Transactional Events

Kevin Donnelly
Boston University
kevind@cs.bu.edu

Matthew Fluet
Cornell University
fluet@cs.cornell.edu

Abstract

Concurrent programs require high-level abstractions in order to manage complexity and enable compositional reasoning. In this paper, we introduce a novel concurrency abstraction, dubbed transactional events, which combines first-class synchronous message-passing events with all-or-nothing transactions. This combination enables simple solutions to interesting problems in concurrent programming. For example, guarded synchronous receive can be implemented as an abstract transactional event, whereas in other languages it requires a non-abstract, non-modular protocol. Likewise, three-way rendezvous can also be implemented as an abstract transactional event, which is impossible using first-class events alone. Both solutions are easy to code and easy to reason about.

The expressive power of transactional events arises from a sequencing combinator whose semantics enforces an all-or-nothing transactional property – either both of the constituent events synchronize in sequence or neither of them synchronizes. This sequencing combinator, along with a non-deterministic choice combinator, gives transactional events the compositional structure of a monad-with-plus. We provide a formal semantics for and a preliminary implementation of transactional events.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language constructs and features—Concurrent programming structures; D.1.3 [*Programming Techniques*]: Concurrent programming

General Terms Languages, Design

Keywords concurrency, synchronous message passing, first-class events, transactions, monads

1. Introduction

Programming with concurrency can be an extremely difficult task. A concurrent program's inherent non-determinism makes it difficult to reason about and even harder to debug. However, concurrency has proven to be a useful tool for structuring programs, as well as an important means of improving performance. Concurrency is indispensable when implementing interactive systems that must quickly react to unpredictable and asynchronous occurrences, like user input or network activity. Concurrent execution of multiple threads also allows programs to take advantage of the presence of multiple cores or processors on a single machine.

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

Given the usefulness of concurrent programs, it is important to provide programmers with abstraction mechanisms that help manage the complexity of reasoning about them. One means of managing this complexity, exemplified by both Concurrent ML (CML) [21] and Software Transactional Memory (STM) Haskell [7], is to introduce first-class, composable operations that encapsulate a particular concurrent programming style: synchronous message passing in CML and shared-memory transactions in STM Haskell. Such first-class, composable operations allow complex thread interactions to be abstractly packaged and exported, which increases modularity and eases reasoning.

STM Haskell utilizes monadic computations as a means of structuring shared-memory transactions, which ensures that isolated sequences of accesses of shared memory are performed atomically. STM computations can be naturally composed in sequence or as alternatives, giving rise to flexible concurrency abstractions. The atomicity and isolation guarantees of STM transactions ease reasoning about interaction of concurrent threads, but preclude the use of STM transactions for thread interactions, like synchronous message passing.

Concurrent ML utilizes first-class events as a means of structuring synchronous message passing. Like STM computations, events are abstractly packaged groups of actions. However, events allow for synchronous message passing and are not isolated. Events can be naturally composed as alternatives, but sequential composition of events can be difficult to reason about. Synchronous message passing is inherently more powerful than asynchronous message passing; because of this, there are abstractions, like swap channels, that may be implemented as CML events but have no implementation as STM computations. However, the "all-or-nothing" (atomic) semantics of STM computations makes composition and reasoning easier, and can lead to greater modularity.

In this work, we explore an abstraction mechanism for concurrent programming that combines first-class synchronous events with all-or-nothing transactions. This combination allows programmers to simultaneously take advantage of the power of synchronous message passing and the modularity and composability of transactions. It also leads to abstractions that are more expressive than either CML events or STM computations. This paper makes the following contributions:

- We introduce the notion of Transactional Events (TE). These synchronous message passing abstractions overcome some limitations of CML events and allow for greater modularity in concurrent programming. The increased modularity and all-or-nothing nature of transactional events eases reasoning and makes these events better suited to composition.
- We give a formal semantics for a language, TE Haskell, which draws inspiration from both Concurrent ML and Concurrent Haskell. The language includes transactional events, concurrent threads, monadic I/O, and exceptions.

- We show the expressive power of TE Haskell with respect to other concurrency primitives by giving implementations of modular guarded receive, CML events, transactional shared memory, and 3-way synchronous rendezvous.
- We describe a prototype implementation of transactional events as a library for Haskell, using the STM Haskell extensions available in GHC.

The remainder of the paper is structured as follows. Section 2 briefly reviews Concurrent ML and Concurrent Haskell. Section 3 discusses the informal semantics of transactional events and gives some simple examples, before turning to the formal semantics of TE Haskell in Section 4. Next, we explore the expressive power of transactional events with more complex examples, including encodings of CML events and transactional shared memory. Section 6 sketches our implementation of transactional events, while Section 7 discusses related work. We conclude with some directions for future work.

2. Background

2.1 Concurrent ML

Concurrent ML is a library for Standard ML that provides threads and synchronous message passing. The CML Library [21, Appendix A] provides types for synchronous events (`'a event`) and synchronous channels (`'a chan`), as well as the following operations:

```
channel : unit -> 'a chan
sendEvt : 'a chan * 'a -> unit event
recvEvt : 'a chan -> 'a event

sync      : 'a event -> 'a
choose    : 'a event list -> 'a event
wrap      : 'a event * ('a -> 'b) -> 'b event
guard     : (unit -> 'a event) -> 'a event
withNack  : (unit event -> 'a event) -> 'a event
alwaysEvt : 'a -> 'a event
never     : 'a event
```

A value of type `'a event` is an abstract synchronous operation that returns a value of type `'a` when it is synchronized upon. An event value represents *potential* communication and synchronous actions, and is itself quiescent; its latent action is only performed when a thread synchronizes on it. The strength of first-class events is that they overcome the tension between abstraction and selective communication. Events are abstract like functions, but can participate in selection. This combination achieves a level of abstraction and modularity not found in previous concurrent languages.

We briefly describe the CML operations below:

`channel ()` creates a new synchronous channel.

`sendEvt (ch, m)` creates an event which sends message `m` on channel `ch`. This event becomes enabled (i.e., can be selected for synchronization) when communication can proceed without blocking (i.e., when there is a matching receiver), and yields `()`.

`recvEvt ch` creates an event which receives a message on channel `ch`. This event becomes enabled when communication can proceed without blocking, and yields the received message.

`sync ev` synchronizes on the event `ev`.

`choose [ev`$_1$`, ..., ev`$_n$`]` creates the event which, when synchronized on, non-deterministically chooses some enabled `ev`$_i$.

`wrap (ev, f)` creates an event which, if `ev` is selected for synchronization yielding the value `v`, evaluates and yields `f v`.

`guard f` creates an event which, at synchronization time, evaluates `f ()` to an event `ev`, and then acts as `ev`.

`withNack f` creates an event which, at synchronization time, evaluates `f nack` to an event `ev`, and then acts as `ev`. The argument `nack` is an event that becomes enabled only if an event other than `ev` is selected for synchronization.

`alwaysEvt a` creates an event which is always enabled, and yields `a`.

`never` creates an event which is never enabled.

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

Limitations of CML events The fact that all CML events must have a single commit point places a limitation on the modularity achievable with these events. Consider the example of trying to program guarded receive in CML. Given a channel `ch : 'a chan` and a guard `g : 'a -> bool`, we would like an event, `grecvEvt g ch : 'a event`, that will receive a message `x` from `ch`, but only if `g x` evaluates to `true`. Because message passing is synchronous, we cannot just receive from the channel and test the result, because the sender will complete its synchronization after the send. Guarded communication can be implemented as an event in CML, but it requires a fairly complicated protocol in which the sender and the receiver cooperate to achieve the desired behavior. In TE Haskell, guarded receive can be given a transparently correct implementation with just a few lines of code.

