consisting of a sequence of communications: $c_1 ; c_2 ; \cdots ; c_n$. When this protocol is used in a selective communication, one of the c_i is designated as the *commit point*, the communication by which this protocol is chosen over the others in a select. The monadic bind combinator of the Evt monad obviates the need to distinguish one communication as the commit point (and the complication of a protocol that must be robust against failures in the communications $c_1 ; \cdots ; c_{i-1}$ and $c_{i+1} ; \cdots ; c_n$).

Instead, we may implement the protocol as a sequence of communications: c_1 >> c_2 >> $\cdots$ >> c_n, where the monadic bind combinator ensures that all of the communications synchronize or none of them synchronize. When this protocol participates in a select, it will be chosen only if all of the communications are able to synchronize with corresponding communications in other synchronizing threads.

6 Encoding CML

In this section, we consider how to encode Concurrent ML in the concurrency language of the previous section. To improve the readability of the encoding, we recall the monadic map and monadic join operations from Section 3 (see Figure 9). Additionally, we will freely use tuple patterns in λ arguments and write recursive let bindings.

$$\begin{array}{rcl}
\mathsf{recvCMLEvt} & :: & \mathsf{Chan}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{recvCMLEvt}\ c & = & \mathsf{lift}\ (\mathsf{recvEvt}\ c) \\
\mathsf{sendCMLEvt} & :: & \mathsf{Chan}\ \alpha \to \alpha \to \mathsf{CMLEvt}\ () \\
\mathsf{sendCMLEvt}\ c\ x & = & \mathsf{lift}\ (\mathsf{sendEvt}\ c\ x) \\
\\
\mathsf{wrapCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \to (\alpha \to \mathsf{IO}\ \beta) \to \mathsf{CMLEvt}\ \beta \\
\mathsf{wrapCMLEvt}\ iei\ f & = & \mathsf{mapIO}\ (\lambda ei.\ \mathsf{mapEvt}\ (\lambda io.\ \mathsf{bindIO}\ io\ f)\ ei)\ iei \\
\mathsf{guardCMLEvt} & :: & \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{guardCMLEvt}\ iiei & = & \mathsf{joinIO}\ iiei \\
\mathsf{chooseCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{chooseCMLEvt}\ iei_1\ iei_2 & = & \mathsf{bindIO}\ iei_1\ (\lambda ei_1. \\
 & & \mathsf{bindIO}\ iei_2\ (\lambda ei_2. \\
 & & \mathsf{unitIO}\ (\mathsf{chooseEvt}\ ei_1\ ei_2)\ )) \\
\\
\mathsf{alwaysCMLEvt} & :: & \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{alwaysCMLEvt}\ x & = & \mathsf{lift}\ (\mathsf{alwaysEvt}\ x) \\
\mathsf{neverCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \\
\mathsf{neverCMLEvt} & = & \mathsf{lift}\ \mathsf{neverEvt} \\
\\
\mathsf{syncCML} & :: & \mathsf{CMLEvt}\ \alpha \to \mathsf{IO}\ \alpha \\
\mathsf{syncCML}\ iei & = & \mathsf{bindIO}\ iei\ (\lambda ei.\ \mathsf{joinIO}\ (\mathsf{sync}\ ei))
\end{array}$$

Figure 10: Simple CML Encoding

6.1 Simple CML

We first consider a simple CML encoding, where we do not support the `withNack` event combinator. (We futher simplify the encoding by only considering a binary `choose` combinator.)

As we noted in Section 4, functions in Stanard ML and Concurrent ML may have arbitrary side-effects, including synchronization and I/O. One way to interpret this fact is to consider that Standard ML functions evaluate in a "built-in" I/O monad. While a general translation from Standard ML with imperative I/O into a language with monadic I/O is beyond the scope of this paper (but see [9, 10]), we note that the general idea is to translate a function of the type $\tau_1 \to \tau_2$ into a function of the type $\tau_1 \to \mathsf{IO}\ \tau_2$.

Recall that the `guard` and `wrap` primitives of Concurrent ML add arbitrary pre- and post-synchronization actions to an event. We may encode this by interpreting a Concurrent ML event as a pre-synchronization IO action that yields an Evt value that yields a post-synchronous IO action:

$$\begin{array}{rcl}
\mathsf{CMLEvt} & :: & \star \to \star \\
\mathsf{CMLEvt}\ \alpha & = & \mathsf{IO}\ (\mathsf{Evt}\ (\mathsf{IO}\ \alpha))
\end{array}$$

There is a trivial lifting from Evt values to CMLEvt values:

$$\begin{array}{rcl}
\mathsf{lift} & :: & \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{lift}\ e & = & \mathsf{unitIO}\ (\mathsf{mapEvt}\ e\ \mathsf{unitIO})
\end{array}$$

The encodings of the Concurrent ML combinators are given in Figure 10. We use lift to coerce the simple event combinators into the CMLEvt type. Note the manner in which syncCML performs the "outer" IO action, then performs the synchronization of the Evt value, then performs the "inner" IO action.

This encoding of Concurrent ML reveals that one significant reason that the `event` type constructor does not form a monad-with-plus is simply the fact that IO does not form a monad-with-plus.

$$
\begin{array}{rcl}
\mathsf{SVar} & :: & \star \\
\mathsf{SVar} & = & \mathsf{Chan}\ () \\[1em]
\mathsf{newSVar} & :: & \mathsf{IO\ SVar} \\
\mathsf{newSVar} & = & \mathsf{newChan} \\[1em]
\mathsf{setSVar} & :: & \mathsf{SVar} \to \mathsf{IO}\ () \\
\mathsf{setSVar}\ s & = & \mathsf{let\ loop} :: \mathsf{SVar} \to \mathsf{IO}\ () \\
 & & \quad\ \mathsf{loop}\ s = \mathsf{bindIO}\ (\mathsf{sync}\ (\mathsf{sendEvt}\ s\ ()))\ (\lambda().\\
 & & \qquad\qquad\qquad\qquad (\mathsf{loop}\ s)\ )\ \mathsf{in} \\
 & & \mathsf{spawn}\ (\mathsf{loop}\ s) \\[1em]
\mathsf{getSVarEvt} & :: & \mathsf{SVar} \to \mathsf{Evt}\ () \\
\mathsf{getSVarEvt}\ s & = & \mathsf{recvEvt}\ s
\end{array}
$$

Figure 11: Signal Variable Encoding

6.2 Signal Variables

Before giving an encoding for `withNack`, we present a simple *signal variable* abstraction (see Figure 11). A signal variable is a variable that may be asynchronously enabled and that may be synchronized upon, blocking until the variable is enabled. Note that since a signal variable may not be disabled, there is no harm in enabling a signal variable multiple times.

The abstraction is quite simple to implement. A signal variable is simply a channel carrying unit. Creating a new signal variable amounts to creating a new channel. Enabling a signal variable requires spawning a thread that repeatedly sends unit along the channel. Note that enabling a signal variable simply spawns additional threads, each repeatedly sending unit along the channel. The event value for synchronizing on a signal variable is simply the read event.

6.3 Full CML

To encode the `withNack` combinator of Concurrent ML, we simply augment the "outer" IO action with a list of the signal variables created for constituent events and augment the "inner" IO action with a list of the signal variables to enable:

$$
\begin{array}{rcl}
\mathsf{CMLEvt} & :: & \star \to \star \\
\mathsf{CMLEvt}\ \alpha & = & \mathsf{IO}\ (\mathsf{List\ SVar} \times \mathsf{Evt}\ (\mathsf{List\ SVar} \times \mathsf{IO}\ \alpha))
\end{array}
$$

As before, there is a trivial lifting from Evt values to CMLEvt values:

$$
\begin{array}{rcl}
\mathsf{lift} & :: & \mathsf{Evt}\ \alpha \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{lift}\ e & = & \mathsf{unitIO}\ (\mathsf{nil}, \mathsf{mapEvt}\ e\ (\lambda x.\ (\mathsf{nil}, \mathsf{unitIO}\ x)))
\end{array}
$$

which may be used to encode recvEvt, sendEvt, alwaysEvt, and neverEvt.

A more interesting encoding is the encoding of withNackCMLEvt:

$$
\begin{array}{rcl}
\mathsf{withNackCMLEvt} & :: & (\mathsf{CMLEvt}\ () \to \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha)) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{withNackCMLEvt}\ f & = & \mathsf{bindIO\ newSVar}\ (\lambda s. \\
 & & \mathsf{bindIO}\ (\mathsf{joinIO}\ (f\ (\mathsf{lift}\ (\mathsf{getSVarEvt}\ s))))\ (\lambda(ss, ei). \\
 & & \mathsf{unitIO}\ (\mathsf{cons}\ s\ ss, ei)\ )) \\
\mathsf{guardCMLEvt} & :: & \mathsf{IO}\ (\mathsf{CMLEvt}\ \alpha) \to \mathsf{CMLEvt}\ \alpha \\
\mathsf{guardCMLEvt}\ iei & = & \mathsf{withNack}\ (\lambda\_.\ iei)
\end{array}
$$

A new signal variable is created and provided to the function f, which returns an IO action yielding a list of signal variables and an event value. The new signal variable is consed onto the head of the list of signal variables and the event value is returned unmodified.

The chooseCMLEvt combinator must combine the created signal variables of each event and also add the created signal variables from one event to the to-be-enabled signal variables of the other event, and vice versa.

$$\begin{array}{rcl}
\mathsf{chooseCMLEvt} & :: & \mathsf{CMLEvt}\ \alpha \rightarrow \mathsf{CMLEvt}\ \alpha \rightarrow \mathsf{CMLEvt}\ \alpha \\
\mathsf{chooseCMLEvt}\ iei_1\ iei_2 & = & \mathsf{bindIO}\ iei_1\ (\lambda(ss_1, ei_1). \\
 & & \mathsf{bindIO}\ iei_2\ (\lambda(ss_2, ei_2). \\
 & & \mathsf{unitIO}\ (\mathsf{append}\ ss_1\ ss_2, \\
 & & \quad \mathsf{chooseEvt}\ (\mathsf{mapEvt}\ (\lambda(ss, i).\ (\mathsf{append}\ ss_2\ ss, i))\ ei_1) \\
 & & \qquad\qquad (\mathsf{mapEvt}\ (\lambda(ss, i).\ (\mathsf{append}\ ss_1\ ss, i))\ ei_2))))
\end{array}$$

The encoding of syncCML performs the "outer" IO action, discards the created signal variables, performs the event synchronization, enables the appropriate signal variables, and finally performs the "inner" IO action:

$$\begin{array}{rcl}
\mathsf{syncCML} & :: & \mathsf{CMLEvt}\ \alpha \rightarrow \mathsf{IO}\ \alpha \\
\mathsf{syncCML}\ iei & = & \mathsf{bindIO}\ iei\ (\lambda(\_, ei). \\
 & & \mathsf{bindIO}\ (\mathsf{sync}\ ei)\ (\lambda(ss, i). \\
 & & \mathsf{fold}\ (\lambda(s, i).\ \mathsf{bindIO}\ (\mathsf{setSVar}\ s)\ (\lambda().\ i))\ i\ ss))
\end{array}$$

Note that the fold ensures that the "inner" IO action of the event encoding is performed after all of the signal variables have been enabled.

6.3.1 Discussion

There is an interesting consequence of the encoding of Full CML. Note that the encoding only makes use of thenEvt through mapEvt. Furthermore, note that when mapEvt is applied to a pure, terminating function (as it is in all of the functions above), then CML's `wrapEvt` combinator provides essentially the same functionality. This suggests that neither `withNack` nor `guard` need be taken as primitives, but rather may be implemented as a stylized use of a core CML. This greatly simplifies both the meta-theory of CML and the implentation of CML, without changing the expressive power of CML.

This "reverse encoding" requires recognizing that CFP's use of IO for pre- and post-synchronization actions may be approximated by thunks in SML. Figures 12 and 13 demonstrate how the complete set of CML combinators may be implemented with only a primitive set of CML combinators. In the future, we hope to experiment with an implementation of CML following this approach and measure the performance. While the extra closure creation may incur a performance penalty, the `PrimCML` implementation will be simpler and possibly faster. We also note that since `PrimCML.wrap` is only ever applied to pure, terminating functions, then one need not be pessimistic about their behavior; in particular, it is not strictly necessary to run `PrimCML.wrap` functions outside of critical regions.

7 Encoding Transactional Shared Memory

It is well known that synchronous message passing may be used to implement shared memory. For instance, there are canonical encodings of mutable variables in Concurrent ML [16, Sections 3.2.3 and 3.2.7]. Since we may view CFP as extending CML with an *all-or-nothing* transactional property, an interesting question is whether or not we may encode shared memory transactions in CFP. This section demonstrates such an encoding.

We take as our starting point the software transactional memory (STM) extension of Concurrent Haskell [2]. STM Haskell provides two monadic types that denote computations: `IO` and `STM`. The former is like `IO` and IO presented previously; it represents an action that may perform some input/output and

```
signature PRIM_CML =
   sig
      type thread_id
      val spawn : (unit -> unit) -> thread_id

      type 'a chan
      val channel : unit -> 'a chan

      type 'a event

      val sync : 'a event -> 'a
      val wrap : 'a event * ('a -> 'b) -> 'b event
      val choose : 'a event * 'a event -> 'a event

      val alwaysEvt : 'a -> 'a event
      val neverEvt : 'a event
      val recvEvt : 'a chan -> 'a event
      val sendEvt : 'a chan * 'a -> unit event
   end

signature FULL_CML =
   sig
      include PRIM_CML

      val withNack : (unit event -> 'a event) -> 'a event
      val guard : (unit -> 'a event) -> 'a event
   end

functor MkSVar(structure PrimCML : PRIM_CML) :
   sig
      type t
      val svar : unit -> t
      val setSVar : t -> unit
      val getSVarEvt : t -> unit PrimCML.event
   end =
   struct
      type t = unit PrimCML.chan
      val svar = PrimCML.channel
      fun setSVar c =
         let
            fun loop () = (PrimCML.sync (PrimCML.sendEvt (c, ()))
                           ; loop ())
         in
            ignore (PrimCML.spawn loop)
         end
      fun getSVarEvt c = PrimCML.recvEvt c
   end
```

Figure 12: Simplified CML Implementation (I)

```
functor MkFullCML(structure PrimCML : PRIM_CML) :> FULL_CML =
   struct
      val spawn = PrimCML.spawn

      type 'a chan = 'a PrimCML.chan
      val channel = PrimCML.channel

      structure SVar = MkSVar(structure PrimCML = PrimCML)

      type 'a thunk = unit -> 'a
      type 'a event = (SVar.t list * (SVar.t list * 'a thunk) PrimCML.event) thunk

      fun lift (ev : 'a PrimCML.event) : 'a event =
         fn () => ([], PrimCML.wrap (ev, fn x => ([], fn () => x)))

      fun sync (ev : 'a event) : 'a =
         let
            val (_, pev) = ev ()
            val (svars, th) = PrimCML.sync pev
         in
            List.app SVar.setSVar svars
            ; th ()
         end
      fun wrap (ev : 'a event, f : 'a -> 'b) : 'b event =
         fn () =>
         let
            val (svars, pev) = ev ()
            val pev' = PrimCML.wrap (pev, fn (svars, th) => (svars, f o th))
         in
            (svars, pev')
         end
      fun choose (ev1 : 'a event, ev2: 'a event) : 'a event =
         fn () =>
         let
            val (svars1, pev1) = ev1 ()
            val (svars2, pev2) = ev2 ()
         in
            (svars1 @ svars2,
             PrimCML.choose (PrimCML.wrap (pev1, fn (svars, th) =>
                                              (svars @ svars2, th)),
                             PrimCML.wrap (pev2, fn (svars, th) =>
                                              (svars @ svars1, th))))
         end
      fun withNack (f : unit event -> 'a event) : 'a event =
         fn () =>
         let
            val svar = SVar.svar ()
            val ev = f (lift (SVar.getSVarEvt svar))
            val (svars, pev) = ev ()
         in
            (svar :: svars, pev)
         end
      fun guard (f : unit -> 'a event) : 'a event =
         fn () => f () ()

      fun alwaysEvt (x : 'a) : 'a event = lift (PrimCML.alwaysEvt x)
      (* The value restriction prohibits the following definition: *)
      (* val neverEvt : 'a event = lift PrimCML.neverEvt           *)
      val neverEvt : 'a event = fn () => ([], PrimCML.neverEvt)
      fun recvEvt (c : 'a chan) : 'a event = lift (PrimCML.recvEvt c)
      fun sendEvt (c : 'a chan, x : 'a) : unit event = lift (PrimCML.sendEvt (c, x))
   end
```