Requiring a protocol to implement guarded receive leads to a lack of modularity. If we decide that we want to perform a guarded receive on some existing channel, then we need to alter all of the code that sends and receives on this channel to use the channel-with-guarded-receive abstraction, although we may have required a guarded receive for only one receiver. In TE Haskell, the implementation of guarded receive is local to the receiver.

Having a single commit point also limits the expressive power of CML events. In CML, given the operations for 2-way synchronization, there is no way to implement 3-way synchronization as an event abstraction [16, 21]. As we show in Section 5.4, TE Haskell allows for abstract implementation of such 3-way synchronous operations.

2.2 Monadic I/O and Concurrent Haskell

TE Haskell is an extension of Concurrent Haskell [18], which extends Haskell with concurrency primitives. Following the Haskell tradition, Concurrent Haskell isolates the side-effect producing concurrency primitives in the I/O monad [19, 17]. Intuitively, values of type `IO a` are actions which, when performed, may do some I/O and then yield an `a`. Actions that read and write a character, `getChar` and `putChar`, are given the following types:

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

Concurrent Haskell supports multiple threads running I/O actions concurrently. I/O actions are turned into threads using the function `forkIO`:

```
forkIO :: IO a -> IO ThreadId
```

`forkIO a` spawns a new thread to perform the I/O action specified by `a` and returns the identifier of the newly spawned thread to the

caller. For example, here is a program that spawns one thread to write the character 'A' and another to write the character 'B':

```
main :: IO ()
main = do { forkIO (putChar 'A')
          ; forkIO (putChar 'B')
          ; return () }
```

Note that 'A' and 'B' are written in a non-deterministic order.

3. Transactional Events

In TE Haskell we overcome the limitations of CML's single commit point requirement by allowing event synchronization to take place in transactions that can be aborted if they do not successfully complete. Transactions are an old idea from the world of databases and have recently found use as powerful technique to ensure atomicity in concurrent programming [8, 5, 26, 7]. We adapt transactions to the setting of synchronous message passing, yielding large gains in modularity and expressiveness.

Transactions group tentative actions made during computations, which can then be committed if the transaction successfully completes or aborted otherwise. In databases, the actions are a single client's database queries; in transactional memory, the actions are a single thread's reads of and writes to shared memory; in both cases, success is a serializable schedule. Tentative actions are not observable by the rest of the system or program until the entire transaction commits, so transactions provide an *all-or-nothing* semantics.

Due to this all-or-nothing semantics, transactions provide a straightforward way of overcoming the single-commit point limitation of CML. By viewing an event synchronization as a transaction, where tentative synchronous message passing between threads merges their event synchronization transactions (so that they either commit or abort together), it becomes possible to construct sophisticated abstract synchronous operations that cannot be constructed in previous concurrent languages.

In the rest of this section we informally describe transactional events in the context of an extension of Concurrent Haskell [18]. Haskell provides an ideal setting for programming with transactional events because the monadic type system makes it easy to keep irrevocable side-effects out of transactional events, which may otherwise need to be aborted.

The basic interface for transactional events is given in Figure 1. An `Evt a` is a transactional event: an abstract synchronous operation that yields an `a` when synchronized upon. An `SChan a` is a synchronous channel used for message passing. The `sync` operation takes a transactional event to an I/O action that performs the synchronization. Note that synchronization is not a pure function, but rather depends on the state of concurrently synchronizing threads. Hence, `sync` yields an I/O action, although it does not itself perform any observable I/O.

New channels are created with `newSChan`, which is given an event type so that we may create local channels inside of transactional event synchronizations. The basic events `sendEvt ch m` and `recvEvt ch` correspond to the event that sends `m` over `ch` and the event that receives on `ch`. Message passing is synchronous, so every send must be matched by a receive; a transactional event synchronization may not successfully complete unless all of the communications are matched by complementary communications in other transactional event synchronizations that may also successfully complete. For example, the following program creates a channel, spawns a thread, receives a character on the channel from the spawned thread, and finally prints the character:

```
main = do { ch <- sync newSChan
          ; forkIO (sync (sendEvt ch 'A'))
          ; c <- sync (recvEvt ch)
          ; putChar c }
```

```
data Evt a     -- The Evt monad

sync :: Evt a -> IO a

thenEvt   :: Evt a -> (a -> Evt b) -> Evt b
alwaysEvt :: a -> Evt a
chooseEvt :: Evt a -> Evt a -> Evt a
neverEvt  :: Evt a

instance Monad Evt where
  (>>=) = thenEvt
  return = alwaysEvt
instance MonadPlus Evt where
  mplus = chooseEvt
  mzero = neverEvt

throwEvt :: Exception -> Evt a
catchEvt :: Evt a -> (Exception -> Evt a) -> Evt a

data SChan a     -- Synchronous channels
newSChan :: Evt (SChan a)
sendEvt  :: SChan a -> a -> Evt ()
recvEvt  :: SChan a -> Evt a
```

Figure 1. The `Evt` Interface

Events can be composed in sequence using the `thenEvt` combinator, which is also available as the monadic bind (`>>=`) and may be used implicitly via Haskell's `do`-notation. The event `ev 'thenEvt' f` is the event which tentatively synchronizes on the event `ev`, yielding `r`, and then synchronizes on the event `f r`. If these events cannot successfully complete in sequence, then the composed event cannot successfully complete. For example, the event which sends `0` and `1`, in sequence, over the channel `ch` is:

```
ev1 = do { sendEvt ch 0 ; sendEvt ch 1 }
```

This event may only successfully complete synchronization if both sends are successful; for example, it may synchronize if another thread is synchronizing on the following event:

```
ev2 = do { recvEvt ch ; recvEvt ch }
```

The event `alwaysEvt e` is an event which immediately yields `e` when synchronized upon. Note that `alwaysEvt` is a left and right unit of `thenEvt`; hence, `Evt` forms a monad [25], with `alwaysEvt` as the monadic unit.

Events may also be composed as non-deterministic alternatives using the `chooseEvt` combinator. The event `ev1 'chooseEvt' ev2` synchronizes as either `ev1` or `ev2`, but only commits to a choice that can successfully complete. Until such a choice can be determined, the composed event cannot successfully complete. For example, the event which chooses between sending `0` and `1` over the channel `ch` or receiving an integer once on the same channel is:

```
ev3 = (do { sendEvt ch 0 ; sendEvt ch 1 })
      'chooseEvt'
      (do { recvEvt ch ; alwaysEvt () })
```

Note that this event cannot be implemented in CML, because the first alternative completes only if two communications successfully complete; there is no single communication to serve as the commit point in CML. However, in TE Haskell, this event may synchronize either with a thread synchronizing on `ev2` or with a thread synchronizing on the following event:

```
ev4 = sendEvt ch 2
```

Note, however, that if there are threads synchronizing on both `ev2` and `ev4`, then the event with which `ev3` synchronizes is non-deterministic; hence, `chooseEvt` is a commutative combinator.

The event `neverEvt` is an event which never successfully completes when synchronized upon. Since `neverEvt` never successfully completes, it may never be chosen by `chooseEvt`; hence, `neverEvt` is a left and right unit for `chooseEvt`. Likewise, `neverEvt` is a left and right zero for `thenEvt`. Hence, `Evt a` forms a monad-with-plus [25, 9, 13], with `chooseEvt` as monadic plus and `neverEvt` as monadic zero.

The semantics of transactional events requires that a synchronization not commit to a particular alternative in a `chooseEvt` unless all of the constituent events in that alternative can successfully complete. A synchronization also does not commit to a communication partner in a `sendEvt` or `recvEvt` unless the communication leads to both partners successfully completing their transactional event synchronizations. Hence, we may view a synchronization as a transaction that commits when a collection of choice alternatives and communication partners may successfully complete, and aborts when such a collection may not successfully complete.

Because synchronous message passing involves other transactional event synchronizations, which may abort, a synchronization can only successfully complete if all of the synchronizations with which it has communicated can also successfully complete. This allows transactional events to achieve n-way synchronization, which is impossible as an event abstraction in CML or as an STM computation in STM Haskell.