Figure 13: Simplified CML Implementation (II)

is the type of a running thread. The second is the type of an atomic memory transaction, which has the following interface:

```
data STM a

atomic :: STM a -> IO a

unitSTM :: a -> STM a
bindSTM :: STM a -> (a -> STM b) -> STM b
retrySTM :: STM a
orElseSTM :: STM a -> STM a -> STM a

instance Monad STM
  return = unitSTM
  >>= = bindSTM
instance MonadPlus STM where
  mzero = retrySTM
  mplus = orElseSTM

data TVar
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

The `STM` type forms a monad-with-plus, where `retry` aborts a transaction and `orElse` selects (with left bias) between transactions. (Note that the left-bias of `orElse` means that it does not satisfy the commutativity law.)

There are obvious connections between the STM extension of Concurrent Haskell and CFP. Both use an "outer" I/O monad to sequence observable, irrevocable effects and both use an "inner" monad to encapsulate thread interactions in a manner that ensures that the effect of those interactions are not apparent until the interaction executes with a consistent view. For STM, this interaction and consistent view are atomic access by a single thread to transactional variables. For CFP, this interaction and consistent view are synchronization by multiple threads on channel communication.

Our encoding of the `STM` monad-with-plus makes two relatively minor changes to the semantics. First, in order that `orElse` may be more easily encoded by chooseEvt, we eliminate the left-bias. Second, our encoding is in the spirit of previous encodings of mutable variables, whereby a server thread maintains the state of the variable and services requests to get or set the variable's contents. Hence, creating a new mutable variable requires spawning a new thread. Therefore, in our implementation, we give newTVar the type $\alpha \rightarrow \mathsf{IO}\ (\mathsf{TVar}\ \alpha)$, which is a consequence of the fact that spawning a thread must occur in the IO monad. Nonetheless, we feel that this encoding is well within the spirit of transactional memory and demonstrates the expressibility of CFP.

Figures 14 and 15 give the encoding, which we discussion in some detail. The high-level view of the encoding is quite simple. Recall that each transaction variable will be represented by a server thread. A thread wishing to read or write transaction variables atomically sends its thread identifier to the server thread. If a server thread receives a thread identifier of a second thread while the first thread's transaction is incomplete, it aborts the transaction (by attempting to synchronize on neverEvt). Hence, a thread completes its atomic transaction if and only if it is the only thread to communicate with those transactional variables during the transaction.

From the description above, it is clear that we essentially want the encoding of the `STM` monad-with-plus to be the Evt monad-with-plus. However, we must also provide the thread identifier as part of each read or write of a transactional variable. Hence, we define $\mathsf{STM}\ \alpha$ to be $\mathsf{TId} \rightarrow \mathsf{Evt}\ \alpha$. Monad aficionados will recognize this as lifting the Evt monad through the reader (a.k.a. environment) monad transformer. The encoding of atomic simply queries the thread identifier, supplies it to the STM computation, and synchronizes on the

$$
\begin{array}{rcl}
\mathsf{STM} & :: & \star \to \star \\
\mathsf{STM}\ \alpha & = & \mathsf{TId} \to \mathsf{Evt}\ \alpha \\
\\
\mathsf{atomic} & :: & \mathsf{STM}\ \alpha \to \mathsf{IO}\ \alpha \\
\mathsf{atomic}\ s & = & \mathsf{bindIO}\ \mathsf{getTId}\ (\lambda tid.\ \mathsf{sync}\ (s\ tid)) \\
\\
\mathsf{unitSTM} & :: & \alpha \to \mathsf{STM}\ \alpha \\
\mathsf{unitSTM}\ x & = & \lambda tid.\ \mathsf{alwaysEvt}\ x \\
\mathsf{bindSTM} & :: & \mathsf{STM}\ \alpha \to (\alpha \to \mathsf{STM}\ \beta) \to \mathsf{STM}\ \beta \\
\mathsf{bindSTM}\ s\ f & = & \lambda tid.\ \mathsf{thenEvt}\ (s\ tid)\ (\lambda x.\ f\ x\ tid) \\
\mathsf{retrySTM} & :: & \mathsf{STM}\ \alpha \\
\mathsf{retrySTM} & = & \lambda tid.\ \mathsf{neverEvt} \\
\mathsf{orElseSTM} & :: & \mathsf{STM}\ \alpha \to \mathsf{STM}\ \alpha \to \mathsf{STM}\ \alpha \\
\mathsf{orElseSTM}\ s_1\ s_2 & = & \lambda tid.\ \mathsf{chooseEvt}\ (\mathsf{s}_1\ tid)\ (\mathsf{s}_2\ tid) \\
\\
\mathsf{TVar} & :: & \star \to \star \\
\mathsf{TVar}\ \alpha & = & \mathsf{Chan}\ \mathsf{TId} \times \mathsf{Chan}\ \alpha \times \mathsf{Chan}\ \alpha \\
\\
\mathsf{readTVar} & :: & \mathsf{TVar}\ \alpha \to \mathsf{STM}\ \alpha \\
\mathsf{readTVar}\ (tc, rd, wr) & = & \lambda tid.\ \mathsf{thenEvt}\ (\mathsf{sendEvt}\ tc\ tid)\ (\lambda().\ \mathsf{recvEvt}\ rd) \\
\mathsf{writeTVar} & :: & \mathsf{TVar}\ \alpha \to \alpha \to \mathsf{STM}\ () \\
\mathsf{writeTVar}\ (tc, rd, wr)\ x & = & \lambda tid.\ \mathsf{thenEvt}\ (\mathsf{sendEvt}\ tc\ tid)\ (\lambda().\ \mathsf{sendEvt}\ wr\ x)
\end{array}
$$

Figure 14: STM Haskell Encoding (I)

resulting Evt. Likewise, the encoding of the monadic operations are quite simple: following the definition of a reader monad, it passes the thread identifier to every constituent STM computation.

A transactional variable is represented as a tuple of three channels: a thread identifier channel (tc), a read channel ((rd), and a write channel ((wr). When a thread in an atomic transaction wishes to read from a transactional variable, it sends its thread identifier along tc and then receives from rd. Likewise, when a thread wishes to write to a transactional variable, it sends its thread identifier along tc and then sends the new value along wr.

All of the interesting action happens in the thread that services a transactional variable, which is spawned when a transactional variable is created (see Figure 15. The server thread is comprised of two nested loops: loopIO and loopEvt. The loopIO is an IO computation that carries the state of the variable between atomic transactions. The loopEvt is a synchronization action that carries the state of the variable through a single atomic transaction. The serve event services a single read or write of the variable, returning the new value of the variable and the thread identifier of the thread that it serviced. The synchronization within loopIO first services a single read or write, which establishes the identifier of a thread that wishes to atomically access this variable, and then enters the loopEvt. The synchronization described by loopEvt chooses between servicing another request and completing the synchronization by returning the final value of the variable. If the loopEvt services another request, it further verifies that the serviced thread is the same as the thread that first accessed the variable. If the serviced thread differs, then the loopEvt transitions to a neverEvt. Since neverEvt may never appear in a “good” terminal configuration for a set of events, such a transition will never be taken during a successful synchronization. Hence, only a single thread will access the variable during a transaction.

Note that when a thread performs atomic, all of the server threads for the variables it accesses are required to synchronize.

```
newTVar   ::  α → IO (TVar α)
newTVar x  =  bindIO newChan (λtc.
              bindIO newChan (λrd.
              bindIO newChan (λwr.
              let serve :: α → Evt (α × TId)
                  serve x = thenEvt (recvEvt tc) (λtid′.
                                chooseEvt (mapEvt (λ(). (x, tid′)) (sendEvt rd x))
                                          (mapEvt (λx′. (x′, tid′)) (recvEvt wr)) )

                  loopEvt :: α → TId → Evt α
                  loopEvt x tid = chooseEvt (thenEvt (serve x) (λ(x′, tid′).
                                               if tidEq tid tid′
                                                  then loopEvt x′ tid′
                                                  else neverEvt ))
                                             (alwaysEvt x)

                  loopIO :: α → IO ()
                  loopIO x = bindIO (sync (thenEvt (serve x) (λ(x′, tid′).
                                                loopEvt x′ tid′ ))) (λx′.
                                loopIO x′ ) in
              bindIO (spawn (loopIO x)) (λ_.
              returnIO (tc, rd, wr)
```

Figure 15: STM Haskell Encoding (II)

8 Expressiveness of CFP

The previous sections have demonstrated that CFP contains all the expressive power of CML and of STM Haskell. A second question is whether CFP contains *more* expressive power than CML.

One of the fundamental results about the expressivity of CML is the following theorem:

Theorem 2 (CML Expressivity)

Given the standard CML event combinators and an n-way rendezvous base-event constructor, one cannot *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value). [16]*

For CML, which provides two-way rendezvous primitives (`sendEvt` and `recvEvt`), this means that it is impossible to construct an event-valued implementation of three-way rendezvous. Hence, it is impossible to implement the following signature

```
signature TRICHAN =
  sig
    type 'a trichan
    val trichannel : unit -> 'a trichan
    val swapEvt : 'a trichan * 'a -> ('a * 'a) event
  end
```

where the intended the semantics of `swapEvt` is to exchange messages among *three* synchronizing processes.

It turns out that CFP is strictly more expressive than CML:

Theorem 3 (CFP Expressivity)

Given the standard CFP event combinators and an n-way rendezvous base-event constructor, one can *implement an $(n+1)$-way rendezvous operation abstractly (i.e., as an event value).*

```
     Trichan   ::  $\star \to \star$
   Trichan $\alpha$   =   Chan (Chan $\alpha \times$ Chan $(\alpha \times \alpha)$)

    swapEvt   ::  Trichan $\alpha \to \alpha \to$ Evt $(\alpha \times \alpha)$
 swapEvt $t$ $x$   =   thenEvt (recvEvt $t, \lambda(c_{in}, c_{out})$.
               thenEvt (sendEvt $c_{in}$ $x, \lambda()$.
                        recvEvt $c_{out}$))

  newTrichan   ::  IO (Trichan $\alpha$)
  newTrichan   =   let loop $t$ = bindIO newChan ($\lambda c^1_{in}$.
                              bindIO newChan ($\lambda c^1_{out}$.
                              bindIO newChan ($\lambda c^2_{in}$.
                              bindIO newChan ($\lambda c^2_{out}$.
                              bindIO newChan ($\lambda c^3_{in}$.
                              bindIO newChan ($\lambda c^3_{out}$.
                              bindIO (sync (thenEvt (sendEvt $t$ ($c^1_{in}, c^1_{out}$)) ($\lambda()$.
                                            thenEvt (sendEvt $t$ ($c^2_{in}, c^2_{out}$)) ($\lambda()$.
                                            thenEvt (sendEvt $t$ ($c^3_{in}, c^3_{out}$)) ($\lambda()$.
                                            thenEvt (recvEvt $c^1_{in}$) ($\lambda x^1$.
                                            thenEvt (recvEvt $c^2_{in}$) ($\lambda x^2$.
                                            thenEvt (recvEvt $c^3_{in}$) ($\lambda x^3$.
                                            thenEvt (sendEvt $c^1_{out}$ ($x^2, x^3$)) ($\lambda()$.
                                            thenEvt (sendEvt $c^2_{out}$ ($x^3, x^1$)) ($\lambda()$.
                                            thenEvt (sendEvt $c^3_{out}$ ($x^1, x^2$)) ($\lambda()$.
                                            alwaysEvt ()))))))))))))
                                     ($\lambda()$. loop $t$))))))) in
               bindIO newChan ($\lambda t$. bindIO (spawn (loop $t$)) ($\lambda()$. unitIO $t$))
```

Figure 16: The Trichan Abstraction

We demonstrate this theorem by providing an implementation of the Trichan abstraction (see Figure 16).

A Trichan is implemented as a channel carrying pairs of in and out channels. The swapEvt combinator is fairly straightforward. The thread wishing to synchronize on the swapEvt receives a pair of in and out channels from the Trichan. It sends on the in channel the value is wishes to provide and then receives on the out channel the two other values. As noted previously, the semantics of the thenEvt combinator ensure that all of the communications succeed in sequence.

The newTrichan action creates a new channel and spawns a thread to facilitate the exchange of values. The helper thread prepares six channels and then enter a synchronization. In the synchronization, it sends out three pairs of channels, then receives three values on the in channels, and finally sends out the appropriate pairs of values on the out channels. Finally, the thread recurses in order to service future swaps.

Note that four threads are required to synchronize in order to accomplish the exchange: the helper thread and the three threads synchronizing on swapEvt event values.

It is worth noting the reason that the above implementation does not suffice for Concurrent ML. The fundamental difficulty is that (from the client's point of view) the protocol requires three communications to accomplish the exchange. However, in Concurrent ML, one of these communications must be choosen as the commit point for the protocol. Taking the first communication as the commit point does not suffice, as the client thread may rendezvous with the helper thread (successfully synchronizing on the commit point), but then block waiting for other threads to swap. Taking the last communication as the commit point does not suffice when the event is in a `choose` combinator, as the client may perform the first two communications (thereby enabling two other threads to swap) but then fail to rendezvous at the third communication, by taking another alternative in the select. This breaks the abstraction, because the other two threads cannot know that the thread received their swap values. Taking the middle communication as the commit point does not suffice for a similar reason.

9 Variations

In this section, we consider two simple variations on CFP.

9.1 Channel Creation

Our first variation takes inspiration from the software transactional memory (STM) extension of Concurrent Haskell [2] (see Section 10). In that concurrency language, the creation of a new transactional memory cell yields a computation in the `STM` monad, rather than in the `IO` monad.

There do not appear to be any significant problems with giving newChan the typing rule:

$$\frac{}{\Gamma \vdash \mathsf{newChan} : \mathsf{Evt}\ (\mathsf{Chan}\ \tau)}$$

this allows channels to be created as part of event synchronization. In terms of the semantics, we would simply remove the NewChan rule from the concurrent evaluation and add a synchronous evaluation transition of the form:

$$\textsc{NewChan}\quad \frac{\kappa' \text{ fresh}}{(M^{Evt}[\mathsf{newChan}]) \rightsquigarrow_1 (M^{Evt}[\mathsf{alwaysEvt}\ \kappa'])}$$

Note, however, that the freshness of κ' is with respect to the entire program state. Furthermore, note that since fresh channel names are chosen non-deterministically, then even in the "djinnish" semantics, we may freely allocate new channels and later abort the event synchronization; the allocation of these channels cannot be observed unless the event synchronizes successfully.

```
       Trichan    ::  $\star \to \star$
     Trichan $\alpha$  =   Chan (Chan $\alpha \times$ Chan $(\alpha \times \alpha)$)

    newTrichan    ::  Evt (Trichan $\alpha$)
    newTrichan    =   newChan

       swapEvt    ::  Trichan $\alpha \to \alpha \to$ Evt $(\alpha \times \alpha)$
   swapEvt $t$ $x$    =   let leader = thenEvt newChan ($\lambda c^1_{in}$.
                                   thenEvt newChan ($\lambda c^1_{out}$.
                                   thenEvt newChan ($\lambda c^2_{in}$.
                                   thenEvt newChan ($\lambda c^2_{out}$.
                                   thenEvt (sendEvt $t$ $(c^1_{in}, c^1_{out})$). ($\lambda()$.
                                   thenEvt (sendEvt $t$ $(c^2_{in}, c^2_{out})$). ($\lambda()$.
                                   thenEvt (recvEvt $c^1_{in}$) ($\lambda x^1$.
                                   thenEvt (recvEvt $c^2_{in}$) ($\lambda x^2$.
                                   thenEvt (sendEvt $c^1_{out}$ $(x^2, x)$) ($\lambda()$.
                                   thenEvt (sendEvt $c^2_{out}$ $(x, x^1)$) ($\lambda()$.
                                   alwaysEvt $(x^1, x^2)$ )))))))))) in
                      let client = thenEvt (recvEvt $t$) ($\lambda(c_{in}, c_{out})$.
                                   thenEvt (sendEvt $c_{in}$ $x$) ($\lambda()$.
                                          (recvEvt $c_{out}$) )) in
                      chooseEvt leader client
```

Figure 17: The Revised Trichan Abstraction (I)

```
       Trichan    ::  $\star \to \star$
     Trichan $\alpha$  =   Chan ($\alpha \times$ Chan $(\alpha \times \alpha)$)