3.1 Exceptions in `Evt a`

Thus far, the features of transactional events have straightforwardly followed the intuition of extending CML event synchronization to be an all-or-nothing transaction. Because Haskell allows exceptions to be thrown from pure code, it is necessary to specify the semantics of exceptions in transactional events and in synchronizations. The treatment of exceptions in TE Haskell is somewhat subtle.

Asynchronous exceptions [14] have a straightforward treatment. If a thread that is synchronizing on a transactional event receives an asynchronous exception, it makes sense for the synchronization to be entirely aborted and the exception raised as if it had been thrown just before the start of the synchronization. Because event synchronization is intended to be an all-or-nothing transaction, an asynchronous exception should always be seen as arriving before or after the synchronization step, never during. Asynchronous exceptions cannot be caught within a transactional event.

For synchronous exceptions that are thrown while synchronizing on an event, there are several possible design choices. Uncaught exceptions could cause an event synchronization to abort without committing, continuing to propagate the exception from the `sync ev` expression. However, because transactional events include synchronous communication between threads, this semantics would break the intuition that uncommitted events have no observable effects. Consider the following threads and synchronizations:

```
t1 = sync (do { i <- recvEvt c
              ; if i == 0 then throw Foo
                          else return i })

t2 = sync (do { sendEvt c 0 ; neverEvt })
```

If the synchronization in `t1` aborts with the exception `Foo`, then this behavior would be caused by an uncommitted (and, in fact, uncommittable) synchronization in `t2`. This not only breaks the basic intuition of transactions, but can lead to some truly strange behavior. For example, consider the following threads:

```
t3 = sync ((do { i <- recvEvt c
               ; if i == 0 then throw Foo
                           else return i })
          `chooseEvt` (sendEvt c 0))

t4 = sync ((do { i <- recvEvt c
               ; if i == 0 then throw Foo
                           else return i })
          `chooseEvt` (sendEvt c 0))
```

Under the immediate abort semantics, the tentative communications in these synchronizations could cause each other to abort. In this case, each aborted synchronization actually has an effect: the effect of causing the other to abort.

One possible solution to these problems would be to require uncaught exceptions to commit a synchronization (including the synchronizations of all communication partners) and consider the propagated exception to be the value produced by the event synchronization. In the first example above, `t1` would not be able to successfully commit since its communication partner cannot commit. In the second example, only one of the synchronizations could commit to an exception, since the other would have to commit to the send that causes the exception. However, this semantics has the disadvantage of complicating common-knowledge reasoning about cooperating transactional event synchronizations. Consider the case of these three synchronizing threads:

```
t5 = sync (sendEvt ch1 0)

t6 = sync (do { x <- recvEvt ch1
              ; if f x then sendEvt ch2 'T'
                       else neverEvt })

t7 = sync (recvEvt ch2)
```

At first glance, it would seem that if `t5` completes it synchronization, then it should know that `t7` has completed its synchronization, because for `t5` to commit to its send on `ch1`, `t6` must commit to its receive on `ch1` and its send on `ch2`; hence, `t7` must commit to its receive on `ch2`. However, if `f 0` throws an exception, then `t6` need only commit to its receive on `ch1`, and `t5` and `t6` may complete their synchronizations without `t7` completing its synchronization. Allowing exceptions that propagate to the top-level of an event synchronization to be treated as an event that successfully completes means that programmers must carefully consider every place that an uncaught exception might be thrown as a possible commitment point for the event synchronization. We believe that this goes against the spirit of exceptions in Haskell, since exceptions are by nature rare and programmers are unlikely to account for all possible origins of exceptions. With these considerations in mind, we have chosen to make uncaught exceptions that reach the top-level of an event synchronization act as an event which never successfully completes, much like `neverEvt`. Hence, under our semantics, the threads `t3` and `t4` block indefinitely, as do the thread `t5`, `t6`, and `t7` when `f 0` throws an exception. The choice of whether or not an uncaught exception may commit its synchronization is a free design decision; our choice was guided by the example above, where we felt it better for the program to do nothing (block) than to behave in a non-intuitive fashion (commit `t5` and `t6` without `t7`).

Nonetheless, there are times when an event must be robust against exceptions. Therefore, despite the potential to complicate reasoning, we do provide the means to throw and catch exceptions in transactional events. The event `throwEvt ex` throws the exception `ex` when it is synchronized upon, while the event `catchEvt ev h` is the event that acts as `ev`, except, if synchronizing on `ev` throws an exception `ex`, then it synchronizes on `h ex`. Exceptions thrown from pure code may also be caught by `catchEvt`. Programmers making use of `catchEvt` have the responsibility to consider all possible origins of the exceptions handled, making sure that the non-local control flow they are introducing does not destroy any important mutual commitment properties.

4. Semantics

In this section, we provide a formal, operational semantics for transactional events. TE Haskell draws inspiration from both Concurrent ML and Concurrent Haskell. Like Concurrent ML, it includes first-class synchronous events and event combinators, but extended with sequential composition. Like Concurrent Haskell, it includes first-class I/O actions and I/O combinators.

4.1 Syntax

Figure 3 gives the syntax of TE Haskell. Values and expressions in the language naturally divide into four categories: constants (characters c, thread identifiers θ, and channel names κ), event combinators, I/O combinators, and standard functional language terms (e.g., λ-abstractions, applications), which we omit.

The event combinators should be familar from the description in the previous section. Likewise, most of the I/O combinators should be familiar from the description of Concurrent Haskell. In order to clarify the behavior of monadic sequencing and exception handling in the `Evt` and `IO` monads, we equip the `IO` monad with distinguished combinators: `unitIO`, `bindIO`, `throwIO`, and `catchIO`.

Values

v	::=	c	characters
	|	θ	thread identifiers
	|	κ	channel names
	|	`alwaysEvt` e | `thenEvt` e_1 e_2	
	|	`neverEvt` | `chooseEvt` e_1 e_2	
	|	`throwEvt` e | `catchEvt` e_1 e_2	
	|	`newSChan` | `recvEvt` κ | `sendEvt` κ_1 e_2	
	|	`unitIO` e | `bindIO` e_1 e_2	
	|	`throwIO` e | `catchIO` e_1 e_2	
	|	`getChar` | `putChar` c	
	|	`forkIO` e | `sync` e	
	|	`\`x`->`e | ...	functional language values

Expressions

e	::=	x	variables
	|	v	values
	|	e_1 e_2 | ...	functional language expressions

Figure 3. TE Haskell: Syntax

4.2 Dynamic Semantics

The essence of the dynamics is to interpret sequential terms, `Evt` terms, and `IO` terms as separate sorts of computations. This is expressed by three levels of evaluation: pure evaluation of sequential terms, synchronous evaluation of transactional events, and concurrent evaluation of concurrent threads. The bridge between the `Evt` and `IO` computations is synchronization, which moves threads from concurrent evaluation to synchronous evaluation and back to concurrent evaluation.

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

The "lowest" level of evaluation is the sequential evaluation of pure functional language terms. Unsurprisingly, our sequential evaluation relation is entirely standard and thus omitted.

4.2.2 Synchronous Evaluation ($\mathcal{S} \rightsquigarrow \mathcal{S}'$)

The "middle" level of evaluation is synchronous evaluation of transactional events. We organize a group of synchronizing events as a set of pairs of thread identifiers and `Evt` expressions:

Synchronizing Event	S	::=	$\langle \theta, e \rangle$
Synchronization Groups	$\mathcal{S}$	::=	$\{S, \ldots\}$

The synchronous evaluation relation is closely related to the event matching relation in the semantics of Concurrent ML [21]. Intuitively, the relation:

$$\{\langle \theta_1, e_1 \rangle, \ldots, \langle \theta_k, e_k \rangle\} \rightsquigarrow \{\langle \theta_1, e'_1 \rangle, \ldots, \langle \theta_k, e'_k \rangle\}$$

means that the events $e_1,\ldots,e_k$ make one step towards synchronization by transforming into the events $e'_1,\ldots,e'_k$. The complete set of rules is given in Figure 2a.

The rule EVTEVAL implements the sequential evaluation of an expression in the active position. The rule EVTSUBSET is a structural rule that admits transitioning on a subset of the synchronizing events. The rule EVTTHENALWAYS implements sequential composition in the `Evt` monad. The rules EVTCHOOSE1 and EVTCHOOSE2 implement a non-deterministic choice between events. The rules EVTTHENTHROW, EVTCATCHALWAYS, and EVTCATCHTHROW propagate exceptions in the standard way.

The rule EVTNEWSCHAN allocates a new channel name; note that the freshness of κ' is with respect to the entire program state. The rule EVTCOMM implements the two-way rendezvous of communication along a channel; note that the transition replaces the `sendEvt` and `recvEvt` events with `alwaysEvt` events.

It is worth considering the possible terminal configurations for a set of events. A "good" terminal configuration is one in which all events are reduced to always events: $\{\langle \theta_1, \texttt{alwaysEvt}\ e_1 \rangle, \ldots, \langle \theta_k, \texttt{alwaysEvt}\ e_k \rangle\}$. The "bad" terminal configurations are ones with never events, or with uncaught exceptions, or with unmatched send/receive events.

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

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

Concurrent Threads	T	::=	$\langle \theta, e \rangle$
Thread Soups	$\mathcal{T}$	::=	$\{T, \ldots\}$

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

Actions	a	::=	$?c \mid !c \mid \epsilon$

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. In a real language, there would be many other observable I/O actions.

The complete set of rules is given in Figure 2b. 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 IOFORK creates a new thread by selecting a fresh thread identifier, which is returned to the parent thread, and adding a new term to the thread soup. The rule IOBINDUNIT implements sequential composition in the `IO` monad, while the rules IOBINDTHROW, IOCATCHUNIT, and IOCATCHTHROW propagate exceptions in the standard way. The two rules IOGETCHAR and IOPUTCHAR perform the appropriate labeled transition, yielding an observable action.

The most interesting rule is IOSYNC. The rule selects some collection of threads that are prepared to synchronize on transactional events. 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 `alwaysEvt` events. That is, the set of events successfully synchronizes to final results. The results of synchronization are moved from the `Evt` computation to the `IO` computation.

There are two interesting facets of the IOSYNC rule. The first is that the concurrent transition has a silent action. Hence, synchro-

Synchronous Evaluation Contexts $M^{Evt} ::= [] \mid \texttt{thenEvt}\ M_1^{Evt}\ e_2 \mid \texttt{catchEvt}\ M_1^{Evt}\ e_2$

EVTEVAL
$$\frac{e \hookrightarrow e'}{\{\langle \theta, M^{Evt}[e] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[e'] \rangle\}}$$

EVTSUBSET
$$\frac{0 < |\mathcal{S}''| \qquad \mathcal{S} \rightsquigarrow \mathcal{S}'}{\mathcal{S} \uplus \mathcal{S}'' \rightsquigarrow \mathcal{S}' \uplus \mathcal{S}''}$$

EVTTHENALWAYS
$\{\langle \theta, M^{Evt}[\texttt{thenEvt}\ (\texttt{alwaysEvt}\ e_1)\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[e_2\ e_1] \rangle\}$

EVTTHENTHROW
$\{\langle \theta, M^{Evt}[\texttt{thenEvt}\ (\texttt{throwEvt}\ e_1)\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[\texttt{throwEvt}\ e_1] \rangle\}$

EVTCHOOSE1
$\{\langle \theta, M^{Evt}[\texttt{chooseEvt}\ e_1\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[e_1] \rangle\}$

EVTCHOOSE2
$\{\langle \theta, M^{Evt}[\texttt{chooseEvt}\ e_1\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[e_2] \rangle\}$

EVTCATCHALWAYS
$\{\langle \theta, M^{Evt}[\texttt{catchEvt}\ (\texttt{alwaysEvt}\ e_1)\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[\texttt{alwaysEvt}\ e_1] \rangle\}$

EVTCATCHTHROW
$\{\langle \theta, M^{Evt}[\texttt{catchEvt}\ (\texttt{throwEvt}\ e_1)\ e_2] \rangle\} \rightsquigarrow \{\langle \theta, M^{Evt}[e_2\ e_1] \rangle\}$

EVTNEWSCHAN
$$\frac{\kappa' \text{ fresh}}{\{\langle \theta, M^{Evt}[\texttt{newSChan}] \rangle\} \rightsquigarrow \{M^{Evt}[\texttt{alwaysEvt}\ \kappa'] \rangle\}}$$

EVTCOMM
$\{\langle \theta_1, M_1^{Evt}[\texttt{sendEvt}\ \kappa\ e] \rangle, \langle \theta_2, M_2^{Evt}[\texttt{recvEvt}\ \kappa] \rangle\}$
$\rightsquigarrow \{\langle \theta_1, M_1^{Evt}[\texttt{alwaysEvt}\ ()] \rangle, \langle \theta_2, M_2^{Evt}[\texttt{alwaysEvt}\ e] \rangle\}$

(a) Synchronous Evaluation

Concurrent Evaluation Contexts $M^{IO} ::= [] \mid \texttt{bindIO}\ M_1^{IO}\ e_2 \mid \texttt{catchIO}\ M_1^{IO}\ e_2$

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

IOFORK
$$\frac{\theta' \text{ fresh}}{\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{forkIO}\ e] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ \theta'] \rangle, \langle \theta', e \rangle\}}$$

IOBINDUNIT
$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{bindIO}\ (\texttt{unitIO}\ e_1)\ e_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[e_2\ e_1] \rangle\}$

IOBINDTHROW
$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{bindIO}\ (\texttt{throwIO}\ e_1)\ e_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{throwIO}\ e_1] \rangle\}$

IOCATCHUNIT
$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{catchIO}\ (\texttt{unitIO}\ e_1)\ e_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ e_1] \rangle\}$

IOCATCHTHROW
$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{catchIO}\ (\texttt{throwIO}\ e_1)\ e_2] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[e_2\ e_1] \rangle\}$

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

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

IOSYNC
$$\frac{\{\langle \theta_1, e_1 \rangle, \ldots, \langle \theta_k, e_k \rangle\} \rightsquigarrow^* \{\langle \theta_1, \texttt{alwaysEvt}\ e_1' \rangle, \ldots, \langle \theta_k, \texttt{alwaysEvt}\ e_k' \rangle\}}{\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{sync}\ e_1] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\texttt{sync}\ e_k] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e_1'] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\texttt{unitIO}\ e_k'] \rangle\}}$$

(b) Concurrent Evaluation

Figure 2. TE Haskell: Dynamic Semantics

nization 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 IOSYNC 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 IOSYNC rule conveys an *all-or-nothing* property on transactional event synchronization.

4.3 Discussion

As noted before, we may 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.

We may also 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 must backtrack at synchronization failures (e.g, never events, uncaught exceptions, and unmatched send/receive events).

Both of these interpretations clarify the nature of the *all-or-nothing* property of the `Evt` monad-with-plus. Note that the silence of synchronous evaluation steps means 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.

We may also see that TE Haskell preserves the "spirit" of Concurrent ML. Recall from Section 2.1 that one often wants to implement a protocol consisting of a sequence of communications: $c_1\,;c_2\,;\cdots;\,c_n$. The `thenEvt` combinator of TE Haskell obviates the need to distinguish one communication c_i 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 $c_{i+1}\,;\cdots;\,c_n$). (Nonetheless, one may still implement a sequence of communications with a dedicated commit point; see Section 5.2.)