    newTrichan    ::  Evt (Trichan $\alpha$)
    newTrichan    =   newChan

       swapEvt    ::  Trichan $\alpha \to \alpha \to$ Evt $(\alpha \times \alpha)$
   swapEvt $t$ $x$    =   let leader = thenEvt (recvEvt $t$) ($\lambda(x^1, c^1_{rep})$.
                                   thenEvt (recvEvt $t$) ($\lambda(x^2, c^2_{rep})$.
                                   thenEvt (sendEvt $c^1_{rep}$ $(x^2, x)$) ($\lambda()$.
                                   thenEvt (sendEvt $c^2_{rep}$ $(x, x^1)$) ($\lambda()$.
                                   alwaysEvt $(x^1, x^2)$ )))) in
                      let client = thenEvt (newChan) ($\lambda c_{rep}$.
                                   thenEvt (sendEvt $t$ $(x, c_{rep})$) ($\lambda()$.
                                          (recvEvt $c_{rep}$) )) in
                      chooseEvt leader client
```

Figure 18: The Revised Trichan Abstraction (II)

9.1.1 Expressiveness of CFP

Recall that our implementation of the Trichan abstraction in Section 8 required the existence of a helper thread to facilitate the exchange of values. We may easily revise the implementation for this variation by moving the channel creation actions into the synchronization as channel creation events.

However, we may also revise the implementation to do away with the helper thread completely. In this implementation, one of the threads synchronizing on a swapEvt is non-deterministically chosen to act as the protocol leader (see Figure 17). Furthermore, like channel creation, tri-channel creation can be exported as an event.

We may yet further revise the implementation so that the clients create reply channels as part of their synchronization action, which reduces the number of channels and communications required to perform the swap (see Figure 18).

9.2 Synchronous Evaluation

Our second variation takes inspiration from the event matching relation in the semantics of CML [16]. One interesting difference between our synchronous evaluation relation of Section 5.3.2 and the event matching relation in the semantics of CML is that the latter has no rule analogous to the former's SUBSET rule. Rather, the k-way event matching relation must terminate in a k-way rendezvous primitive.

We could eliminate the SUBSET rule from our synchronous evaluation relation by adopting the following rules:

$$\textsc{EvtEval}\quad \frac{E[e] \hookrightarrow E[e']}{(M^{Evt}[E[e]], e_2, \ldots, e_k) \leadsto_k (M^{Evt}[E[e']], e_2, \ldots, e_k)}$$

$$\textsc{EvtBind}\quad \frac{}{(M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2], e_2, \ldots, e_k) \leadsto_k (M^{Evt}[v_2\ v_1], e_2, \ldots, e_k)}$$

$$\textsc{EvtPlus1}\quad \frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2], e_2, \ldots, e_k) \leadsto_k (M^{Evt}[v_1], e_2, \ldots, e_k)}$$

$$\textsc{EvtPlus2}\quad \frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2], e_2, \ldots, e_k) \leadsto_k (M^{Evt}[v_2], e_2, \ldots, e_k)}$$

$$\textsc{EvtSendRecv}\quad \frac{}{(M_1^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\mathsf{recvEvt}\ \kappa]) \leadsto_2 (M_1^{Evt}[\mathsf{alwaysEvt}\ ()], M_2^{Evt}[\mathsf{alwaysEvt}\ v])}$$

$$\textsc{Permutation}\quad \frac{p \in \mathrm{Perm}_k \qquad (e_{p(1)}, \ldots, e_{p(k)}) \leadsto_k (e'_1, \ldots, e'_k)}{(e_1, \ldots, e_k) \leadsto_k (e'_{\overline{p}(1)}, \ldots, e'_{\overline{p}(k)})}$$

A consequence of these rules is that if a thread synchronizes on an event that includes channel communication, then the thread may only synchronize with exactly one other thread (that synchronizes on the matching channel communication).

While this variation is closer to the event matching relation in the semantics of Concurrent ML, note that the Evt type constructor continues to form a monad-with-plus.

9.2.1 Expressiveness of CFP

Note that this variation does not admit the implementation of the Trichan abstraction, since all implementations require the simultaneous synchronization of more than two threads. Likewise, this variation does not admit the encoding of STM Haskell of Section 7, since that encoding required the thread performing atomic and all the server threads for the variables accessed during the transaction to synchronize.

On the other hand, we believe that this variation still allows interesting programs that are not as conveniently expressed in Concurrent ML. As we have noted before, Concurrent ML requires on communication in a protocol to be designated as the commit point for the protocol. This variation obviates the need to designate a commit point when the entire protocol is with one other thread; the protocol will be chosen if *all* of the communications succeed with a similar sequence of complementary communications in the other thread.

10 Related Work

The the UniForM Workbench [6, 17] is a Concurrent Haskell extension that provides a library of abstract data types for shared memory and message passing communication. The message passing model is very similar to that of Concurrent ML. Russell [17] describes an implementation of events in Concurrent Haskell. The implementation provides events with the following interface:

```
data Event a

sync :: Event a -> IO a
(>>>=) :: Event a -> (a -> IO b) -> Event b
computeEvent :: IO (Event a) -> Event a
(+>) :: Event a -> Event a -> Event a

never :: Event a
always :: IO a -> Event a

instance Monad Event where
  (>>=) event1 getEvent2 = event1 >>>= (\ val -> sync (getEvent2 val))
  return val = always (return val)
```

A significant difference with respect to Concurrent ML is the fact that the choice operator +> is asymmetric; it is biased towards the first event. Although the interface makes Event an instance of the Monad typeclass, the author points out that events do not strictly form a monad, since return is not a left identity.

We may see that the interface above is closely related to our encoding of Concurrent ML in Section 6.1. The computeEvent operator is equivalent to our guardCMLEvt operator, providing pre-synchronous actions. The >>>= operator is equivalent to our wrapCMLEvt operator, providing post-synchronous actions. The always operator turns a post-synchronous action into an event; hence, the implementation of return in the instantiation of Event as a Monad requires a return in the IO monad.

Panangaden and Reppy [12] discuss the algebraic structure of first-class events and the extent to which they form a monad. Our analysis in Section 4 follows this work closely, but significantly expands upon the extend to which events (do not) form a monad-with-plus.

Alan Jeffrey [4, 5, 1] has given a denotational semantics of CML using (variants) of the ideas from Moggi's computational monad program [9, 10]. Further comparison with the work of Jeffrey is required, but a distinguishing characteristic appears to be the use of a single computation type. In contrast, our concurrency language in Section 5 has two types that denote latent computations.

We discussed the software transactional memory (STM) extension of Concurrent Haskell [2] in Section 7.

11 Future Work

The most obvious unanswered question is whether there exists an efficient implementation of the concurrency language presented in Section 5. Clearly the expressivity result of Section 8 indicates that an implementation will be more complicated than the implementaion of CML. Nonetheless, there are some reasons that may provide some hope for implementation.

First, note that in the "djinnish" semantics, even though a synchronization may be repeatedly initialized and aborted before committing, it's IO continuation (the M^{IO} components of a synchronizing thread collection) is only ever invoked once. That is, as we noted earlier, no irrevocable effects from the IO monad need ever be undone.

Second, recall that we may interpret synchronous evaluation of events as a non-deterministic search. In the absense of channel communication, we could implement this search via parallel evaluation of choice alternatives. Adding channel communication complicates the search. Clearly, we need to allow tentative communication along channels; however, the *all-or-nothing* property of the Evt monad requires that such tentative communication must be unobservable.

Third, we noted the similarities of CFP and the STM extension to Concurrent Haskell [2]. The existence of the efficient implementation of software transaction memory for Haskell offers some hope for finding an efficient implementation of CFP.

As a very rough proposal, we are considering pursuing an approach like the following. Individual threads that perform a sync action are required to enter a transaction to govern their synchronization. When channel communication occurs between two threads, then each of those threads must be in a sync transaction; the effect of the communication is to *merge* their sync transactions so that the two thread synchronizations now abort or commit together. At the same time, the merged transaction allows the two threads to see the results of the communcation operation.

We note that there are interesting questions about progress and fairness in an implementation of this concurrency language. With regard to progress, we would expect that an implementation of CFP should satisfy the property that if a successful synchronization exists in the angelic semantics, then the implementation will (in a finite time) also perform the successful synchronization. With regard to fairness, consider, for example, the following event:

```
let loop $n$ = if $m \leq 0$ then alwaysEvt true else loop $(n-1)$ in
choose (alwaysEvt false) (loop 10000)
```

The semantics of CFP suggest that synchronizing on this event may reasonably return either true or false, yet in an efficient implementation, we would almost certainly expect it to always (and immediately) return false.

We also note that the variation discussed in Section 9.2 might prove more ammenable to efficient implementation. The advantage of this variation is that the "search" for successful synchronizations need only be limited to pairs of threads, rather than arbitrary collections of threads.

Finally, we plan on investigating applications of the concurrency calculus. Clearly, more powerful abstractions may be designed and implemented in this language. Also, we believe that the *all-or-nothing* property of event synchronization obviates the need for the `withNack` combinator in some communication protocols.

References

[1] William Ferreira, Matthew Hennessy, and Alan Jeffrey. A theory of weak bisimulation for core CML. In *ICFP'96: Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 201–212. ACM Press, September 1996.

[2] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60. ACM Press, June 2005.

[3] Ralf Hinze. Deriving backtracking monad transformers (functional pearl). In *ICFP'00: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 186–197. ACM Press, September 2000.

[4] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *LICS'95: Proceedings of the Tenth IEEE Symposium on Logic in Computer Science*, pages 255–264, San Diego, CA, June 1995. IEEE Computer Society Press.

[5] Alan Jeffrey. A fully abstract semantics for a nondeterministic functional language with monadic types. In *MFPS'95: Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, LA, March 1995. Elsevier.

[6] Einar Karlsen. The UniForM concurrency toolkit and its extensions to Concurrent Haskell. In *GFPW'97: Proceedings of the Glasgow Functional Programming Workshop*, September 1997.

[7] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *ICFP'05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 192–203. ACM Press, September 2005.

[8] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17(3):348–375, 1978.

[9] Eugino Moggi. Computational lambda calculus and monads. In *LICS'89: Proceedings of the Fourth IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, August 1989. IEEE Computer Society Press.

[10] Eugino Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, January 1991.

[11] MonadPlus, 2005. `http://www.haskell.org/hawiki/MonadPlus`.

[12] Prakash Panangaden and John Reppy. The essence of concurrent ML. In Flemming Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation, and Application*, Springer Monographs in Computer Science. Springer-Verlag, 1997.

[13] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In T. Hoare, B. Broy, and R. Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series: Computer & Systems Sciences.* IOS Press, 2001.

[14] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.

[15] John Reppy. *Higher-Order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, 1992. Available as Technical Report TR 92-1285.

[16] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[17] George Russell. Events in Haskell, and how to implement them. In *ICFP'01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 157–168. ACM Press, September 2001.

[18] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

A Dynamic Semantics

A.1 Angelic

A.1.1 Sequential Evaluation ($e \hookrightarrow e'$)

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [] \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ e_2 \mid E_1\ e_2 \mid v_1\ E_2 \mid \pi_i\ E$$

$$\frac{}{E[\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2] \hookrightarrow E[e_2[v_1/x]]}$$

$$\frac{}{E[\mu f.x.e] \hookrightarrow E[\lambda x.e[\mu f.x.e/f]]}$$

$$\frac{\delta(\mathfrak{b}, v) = v'}{E[\mathfrak{b}\ v] \hookrightarrow E[v']}$$

$$\frac{}{E[(\lambda x.e)\ v] \hookrightarrow E[e[v/x]]}$$

$$\frac{}{E[\pi_i\ (v_1, \ldots, v_n)] \hookrightarrow E[v_i]}$$

A.1.2 Synchronous Evaluation ($(e_1, \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_k)$)

$$\textit{Synchronous Evaluation Contexts} \quad M^{Evt} \quad ::= \quad [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$$

EvtEval
$$\frac{E[e] \hookrightarrow E[e']}{(M^{Evt}[E[e]]) \rightsquigarrow_1 (M^{Evt}[E[e']])}$$

EvtBind
$$\frac{}{(M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2\ v_1])}$$

EvtPlus1
$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_1])}$$

EvtPlus2
$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2])}$$

EvtSendRecv
$$\frac{}{(M_1^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\mathsf{recvEvt}\ \kappa]) \rightsquigarrow_2 (M_1^{Evt}[\mathsf{alwaysEvt}\ ()], M_2^{Evt}[\mathsf{alwaysEvt}\ v])}$$

Permutation
$$\frac{p \in \mathrm{Perm}_k \qquad (e_{p(1)}, \ldots, e_{p(k)}) \rightsquigarrow_k (e'_1, \ldots, e'_k)}{(e_1, \ldots, e_k) \rightsquigarrow_k (e'_{\overline{p}(1)}, \ldots, e'_{\overline{p}(k)})}$$

Subset
$$\frac{1 \leq j \leq k \qquad (e_1, \ldots, e_j) \rightsquigarrow_j (e'_1, \ldots, e'_j)}{(e_1, \ldots, e_j, e_{j+1} \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_j, e_{j+1} \ldots, e_k)}$$

A.1.3 Concurrent Evaluation ($\mathcal{T} \xrightarrow{a} \mathcal{T}'$)

$$\textit{Concurrent Evaluation Contexts} \quad M^{IO} \quad ::= \quad [] \mid \mathsf{bindIO}\ M_1^{IO}\ v_2$$

$$\textit{Actions} \quad a \quad ::= \quad ?c \mid !c \mid \epsilon$$

$$\begin{array}{llll} \textit{Concurrent Threads} & T & ::= & \langle \theta, e \rangle \\ \textit{Concurrent Thread Groups} & \mathcal{T} & ::= & \{T, \ldots\} \end{array}$$

IOEval

$$\frac{E[e] \hookrightarrow E[e']}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e]] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e']] \rangle\}}$$

IOBind

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{bindIO}\ (\mathsf{unitIO}\ v_1)\ v_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1] \rangle\}}$$

IOGetChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}] \rangle\} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c] \rangle\}}$$

IOPutChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c] \rangle\} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()] \rangle\}}$$

Spawn

$$\frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{spawn}(v)] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta'] \rangle, \langle \theta', v \rangle\}}$$

GetTId

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getTId}] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta] \rangle\}}$$

NewChan

$$\frac{\kappa' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{newChan}] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \kappa'] \rangle\}}$$

Sync

$$\frac{(v_1, \ldots, v_k) \leadsto_k^* (\mathsf{alwaysEvt}\ v_1', \ldots, \mathsf{alwaysEvt}\ v_k')}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1'] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k'] \rangle\}}$$

A.2 Djinnish

A.2.1 Sequential Evaluation ($e \hookrightarrow e'$)

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [] \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ e_2 \mid E_1\ e_2 \mid v_1\ E_2 \mid \pi_i\ E$$

$$\frac{}{E[\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2] \hookrightarrow E[e_2[v_1/x]]}$$

$$\frac{}{E[\mu f.x.e] \hookrightarrow E[\lambda x.e[\mu f.x.e/f]]}$$

$$\frac{\delta(\mathfrak{b}, v) = v'}{E[\mathfrak{b}\ v] \hookrightarrow E[v']}$$

$$\frac{}{E[(\lambda x.e)\ v] \hookrightarrow E[e[v/x]]}$$

$$\frac{}{E[\pi_i\ (v_1, \ldots, v_n)] \hookrightarrow E[v_i]}$$

A.2.2 Synchronous Evaluation ($(e_1, \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_k)$)

$$\textit{Synchronous Evaluation Contexts} \quad M^{Evt} \quad ::= \quad [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2$$

EvtEval

$$\frac{E[e] \hookrightarrow E[e']}{(M^{Evt}[E[e]]) \rightsquigarrow_1 (M^{Evt}[E[e']])}$$

EvtBind

$$\frac{}{(M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2\ v_1])}$$

EvtPlus1

$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_1])}$$

EvtPlus2

$$\frac{}{(M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2]) \rightsquigarrow_1 (M^{Evt}[v_2])}$$

EvtSendRecv

$$\frac{}{(M_1^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M_2^{Evt}[\mathsf{recvEvt}\ \kappa]) \rightsquigarrow_2 (M_1^{Evt}[\mathsf{alwaysEvt}\ ()], M_2^{Evt}[\mathsf{alwaysEvt}\ v])}$$

Permutation