Instead, we may implement the protocol as a sequence of communications using the `thenEvt` combinator to ensure that all of the communications synchronize or none of them synchronize. When this protocol participates in selective communication, it will be chosen only if all of the communications are able to synchronize with corresponding communications in other synchronizing threads.

5. Expressiveness

In this section, we explore the expressiveness of TE Haskell by demonstrating how a number of powerful concurrency abstractions may be encoded. We begin by discussing an implementation of the guarded receive abstraction (Section 5.1). It should come at no surprise that we may easily encode Concurrent ML (Section 5.2). Interestingly, we may also easily encode transactional shared memory (Section 5.3). Finally, we demonstrate that TE Haskell is strictly more powerful than Concurrent ML by encoding an abstract three-way rendezvous operation (Section 5.4).

Throughout this section, we will make extensive use of Haskell's `do`-notation for monadic computations. To improve the readability of the encodings, we recall the `fmap` and `join` operations, which may be defined for any monad:

```
fmap :: Monad m => (a -> b) -> m a -> m b
fmap f m = do { x <- m ; return (f x) }

join :: Monad m => m (m a) -> m a
join mm = do { m <- mm; m }
```

5.1 Guarded Receive

The transactional nature of events in TE Haskell admits the implementation of useful synchronous operations and abstractions that cannot be constructed in CML. One example of a synchronization operation that is enabled by transactional events is guarded receive: the receipt of a message on a channel only if the message satisfies a boolean guard. Several message-passing languages and formalisms, including Erlang [1] and CSP [10], support some form of guarded receive.

In TE Haskell we can add guarded receive to channel communications by modifying only the receiver's code. This cannot be done in CML and cannot be done for synchronous communication in STM Haskell. This section shows that the use of transactional events can result in simpler and more modular implementations of synchronous abstractions, like guarded receive, than can be achieved with CML's primitives alone.

In TE Haskell it is a simple matter to write a function that creates an event to perform a guarded receive:

```
grecvEvt :: (a -> Bool) -> SChan a -> Evt a
grecvEvt g ch = do { x <- recvEvt ch
                   ; if g x then return x
                            else neverEvt }
```

This new synchronous operation can now be freely composed, either sequentially (with `thenEvt`) or alternatively (with `chooseEvt`), with other synchronous operations. For example, the event that chooses between receiving a tuple whose first element is 0 and one whose first element is 1 can be written:

```
(grecvEvt (\(x,y) -> x = 0) ch)
`chooseEvt`
(grecvEvt (\(x,y) -> x = 1) ch)
```

This synchronous abstraction cannot be implemented in CML because as soon as a receive is performed, the sending thread completes its synchronization and becomes unblocked. There is no way for the receive to be undone and the sending thread reblocked. It is possible to implement this behavior in CML using a protocol in which the sender and the receiver cooperatively interact, but this is much less modular than the use of `grecvEvt`.

This implementation of guarded receive is also kill-safe [2]: if the reading thread is killed and the event synchronization is aborted, then the program state is kept consistent. In addition, threads performing a guarded receive can condition the receipt on arbitrary, possibly non-terminating, predicates without interfering with other threads that may wish to read on the channel. Achieving these properties with CML events, while possible, requires much more complexity in addition to the lack of modularity.

```
alwaysCMLEvt :: a -> CMLEvt a
alwaysCMLEvt x = lift (alwaysEvt x)

wrapCMLEvt :: CMLEvt a -> (a -> IO b) -> CMLEvt b
wrapCMLEvt iei f = fmap (fmap (>>= f)) iei

guardCMLEvt :: IO (CMLEvt a) -> CMLEvt a
guardCMLEvt iiei = join iiei

neverCMLEvt :: CMLEvt a
neverCMLEvt = lift (neverEvt)

chooseCMLEvt :: CMLEvt a -> CMLEvt a -> CMLEvt a
chooseCMLEvt iei1 iei2 =
  do { ei1 <- iei1 ; ei2 <- iei2
     ; return (ei1 `chooseEvt` ei2) }

recvCMLEvt :: SChan a -> CMLEvt a
recvCMLEvt ch = lift (recvEvt ch)

sendCMLEvt :: SChan a -> a -> CMLEvt ()
sendCMLEvt ch x = lift (sendEvt ch x)

syncCML :: CMLEvt a -> IO a
syncCML iei = do { ei <- iei ; i <- sync ei ; i }
```

Figure 4. CML Encoding

5.2 Encoding Concurrent ML

We consider a simple CML encoding, making two relatively minor changes to the semantics. First, we only consider a binary `choose` combinator. Second, we omit the `withNack` combinator. (Since `withNack` may be implemented as a stylized use of the other CML combinators, there is no loss of expressive power.)

Recall that functions in Standard 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 a language with imperative I/O to a language with monadic I/O is beyond the scope of this paper (but is a well-understood problem [15]), we note that the general idea is to translate a function of the type τ_1 `->` τ_2 to a function of the type τ_1 `-> IO` τ_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 CML event as a pre-synchronization `IO` action that yields an `Evt` value that in turn yields a post-synchronous `IO` action:

```
type CMLEvt a = IO (Evt (IO a))
```

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

```
lift :: Evt a -> CMLEvt a
lift e = return (fmap return e)
```

The encodings of the CML combinators are given in Figure 4. 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.

5.3 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 [21, Sections 3.2.3 and 3.2.7]. Since transactional events extend CML synchroniza-

tions with an all-or-nothing transactional property, an interesting question is whether or not we may encode shared memory transactions in TE Haskell. This section demonstrates such an encoding.

We take as our starting point the software transactional memory (STM) extension of Concurrent Haskell [7]. STM Haskell provides a monadic type (`STM a`) that denotes an atomic memory transaction and a type (`TVar a`) that denotes a transactional variable, along with the following interface:

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

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
```

The `STM a` type represents computations that accesses transactional variables. An `STM a` computation may be passed to `atomic`, which returns an `IO a` action that, when performed, runs the transaction atomically with respect to all other memory transactions. The `retrySTM` operation aborts a transaction and `orElse` operation selects (with left bias) between transactions. Hence, `STM a` forms a (non-commutative) monad-with-plus.

There are obvious connections between STM Haskell and TE Haskell. Both use an "outer" `IO` 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 visible until the interaction executes with a consistent view.

Our encoding of the `STM` monad-with-plus makes three 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 `a -> IO (TVar a)`, which is a consequence of the fact that forking a thread must occur in the `IO` monad. Finally, the semantics of uncaught exceptions are slightly different. Nonetheless, we feel that this encoding is well within the spirit of transactional memory and demonstrates the expressibility of TE Haskell.

Figure 5 gives the encoding, which we discuss 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 sends its thread identifier to the server thread. If a server thread receives the thread identifier of a second thread while the first thread's transaction is incomplete, it aborts the transaction (by synchronizing on `neverEvt`). Hence, a thread completes its atomic transaction if and only if it is the only thread to communicate with those transactional variables accessed during the transaction.

From the description above, it is clear that we may take the encoding of the `STM a` type to be the `Evt a` type. From this definition, the encoding of the simple monadic operations follows directly. However, we must also provide the thread identifier at each read or write of a transactional variable. Hence, we extend the `Evt` interface with `myThreadIdEvt :: Evt ThreadId`, which is an event that immediately yields the thread identifier of the synchronizing thread. The semantics of Section 4 may be easily extended with `myThreadIdEvt`, as the thread identifiers of synchronizing threads are available in the synchronous evaluation relation.

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

```
type STM a = Evt a

atomic = sync
unitSTM = alwaysEvt
bindSTM = thenEvt
retrySTM = neverEvt
orElseSTM = chooseEvt

TVar a = (SChan ThreadId, SChan a, SChan a)

readTVar (tch, rch, wch) =
  do { tid <- myThreadIdEvt
     ; sendEvt tch tid
     ; readEvt rch }

writeTVar (tch, rch, wch) x =
  do { tid <- myThreadIdEvt
     ; sendEvt tch tid
     ; sendEvt wch x }

newTVar :: a -> IO (TVar a)
newTVar x =
  do { tch <- sync newSChan
     ; rch <- sync newSChan
     ; wch <- sync newSChan
     ; let serve x =
             do { tid' <- recvEvt tch
                ; x' <- (do { sendEvt wch x
                            ; alwaysEvt x })
                        `chooseEvt`
                        (recvEvt rch)
                ; return (tid', x') }
     ; let loopEvt tid x =
             (do { (tid', x') <- serve x
                 ; if tid /= tid'
                      then neverEvt
                      else loopEvt tid' x' })
             `chooseEvt`
             (alwaysEvt x)
     ; let loopIO x =
             do { x'' <- sync (do
                                { (tid', x') <- serve x
                                ; loopEvt tid' x' })
                ; loopIO x'' }
     ; forkIO (loopIO x)
     ; return (tch, rch, wch) }
```

Figure 5. Transactional Shared Memory Encoding

All of the interesting action happens in the thread that services a transactional variable, which is spawned when a transactional variable is created. 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 an `Evt` action that carries the state of the variable through a single atomic transaction. The `serve` function is an `Evt` computation that 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` ser-

vices 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 the synchronization of 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. Furthermore, the encoding has a progress guarantee: if there are two STM computations which could each commit in isolation given the current state of shared `TVar`s, then at least one will successfully commit if they are run concurrently. This property follows from the fact that the semantics of TE Haskell requires a successful synchronization to be found if one exists – and such a synchronization does exist, namely, the one where each server thread communicates with same STM computation.

5.4 Encoding Three-way Rendezvous

The previous sections have demonstrated that TE Haskell is as expressive as CML and transactional shared memory. A second question is whether TE Haskell is *more* expressive than than CML.

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

THEOREM 1. **(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). [21, Section 6.4]*

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.

TE Haskell is strictly more expressive than CML:

THEOREM 2. **(TE Haskell Expressivity)**

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

We demonstrate this theorem (for the case $n = 2$, though the theorem holds in general) by providing an implementation of three-way rendezvous, using the two-way rendezvous primitives `sendEvt` and `recvEvt` (see Figure 6).

Our three-way rendezvous example is the *triple-swap channel*. This type of channel allows three threads to swap values when they synchronize; each thread offers a value and each thread accepts the two values offered by the other two threads. Note that we require each thread to be matched with precisely two other threads; if more than three threads attempt to swap on the same triple-swap channel at roughly the same time, it should not be the case that values are swapped amongst more than three threads.

A value of type `TriSChan a` is implemented as a channel carrying pairs of a value (of type `a`) and a reply channel (of type `SChan (a,a)`). Hence, the `newTriSChan` action simply creates a new channel.

More interesting is the implementation of the `swapEvt` action. A thread that swaps on a channel non-deterministically chooses between acting as a client or as a leader in the exchange protocol. A client thread creates a new reply channel, sends its value and reply channel along the triple-swap channel, and then receives the two other values along the reply channel. A leader thread receives the values and reply channels from two client threads, sends the appropriate pairs of values along the reply channels, and returns the appropriate pair of values as the result of the synchronization.

```
type TriSChan a = SChan (a, SChan (a, a))

newTriSChan :: Evt (TriSChan a)
newTriSChan = newSChan

swapEvt :: TriSChan a -> a -> Evt (a, a)
swapEvt ch x1 = client `chooseEvt` leader
  where client = do { replyCh <- newSChan
                    ; sendEvt ch (x1, replyCh)
                    ; recvEvt replyCh }
        leader = do { (x2, replyCh2) <- recvEvt ch
                    ; (x3, replyCh3) <- recvEvt ch
                    ; sendEvt replyCh2 (x3, x1)
                    ; sendEvt replyCh3 (x1, x2)
                    ; alwaysEvt (x2, x3) }
```

Figure 6. The `TriSChan` Abstraction

It is worth noting the reason that the above implementation does not suffice for CML. The fundamental difficulty is that (from the client's point of view) the protocol requires two communications to accomplish the exchange. However, in CML, one of these communications must be chosen 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 leader (successfully synchronizing on the commit point), but then block waiting for another thread to complete the 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 communication (thereby enabling a leader thread and another client thread to swap) but then fail to rendezvous at the second communication, by taking another alternative in the `choose`. This breaks the abstraction, because the other two threads cannot know that the thread received their swap values.

This implementation may be easily extended to arbitrary n-way synchronization, for any fixed or dynamic n. For the dynamic case, we have the following interface:

```
type NWaySChan a
newNWaySChan :: Int -> Evt (NWaySChan a)
swapEvt :: NWaySChan a -> a -> [a]
```

where `newNWaySChan n` yield a synchronous channel for swapping among n threads. With this interface, we may easily encode first-class synchronization barriers [24].

6. Implementation

In this section, we describe a "proof-of-concept" implementation of transactional events.[1] While we make no claims that this implementation is optimal, we believe that it has a number of attractive properties. First, the entire implementation is written in Haskell, using the software transactional memory (STM) extensions of Concurrent Haskell [7] available in the Glasgow Haskell Compiler (GHC) [4].[2] Hence, the implementation required no changes to the compiler or run-time system. Furthermore, the implementation may take advantage of future developments that will extend GHC to execute Haskell (with STM) on shared-memory multiprocessors [6]. Second, we may see from the implementation (and from knowledge of the underlying implementation of STM) that transactional events do not require a global lock to coordinate the synchronization among communicating threads. Hence, the synchronization of

[1] The implementation may be obtained at `http://www.cs.cornell.edu/People/fluet/research/tx-events`.

[2] However, the implementation of transactional events using STM is significantly more involved than the encoding of STM using transactional events given in Section 5.3.

SYNCINIT

$$\mathcal{P} \uplus \{\langle \theta, M^{IO}[\texttt{sync}\ e]\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{IO}, e, \bullet\rangle\}$$

SYNCCOMMIT

$$\frac{\mathsf{Committable}(\{\langle \theta_1, M_1^{IO}, \texttt{alwaysEvt}\ e_1, \rho_1\rangle, \ldots, \langle \theta_n, M_n^{IO}, \texttt{alwaysEvt}\ e_n, \rho_n\rangle\})}{\mathcal{P} \uplus \{\langle \theta_1, M_1^{IO}, \texttt{alwaysEvt}\ e_1, \rho_1\rangle, \ldots, \langle \theta_n, M_n^{IO}, \texttt{alwaysEvt}\ e_n, \rho_n\rangle\} \xrightarrow{\epsilon} \mathcal{P} \setminus_{\{\theta_1,\ldots,\theta_n\}} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e_1]\rangle, \ldots, \langle \theta_n, M_n^{IO}[\texttt{unitIO}\ e_n]\rangle\}}$$

EVTCHOOSE

$$\mathcal{P} \uplus \{\langle \theta, M^{IO}, M^{Evt}[\texttt{chooseEvt}\ e_1\ e_2], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{IO}, M^{Evt}[e_1], \texttt{Left:}\rho\rangle, \langle \theta, M^{IO}, M^{Evt}[e_2], \texttt{Right:}\rho\rangle\}$$

EVTCOMM

$$\frac{\mathsf{Coherent}(\langle \theta_1, \rho_1\rangle, \langle \theta_2, \rho_2\rangle)}{\mathcal{P} \uplus \{\langle \theta_1, M_1^{IO}, M_1^{Evt}[\texttt{sendEvt}\ \kappa\ e], \rho_1\rangle, \langle \theta_2, M_2^{IO}, M_2^{Evt}[\texttt{recvEvt}\ \kappa], \rho_2\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta_1, M_1^{IO}, M_1^{Evt}[\texttt{sendEvt}\ \kappa\ e], \rho_1\rangle, \langle \theta_2, M_2^{IO}, M_2^{Evt}[\texttt{recvEvt}\ \kappa], \rho_2\rangle, \langle \theta_1, M_1^{IO}, M_1^{Evt}[\texttt{alwaysEvt}\ ()], \mathsf{Send}(\langle \theta_2, \rho_2\rangle){:}\rho_1\rangle, \langle \theta_2, M_2^{IO}, M_2^{Evt}[\texttt{alwaysEvt}\ e], \mathsf{Recv}(\langle \theta_1, \rho_1\rangle){:}\rho_2\rangle\}}$$

Figure 7. TE Haskell: Dynamic Semantics: Refined

threads will not impact the progress of non-synchronizing threads, nor will the synchronization of one group of threads impact the progress towards the synchronization of another independent group of threads; that is, the implementation has a property similar to disjoint access parallelism.

Since there is a large gap between the semantics of Section 4 and a viable implementation of TE Haskell, we proceed in two stages. First, we close the gap by giving a refined the semantics that eliminates the most glaring impediment to implementation. Then, we briefly discuss how the remaining impediments in the refined semantics are eliminated in our implementation.

6.1 Refined Semantics

It should come as no surprise that the major stumbling block in implementing TE Haskell is how to effectively implement the IOSYNC rule of Figure 2b. A naïve interpretation of this rule requires an implementation to omnisciently choose, up front, a set of synchronizing threads and a sequence of synchronous evaluation transitions, which may be arbitrarily long. A semantics that is more readily seen to have a viable implementation is one where the choice of threads and evaluation transitions in a synchronization may be delayed.

In our refined semantics, we distinguish between the concurrent threads of Section 4.2.3 and search threads:

Concurrent Threads	T	::=	$\langle \theta, e\rangle$
Search Threads	S	::=	$\langle \theta, M^{IO}, e, \rho\rangle$
Thread Soups	$\mathcal{P}$	::=	$\{T, \ldots, S, \ldots\}$

Concurrent threads continue to execute according to the rules in Figure 2b, except that we will shortly revise the IOSYNC rule. A search thread includes the thread identifier and `IO` continuation of the concurrent thread on whose behalf it is searching for a synchronization, a transactional event to be evaluated, and a transactional event path recording the history of the search thread. A completed search thread is one where the transactional event has the form `alwaysEvt e'`.

A transactional event path records the non-deterministic choices (including communication) made during the evaluation of a transactional event, while a trail pairs a thread identifier with a path:

Path	ρ	::=	$\mathsf{Left}{:}\rho \mid \mathsf{Right}{:}\rho \mid \mathsf{Send}(\tau){:}\rho \mid \mathsf{Recv}(\tau){:}\rho \mid \bullet$
Trail	τ	::=	$\langle \theta, \rho\rangle$

We say that a path ρ_a *extends* the path ρ_b, written $\rho_a \succeq \rho_b$ if ρ_b is a suffix of ρ_a.

The dependencies of a trail $\langle \theta, \rho\rangle$, written $\mathsf{Dep}(\langle \theta, \rho\rangle)$, is the set of trails that it interacts with, either directly or indirectly:

$$\begin{array}{rcl} \mathsf{Dep}(\langle \theta, \bullet\rangle) & = & \emptyset \\ \mathsf{Dep}(\langle \theta, \mathsf{Left}{:}\rho\rangle) & = & \mathsf{Dep}(\langle \theta, \rho\rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Right}{:}\rho\rangle) & = & \mathsf{Dep}(\langle \theta, \rho\rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Send}(\langle \theta', \rho'\rangle){:}\rho\rangle) & = & \{\langle \theta', \mathsf{Recv}(\langle \theta, \rho\rangle){:}\rho'\rangle\} \\ & & \cup\, \mathsf{Dep}(\langle \theta, \rho\rangle) \cup \mathsf{Dep}(\langle \theta', \rho'\rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Recv}(\langle \theta', \rho'\rangle){:}\rho\rangle) & = & \{\langle \theta', \mathsf{Send}(\langle \theta, \rho\rangle){:}\rho'\rangle\} \\ & & \cup\, \mathsf{Dep}(\langle \theta, \rho\rangle) \cup \mathsf{Dep}(\langle \theta', \rho'\rangle) \end{array}$$

Note that the dependencies at a communication adds a trail in which the partner's path includes the matching communication.

In order for a completed search thread to commit, all of its dependencies must be willing to commit. We formalize this intuition in the definition of a committable set of completed search threads:

DEFINITION 1. **(Committable)**
A set of completed search threads $\{\langle \theta_i, M_i^{IO}, \texttt{alwaysEvt}\ e_i, \rho_i\rangle, \ldots\}$ *is* committable *if*

- *each* θ_i *is unique,*
- *for each* $\langle \theta_i, \rho_i\rangle$*, if* $\langle \theta, \rho\rangle \in \mathsf{Dep}(\langle \theta_i, \rho_i\rangle)$*, then there exists* j *such that* $\theta = \theta_j$ *and* $\rho_j \succeq \rho$*.*

The definition of a committable set of search threads ensures that each search thread corresponds to a unique synchronization and that all dependencies of each search thread are included in the set.

Figure 7 revises the dynamic semantics of Section 4 to evaluate search threads. The single IOSYNC rule is replaced by the SYNCINIT and SYNCCOMMIT rules, while the synchronous evaluation rules dealing with event expressions are replaced by rules dealing with search threads. The SYNCINIT rule transitions a concurrent thread at a `sync` to a search thread with an empty path. The SYNCCOMMIT rule transitions a set of committable search threads to concurrent threads, while also removing all other search threads that were searching on behalf of the now synchronized concurrent threads. (The notation $\mathcal{P}\backslash_\Theta$ removes all search threads with a thread identifier in Θ from the thread soup $\mathcal{P}$.)

We elide the straightforward adaptations of the rules for sequential evaluation of pure terms in a search thread, propagation of `alwaysEvt` and `throwEvt`, and channel allocation. The EVTCHOOSE rule transitions a single search thread evaluating a `chooseEvt` to two search threads evaluating the choice alternatives; note that the paths of the search threads are extended to record the non-deterministic choice.

The EVTCOMM rule implements synchronous message passing along a channel. Note that the search threads corresponding to the sender and receiver remain in the thread soup to participate in other communications, as there is no guarantee that this tentative communication will lead to synchronization. While it would be acceptable to spawn new search threads for any sender/receiver pair on the same channel, there are some communications for which the resulting search threads may never commit together. Hence, we only allow communication between coherent search threads:

DEFINITION 2. **(Coherent)**
The trails $\langle \theta_1, \rho_1\rangle$ *and* $\langle \theta_2, \rho_2\rangle$ *are* coherent *if*

- $\theta_1 \neq \theta_2$*,*
- *if* $\langle \theta_1, \rho\rangle \in \mathsf{Dep}(\langle \theta_2, \rho_2\rangle)$*, then* $\rho_1 \succeq \rho$*,*

- *if* $\langle \theta_2, \rho \rangle \in \mathsf{Dep}(\langle \theta_1, \rho_1 \rangle)$*, then* $\rho_2 \succeq \rho$*, and*
- *if* $\langle \theta, \rho_a \rangle \in \mathsf{Dep}(\langle \theta_1, \rho_1 \rangle)$ *and* $\langle \theta, \rho_b \rangle \in \mathsf{Dep}(\langle \theta_2, \rho_2 \rangle)$*, then* $\rho_a \succeq \rho_b$ *or* $\rho_b \succeq \rho_a$.

Threads may not communicate with themselves; hence, we require that $\theta_1 \neq \theta_2$. If θ_2 interacted (directly or indirectly) with some θ_1 search thread in the past, then it must have been in the history of *this* θ_1 search thread; and vice versa. Finally, if there is a common depended upon thread, then the path in one dependency must be an extension of the path in the other dependency; that is, the search threads θ_1 and θ_2 have a consistent view of the common depended upon thread's history.

6.2 Implementation Details