$$\frac{p \in \mathrm{Perm}_k \qquad (e_{p(1)}, \ldots, e_{p(k)}) \rightsquigarrow_k (e'_1, \ldots, e'_k)}{(e_1, \ldots, e_k) \rightsquigarrow_k (e'_{\overline{p}(1)}, \ldots, e'_{\overline{p}(k)})}$$

Subset

$$\frac{1 \leq j \leq k \qquad (e_1, \ldots, e_j) \rightsquigarrow_j (e'_1, \ldots, e'_j)}{(e_1, \ldots, e_j, e_{j+1} \ldots, e_k) \rightsquigarrow_k (e'_1, \ldots, e'_j, e_{j+1} \ldots, e_k)}$$

A.2.3 Concurrent Evaluation ($\mathcal{T};\mathcal{S} \xrightarrow{a} \mathcal{T}';\mathcal{S}'$)

$$\textit{Concurrent Evaluation Contexts} \quad M^{IO} ::= [] \mid \mathsf{bindIO}\ M_1^{IO}\ v_2$$

$$\textit{Actions} \quad a ::= ?c \mid !c \mid \epsilon$$

$$\begin{array}{lll} \textit{Concurrent Threads} & T & ::= \langle \theta, e \rangle \\ \textit{Concurrent Thread Groups} & \mathcal{T} & ::= \{T, \ldots\} \end{array}$$

$$\begin{array}{lll} \textit{Synchronization IDs} & \phi & \in \textit{SId} \\ \textit{Synchronizing Threads} & S & ::= \langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1' \rangle, \ldots, \langle \theta_k, M_k^{IO}, v_k, e_k' \rangle\rangle\rangle \\ \textit{Synchronizing Thread Groups} & \mathcal{S} & ::= \{S, \ldots\} \end{array}$$

$$\textit{Concurrent Evaluation Contexts} \quad M^{IO} ::= [] \mid \mathsf{thenEvt}\ M_1^{IO}\ v_2$$

IOEval

$$\frac{E[e] \hookrightarrow E[e']}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e]] \rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e']] \rangle\}; \mathcal{S}}$$

IOBind

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{bindIO}\ (\mathsf{unitIO}\ v_1)\ v_2] \rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1] \rangle\}; \mathcal{S}}$$

IOGetChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}] \rangle\}; \mathcal{S} \xrightarrow{?c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c] \rangle\}; \mathcal{S}}$$

IOPutChar

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c] \rangle\}; \mathcal{S} \xrightarrow{!c} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()] \rangle\}; \mathcal{S}}$$

Spawn

$$\frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{spawn}\ v] \rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta'] \rangle, \langle \theta', v \rangle\}; \mathcal{S}}$$

GetTId

$$\frac{}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getTId}] \rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta] \rangle\}; \mathcal{S}}$$

NewChan

$$\frac{\kappa' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{newChan}] \rangle\}; \mathcal{S} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \kappa'] \rangle\}; \mathcal{S}}$$

SyncInit

$$\frac{\phi' \text{ fresh}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S} \\ \quad \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \phi', \langle\langle \theta_1, M_1^{IO}, v_1, v_1\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, v_k\rangle\rangle\rangle\}\end{array}}$$

SyncStep

$$\frac{(e_1', \ldots, e_k') \rightsquigarrow_k (e_1'', \ldots, e_k'')}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1''\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k''\rangle\rangle\rangle\}\end{array}}$$

SyncCommit

$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, \mathsf{alwaysEvt}\ v_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, \mathsf{alwaysEvt}\ v_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{unitIO}\ v_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{unitIO}\ v_k']\rangle\}; \mathcal{S}\end{array}}$$

SyncAbort

$$\frac{}{\begin{array}{l}\mathcal{T}; \mathcal{S} \uplus \{\langle \phi, \langle\langle \theta_1, M_1^{IO}, v_1, e_1'\rangle, \ldots \langle \theta_k, M_k^{IO}, v_k, e_k'\rangle\rangle\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\mathsf{sync}\ v_1]\rangle, \ldots, \langle \theta_k, M_k^{IO}[\mathsf{sync}\ v_k]\rangle\}; \mathcal{S}\end{array}}$$

A.3 Dwarvish

A.3.1 Sequential Evaluation ($e \hookrightarrow e'$)

$$\textit{Evaluation Contexts} \quad E \quad ::= \quad [] \mid \mathsf{let}\ x = E_1\ \mathsf{in}\ e_2 \mid E_1\ e_2 \mid v_1\ E_2 \mid \pi_i\ E$$

$$\frac{}{E[\mathsf{let}\ x = v_1\ \mathsf{in}\ e_2] \hookrightarrow E[e_2[v_1/x]]}$$

$$\frac{}{E[\mu f.x.e] \hookrightarrow E[\lambda x.e[\mu f.x.e/f]]}$$

$$\frac{\delta(\mathfrak{b}, v) = v'}{E[\mathfrak{b}\ v] \hookrightarrow E[v']}$$

$$\frac{}{E[(\lambda x.e)\ v] \hookrightarrow E[e[v/x]]}$$

$$\frac{}{E[\pi_i\ (v_1, \ldots, v_n)] \hookrightarrow E[v_i]}$$

A.3.2 Synchronous and Concurrent Evaluation ($\mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \xrightarrow{a} \mathbb{K}'; \mathbb{L}'; \mathbb{B}'; \mathcal{T}'$)

$$\begin{array}{lll} \textit{Synchronous Evaluation Contexts} & M^{Evt} & ::= \quad [] \mid \mathsf{thenEvt}\ M_1^{Evt}\ v_2 \\ \textit{Concurrent Evaluation Contexts} & M^{IO} & ::= \quad [] \mid \mathsf{bindIO}\ M_1^{IO}\ v_2 \end{array}$$

$$\begin{array}{lll} \text{Boolean} & \mathscr{b} & ::= \quad \mathit{true} \mid \mathit{false} \\ \text{Boolean Flag Map} & \mathbb{B} & ::= \quad \{\mathbb{b} \mapsto \mathscr{b}, \ldots\} \end{array}$$

$$\mathbb{b} \quad :: \quad \mathsf{type\ B} \ = \ \mathsf{TVar\ Bool}$$

$$\text{Potential Commit List Map} \quad \mathbb{L} \quad ::= \quad \{\mathbb{l} \mapsto \{\langle \rho, \mathbb{b}_\rho \rangle, \ldots\}, \ldots\}$$

$$\mathbb{l} \quad :: \quad \mathsf{type\ L} \ = \ \mathsf{TVar\ (List\ (P, B))}$$

$$\begin{array}{lll} \text{Path Element} & \overline{\rho} & ::= \quad \mathsf{Left} \mid \mathsf{Right} \mid \mathsf{Send}(\theta, \mathbb{b}_\theta, \rho_\theta, \mathbb{l}_\theta; \mathbb{l}) \mid \mathsf{Recv}(\theta, \mathbb{b}_\theta, \rho_\theta, \mathbb{l}_\theta; \mathbb{l}) \\ \text{Path} & \rho & ::= \quad \bullet \mid \overline{\rho}{:}\rho \end{array}$$

$$\overline{\rho} \quad :: \quad \mathsf{type\ PE} \ = \ \mathsf{Left} \mid \mathsf{Right} \mid \mathsf{Send(((ThreadId, B, P), L), L)} \mid \mathsf{Recv(((ThreadId, B, P), L), L)}$$

$$\rho \quad :: \quad \mathsf{type\ P} \ = \ \mathsf{List\ PE}$$

$$\begin{array}{lll} \text{Channel Send Queue} & \mathscr{q}_{\mathsf{S}} & ::= \quad \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{l}, v, M^{Evt}, M^{IO} \rangle, \ldots\} \\ \text{Channel Receive Queue} & \mathscr{q}_{\mathsf{R}} & ::= \quad \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{l}, M^{Evt}, M^{IO} \rangle, \ldots\} \\ \text{Channel Map} & \mathbb{K} & ::= \quad \{\kappa \mapsto \langle \mathscr{q}_{\mathsf{S}}, \mathscr{q}_{\mathsf{R}} \rangle, \ldots\} \end{array}$$

$$\begin{array}{llll} \textit{Threads} & T & ::= \ \langle \theta, e \rangle & \text{Concurrent thread} \\ & & \mid \ \langle \theta, \mathbb{b}_\theta, \rho, e, M^{IO} \rangle & \text{Synchronizing thread} \\ & & \mid \ \langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle^{?} & \text{Potential commit thread (pending commit search)} \\ & & \mid \ \langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle & \text{Potential commit thread (completed commit search)} \\ \textit{Thread Groups} & \mathcal{T} & ::= \ \{T, \ldots\} & \end{array}$$

IOEval

$$\frac{E[e] \hookrightarrow E[e']}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e]]\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[E[e']]\rangle\}}$$

IOBind

$$\frac{}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{bindIO}\ (\mathsf{unitIO}\ v_1)\ v_2]\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[v_2\ v_1]\rangle\}}$$

IOGetChar

$$\frac{}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getChar}]\rangle\} \xrightarrow{?c} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ c]\rangle\}}$$

IOPutChar

$$\frac{}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{putChar}\ c]\rangle\} \xrightarrow{!c} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ ()]\rangle\}}$$

Spawn

$$\frac{\theta' \text{ fresh}}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{spawn}(v)]\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta']\rangle, \langle \theta', v\rangle\}}$$

GetTId

$$\frac{}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{getTId}]\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \theta]\rangle\}}$$

NewChan

$$\frac{\kappa' \text{ fresh}}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{newChan}]\rangle\} \xrightarrow{\epsilon} \mathbb{K}[\kappa' \mapsto \langle\{\},\{\}\rangle];\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ \kappa']\rangle\}}$$

SyncInit

$$\frac{\mathbb{b}_\theta \text{ fresh}}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{sync}\ v]\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B}[\mathbb{b}_\theta \mapsto \mathit{false}];\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \bullet, v, M^{IO}\rangle\}}$$