While the semantics in Section 6.1 interleave the evaluation of many search threads with the evaluation of concurrent threads, there remain a number of impediments to implementation, notably in the EVTCOMM and SYNCCOMMIT rules.

The EVTCOMM rule is problematic for two reasons. First, it requires matching two threads in the thread soup that are attempting to communicate on the same channel. Second, since the search threads corresponding to the sender and the receiver remain in the thread soup, evaluation may repeatedly spawn redundant search threads. Both of these issues are handled in our implementation by representing channels as a lists of suspended search threads for senders and receivers, implemented as `TVar ([Sender a],[Recver a])`. A search thread wishing to send on a channel atomically adds itself to the list of senders and takes a copy of the list of receivers, then forks pairs of new search threads for each coherent thread in the list of receivers; a search thread wishing to receive from a channel behaves similarly. Note that by atomically manipulating the pair of lists, a sending search thread is guaranteed to send to all receivers already on the list of receivers and to be on list of senders for all future receivers.

The SYNCCOMMIT rule is problematic for a number of reasons. First, it requires finding a committable set of completed search threads in the thread soup. Second, it requires removing from the thread soup all other search threads that were searching on behalf of the now synchronized concurrent threads. The second issue is dealt with in our implementation by ensuring that all search threads arising from the same concurrent thread synchronization share a boolean flag, implemented as `TVar Bool`. This boolean flag is allocated, with an initial value of `False`, at the equivalent of the SYNCINIT rule and is copied to each forked search thread. When a set of search threads commit, at the equivalent of the SYNCCOMMIT rule, all of their boolean flags are set to `True`. Search threads periodically check the flag, terminating if the flag is set to `True`. Similarly, the definitions of a committable and coherence are modified to require that the boolean flags of search threads are set to `False`. Finally, we periodically filter the list of senders and list of receivers of a channel of suspended search threads with boolean flags set to `True`; doing so prevents space leaks and limits the number of potential partners that need to be considered at each communication.

Finding a committable set of search threads in the thread soup is the most subtle part of the implementation. Note that a search thread that transitions to a completed search thread may determine a minimal set of thread identifiers required for commitment by consulting its dependencies. If a completed search thread could determine not just a minimal set of thread identifiers required for commitment, but the entire set of completed search threads required for commitment, then it could locally determine a committable set of completed search threads. We make this possible by augmenting each communication element in a transactional event path with two completed search thread lists, implemented as `TVar [Completed]`. These lists maintain the completed search threads that extend the current path and the path of the communication partner. Pairs of these lists are allocated at each communication and are shared by the pairs of search threads forked at a communication. When a search thread transitions to a completed search thread, it atomically adds itself to the completed search thread lists on its path, thereby making itself available for commitment to all of the search threads with which it communicated. Finally, a completed search thread performs *one* atomic scan of the completed search thread lists of its communication partners, attempting to find a committable set of completed search threads. If such a set exists, then the boolean flags of the completed search threads are set to `True` and the completed search threads are transitioned to concurrent threads. If no such set exists, then the completed search thread remains suspended on the completed search thread lists, awaiting commitment to be initiated by another completed search thread.

There remains one minor discrepancy between the semantics of Section 6.1 and our implementation. In the semantics, we use distinguished search threads and explicit evaluation contexts. In the Haskell implementation, we use a CPS-like encoding of the `Evt` monad in order to fork search threads. When a concurrent thread performs a `sync`, it allocates a new `TVar` to return the synchronization result to the concurrent thread, spawns the initial search thread with an initial continuation, and blocks reading the `TVar` for the synchronization result. The interested reader is encouraged to consult the implementation.

7. Related Work

The UniForM Workbench [12, 22] 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 [22] 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 5.2. 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 [16] discuss the algebraic structure of first-class events and the extent to which they form a monad. Their conclusion is that events very nearly form a monad, but the monad laws do not hold under the observability of deadlock. A closer examination of their analysis reveals that the difficulty lies with the `neverEvt` of CML failing to be a right zero of their derived

monadic bind operation. As noted in Section 3, `neverEvt` is a right zero of `thenEvt` in TE Haskell, and `Evt` forms both a monad and a monad-with-plus.

Alan Jeffrey [11] has given a denotational semantics of CML using (variants) of the ideas from Moggi's computational monad program [15]. 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, TE Haskell has two types that denote latent computations.

8. Conclusion

We have introduced transactional events, a novel concurrency abstraction that combines first-class synchronous operations (events) with all-or-nothing transactional semantics. The benefit of this combination is that it admits greater compositionality and modularity in concurrent programming than is available in Concurrent ML. Similarly, by adapting transactional semantics to the context of synchronous message passing, we admit simple implementations of abstractions (such as the `TriSChan` abstraction) that are not (easily) expressible using transactional shared memory.

We believe that there are many directions for future work. On the practical side, we hope to investigate the degree to which transactional events may improve the modularity of and ease the reasoning about applications that naturally fit with synchronous message passing (e.g., graphical user interfaces [20, 3, 22]). Clearly, more powerful abstractions may be designed and implemented with transactional events. Also, we believe that the transactional property of event synchronization obviates the need for the `withNack` combinator in many communication protocols.

On the implementation side, we are interested ways that compiler and run-time support may improve the efficiency of an implementation of TE Haskell. For example, although STM Haskell proved to be immensely useful when developing our "proof-of-concept" implementation, we do not use all of the features provided by STM Haskell. Hence, we may ask: given our stylized use of STM, does a simpler implementation (of either STM or TE) suffice for the same guarantees?

Finally, on the theoretical side, there are interesting questions about the relationship between transactional events and other concurrency calculi (e.g., CSP [10], π-calculus [23]), about the right notions of behavioral equivalence (including a formal demonstration of `Evt` satisfying the requisite monad and monad-with-plus laws), and about progress and fairness in an implementation.

References

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *The Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58, 2004.

[3] Emden R. Gasner and John H. Reppy. A multi-threaded high-order user interface toolkit. In Len Bass and Prasun Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*, chapter 4, pages 61–80. John Wiley & Sons, 1993.

[4] Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[5] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *The Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.

[6] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *The Workshop on Haskell*, pages 49–61, 2005.

[7] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *The Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, 2005.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *The International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

[9] Ralf Hinze. Deriving backtracking monad transformers (functional pearl). In *The International Conference on Functional Programming (ICFP)*, pages 186–197, 2000.

[10] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[11] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *The Symposium on Logic in Computer Science (LICS)*, pages 255–264, 1995.

[12] Einar Karlsen. The UniForM concurrency toolkit and its extensions to Concurrent Haskell. In *The Glasgow Functional Programming Workshop (GFPW)*, 1997.

[13] Oleg Kiselyov, Chung-chieh Shan, Daniel Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *The International Conference on Functional Programming (ICFP)*, pages 192–203, 2005.

[14] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *The Conference on Programming Language Design and Implementation (PLDI)*, pages 274–285, 2001.

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

[16] 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.

[17] 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.

[18] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *The Symposium on Principles of Programming Languages (POPL)*, pages 295–308, 1996.

[19] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *The Symposium on Principles of Programming Languages (POPL)*, pages 71–84, 1993.

[20] Rob Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 1989.

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

[22] George Russell. Events in Haskell, and how to implement them. In *The International Conference on Functional Programming (ICFP)*, pages 157–168, 2001.

[23] Davide Sanggiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[24] Franklyn Turbak. First-class synchronization barriers. In *The International Conference on Functional Programming (ICFP)*, pages 157–168, 1996.

[25] 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.

[26] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *The European Conference on Object-Oriented Programming (ECOOP)*, pages 519–542, 2004.