EvtFizzle

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{true}}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, e, M^{IO}\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T}}$$

EvtEval

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{false} \qquad E[e] \hookrightarrow E[e']}{\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[E[e]], M^{IO}\rangle\} \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[E[e']], M^{IO}\rangle\}}$$

EvtBind

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{false}}{\begin{array}{l}\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2], M^{IO}\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[v_2\ v_1], M^{IO}\rangle\}\end{array}}$$

EvtPlus

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{false}}{\begin{array}{l}\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[\mathsf{chooseEvt}\ v_1\ v_2], M^{IO}\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \mathsf{Left}{:}\rho, M^{Evt}[v_1], M^{IO}\rangle, \langle \theta, \mathbb{b}_\theta, \mathsf{Right}{:}\rho, M^{Evt}[v_1], M^{IO}\rangle\}\end{array}}$$

EvtSend

$$\frac{\begin{array}{c}\mathbb{B}(\mathbb{b}_\theta) = \mathit{false} \qquad \mathbb{l} \text{ fresh} \qquad \mathbb{K}(\kappa) = \langle q_{\mathsf{S}}, q_{\mathsf{R}}\rangle \\ \begin{array}{rl}\mathcal{T}' = \bigcup\{ & \{\langle \theta, \mathbb{b}_\theta, \mathsf{Send}(\theta', \mathbb{b}_{\theta'}, \rho', \mathbb{l}'; \mathbb{l}){:}\rho, M^{Evt}[\mathsf{alwaysEvt}\ ()], M^{IO}\rangle, \\ & \langle \theta', \mathbb{b}_{\theta'}, \mathsf{Recv}(\theta, \mathbb{b}_\theta, \rho, \mathbb{l}; \mathbb{l}'){:}\rho', M^{Evt'}[\mathsf{alwaysEvt}\ v], M^{IO'}\rangle\} \\ & \quad \mid \langle \theta', \mathbb{b}_{\theta'}, \rho', \mathbb{l}', M^{Evt}, M^{IO''}\rangle \in q_{\mathsf{R}} \\ & \qquad \wedge\ \mathbb{B}(\mathbb{b}_{\theta'}) = \mathit{false} \\ & \qquad \wedge\ (\theta, \mathbb{b}_\theta, \rho) \text{ and } (\theta', \mathbb{b}_{\theta'}, \rho') \text{ are coherent } \}\end{array}\end{array}}{\begin{array}{l}\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[\mathsf{sendEvt}\ \kappa\ v], M^{IO}\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K}[\kappa \mapsto \langle q_{\mathsf{S}} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{l}, v, M^{Evt}, M^{IO}\rangle\}, q_{\mathsf{R}}\rangle];\mathbb{L}[\mathbb{l} \mapsto \{\}];\mathbb{B};\mathcal{T} \uplus \mathcal{T}'\end{array}}$$

EvtRecv

$$\frac{\begin{array}{c}\mathbb{B}(\mathbb{b}_\theta) = \mathit{false} \qquad \mathbb{l} \text{ fresh} \qquad \mathbb{K}(\kappa) = \langle q_{\mathsf{S}}, q_{\mathsf{R}}\rangle \\ \begin{array}{rl}\mathcal{T}' = \bigcup\{ & \{\langle \theta, \mathbb{b}_\theta, \mathsf{Recv}(\theta', \mathbb{b}_{\theta'}, \rho', \mathbb{l}'; \mathbb{l}){:}\rho, M^{Evt}[\mathsf{alwaysEvt}\ v'], M^{IO}\rangle, \\ & \langle \theta', \mathbb{b}_{\theta'}, \mathsf{Send}(\theta, \mathbb{b}_\theta, \rho, \mathbb{l}; \mathbb{l}'){:}\rho', M^{Evt'}[\mathsf{alwaysEvt}\ ()], M^{IO'}\rangle\} \\ & \quad \mid \langle \theta', \mathbb{b}_{\theta'}, \rho', \mathbb{l}', v', M^{Evt'}, M^{IO'}\rangle \in q_{\mathsf{S}} \\ & \qquad \wedge\ \mathbb{B}(\mathbb{b}_{\theta'}) = \mathit{false} \\ & \qquad \wedge\ (\theta, \mathbb{b}_\theta, \rho) \text{ and } (\theta', \mathbb{b}_{\theta'}, \rho') \text{ are coherent } \}\end{array}\end{array}}{\begin{array}{l}\mathbb{K};\mathbb{L};\mathbb{B};\mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, M^{Evt}[\mathsf{recvEvt}\ \kappa], M^{IO}\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K}[\kappa \mapsto \langle q_{\mathsf{S}}, q_{\mathsf{R}} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{l}, M^{Evt}, M^{IO}\rangle\}\rangle];\mathbb{L}[\mathbb{l} \mapsto \{\}];\mathbb{B};\mathcal{T} \uplus \mathcal{T}'\end{array}}$$

COMMITINIT

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{false} \qquad \mathbb{b}_\rho \text{ fresh} \qquad \mathbb{L}' = L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \rho)}{\mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathsf{alwaysEvt}\ v, M^{IO} \rangle\} \xrightarrow{\epsilon} \mathbb{K}; \mathbb{L}'; \mathbb{B} \uplus \{\mathbb{b}_\rho \mapsto \mathit{false}\}; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle^?\}}$$

$$\begin{array}{rcl}
L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \bullet) & = & \mathbb{L} \\
L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \mathsf{Left}{:}\rho') & = & L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \rho') \\
L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \mathsf{Right}{:}\rho') & = & L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \rho') \\
L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \mathsf{Send}(\_; \mathbb{l}){:}\rho') & = & L_{\rho,\mathbb{b}_\rho}(\mathbb{L}[\mathbb{l} \mapsto \mathbb{L}(\mathbb{l}) \uplus \{\langle \rho, \mathbb{b}_\rho \rangle\}], \rho') \\
L_{\rho,\mathbb{b}_\rho}(\mathbb{L}, \mathsf{Recv}(\_; \mathbb{l}){:}\rho') & = & L_{\rho,\mathbb{b}_\rho}(\mathbb{L}[\mathbb{l} \mapsto \mathbb{L}(\mathbb{l}) \uplus \{\langle \rho, \mathbb{b}_\rho \rangle\}], \rho')
\end{array}$$

COMMITSEARCH

$$\frac{\mathbb{B}' = \begin{cases} B(\mathbb{B}, \mathcal{M}) & \text{if } S_{\mathbb{L}}(\theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, \{\}) = \{\mathcal{M}, \ldots\} \\ \mathbb{B} & \text{if } S_{\mathbb{L}}(\theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, \{\}) = \{\} \end{cases}}{\mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle^?\} \xrightarrow{\epsilon} \mathbb{K}; \mathbb{L}; \mathbb{B}'; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle\}}$$

$$S_{\mathbb{L}}(\theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, \mathcal{M}) \quad = \quad \begin{cases} \{\mathcal{M}\} & \text{if } \mathcal{M}(\theta) = \langle \mathbb{b}_\theta, \mathbb{b}_\rho \rangle \\ \{\} & \text{if } \mathcal{M}(\theta) = \langle \mathbb{b}'_\theta, \mathbb{b}'_\rho \rangle \wedge (\mathbb{b}_\theta \neq \mathbb{b}'_\theta \vee \mathbb{b}_\rho \neq \mathbb{b}_\rho) \\ \{\} & \text{if } \theta \notin \mathit{dom}(\mathcal{M}) \wedge \mathbb{B}(\mathbb{b}_\theta) = \mathit{true} \\ \overline{S}_{\mathbb{L}}(\rho, \mathcal{M} \uplus \{\theta \mapsto (\mathbb{b}_\theta, \mathbb{b}_\rho)\}) & \text{if } \theta \notin \mathit{dom}(\mathcal{M}) \wedge \mathbb{B}(\mathbb{b}_\theta) = \mathit{false} \end{cases}$$

$$\begin{array}{rcl}
\overline{S}_{\mathbb{L}}(\bullet, \mathcal{M}) & = & \{\mathcal{M}\} \\
\overline{S}_{\mathbb{L}}(\mathsf{Left}{:}\rho, \mathcal{M}) & = & \overline{S}(\rho, \mathcal{M}) \\
\overline{S}_{\mathbb{L}}(\mathsf{Right}{:}\rho, \mathcal{M}) & = & \overline{S}(\rho, \mathcal{M}) \\
\overline{S}_{\mathbb{L}}(\mathsf{Send}(\theta', \mathbb{b}_{\theta'}, \_, \mathbb{l}'; \_){:}\rho, \mathcal{M}) & = & \bigcup\{\ S_{\mathbb{L}}(\theta', \mathbb{b}_{\theta'}, \rho', \mathbb{b}_{\rho'}, \mathcal{M}') \mid \mathcal{M}' \in \overline{S}_{\mathbb{L}}(\rho, \mathcal{M}) \wedge \langle \rho', \mathbb{b}_{\rho'} \rangle \in \mathbb{L}(\mathbb{l}')\ \} \\
\overline{S}_{\mathbb{L}}(\mathsf{Recv}(\theta', \mathbb{b}_{\theta'}, \_, \mathbb{l}'; \_){:}\rho, \mathcal{M}) & = & \bigcup\{\ S_{\mathbb{L}}(\theta', \mathbb{b}_{\theta'}, \rho', \mathbb{b}_{\rho'}, \mathcal{M}') \mid \mathcal{M}' \in \overline{S}_{\mathbb{L}}(\rho, \mathcal{M}) \wedge \langle \rho', \mathbb{b}_{\rho'} \rangle \in \mathbb{L}(\mathbb{l}')\ \}
\end{array}$$

$$\begin{array}{rcl}
B(\mathbb{B}, \{\}) & = & \mathbb{B} \\
B(\mathbb{B}, \mathcal{M} \uplus \{\theta \mapsto \langle \mathbb{b}_\theta, \mathbb{b}_\rho \rangle\}) & = & B(\mathbb{B}[\mathbb{b}_\theta \mapsto \mathit{true}, \mathbb{b}_\rho \mapsto \mathit{true}], \mathcal{M})
\end{array}$$

COMMITFIZZLE

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{true} \qquad \mathbb{B}(\mathbb{b}_\rho) = \mathit{false}}{\mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle\} \xrightarrow{\epsilon} \mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T}}$$

COMMITCOMMIT

$$\frac{\mathbb{B}(\mathbb{b}_\theta) = \mathit{true} \qquad \mathbb{B}(\mathbb{b}_\rho) = \mathit{true}}{\mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \uplus \{\langle \theta, \mathbb{b}_\theta, \rho, \mathbb{b}_\rho, v, M^{IO} \rangle\} \xrightarrow{\epsilon} \mathbb{K}; \mathbb{L}; \mathbb{B}; \mathcal{T} \uplus \{\langle \theta, M^{IO}[\mathsf{unitIO}\ v] \rangle\}}$$