JFP **18** (5 & 6): 649–706, 2008. © 2008 Cambridge University Press
doi:10.1017/S0956796808006916 Printed in the United Kingdom

Transactional events[1]

KEVIN DONNELLY
Boston University, Boston, MA 02215, US
(*e-mail:* kevind@cs.bu.edu)

MATTHEW FLUET[2]
Toyota Technological Institute at Chicago, Chicago, IL 60637, US
(*e-mail:* fluet@tti-c.org)

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. As another example, three-way rendezvous can 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 transactional events and give a detailed account of an implementation.

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 increasingly common presence of multiple cores or processors on a single machine.

[1] This is a revised and extended version of the paper that appeared in the *Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)* (Donnelly & Fluet, 2006).
[2] Portions of this work were completed while the author was affiliated with Cornell University, Ithaca, NY 14853, US.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Given the usefulness of concurrent programs, it is important to provide programmers with abstraction mechanisms that help manage the complexity of reasoning about such programs. One means of managing this complexity, exemplified by both software transactional memory (STM) Haskell (Harris *et al.*, 2005b) and Concurrent ML (CML) (Reppy, 1999), is to introduce first-class, composable operations that encapsulate a particular concurrent programming paradigm: shared-memory transactions in STM Haskell and synchronous message passing in CML. Such first-class, composable operations allow complex thread interactions to be abstractly packaged and exported, which increase modularity and ease reasoning.

STM Haskell uses monadic computations as a means of structuring shared-memory transactions, which ensures that isolated sequences of shared-memory operations 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 the interactions of concurrent threads. However, the same isolation guarantees preclude the use of STM transactions for some thread interactions, like synchronous message passing.

Concurrent ML uses first-class events as means of structuring synchronous message passing. Like an STM computation, a CML event is an abstractly packaged group of actions. However, the act of synchronizing on an event is not an isolated operation, because events may include synchronous message passing. Events can be naturally composed as alternatives, but sequential composition of events can be difficult to reason about. Because synchronous message passing is strictly more powerful than asynchronous message passing, there are abstractions, such as swap channels, that may be implemented as CML events but have no implementation as STM transactions. However, the 'all-or-nothing' (atomic) semantics of STM computations can allow certain compositions to be more easily expressed and can lead to greater modularity.

In this work, we explore a novel 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. 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 ease reasoning and make 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 input/output (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,

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

transactional shared memory and three-way synchronous rendezvous. We also consider the computational complexity of any implementation of transactional events by giving encodings of the Boolean satisfiability problem as concurrent programs using transactional events.

- We give a detailed description of a prototype implementation of transactional events as a library for Haskell, built using the STM Haskell extensions available in the Glasgow Haskell Compiler (GHC).

The remainder of the paper is structured as follows. Section 2 briefly reviews CML 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 describes 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 high-level language, formulated as an extension of Standard ML, that provides threads and synchronous message passing. In its implementation as a library for Standard ML, the CML Library (Appendix A of Reppy 1999) provides types for thread identifiers (`thread_id`), synchronous events (`'a event`) and synchronous channels (`'a chan`), as well as the following operations:

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

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

val sync      : 'a event -> 'a
val choose    : 'a event list -> '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 alwaysEvt : 'a -> 'a event
val 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.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

We briefly describe the CML operations below:

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

`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 (i.e., when there is a matching sender), and yields the received message.

`sync ev` synchronizes on the event `ev`. This operation blocks until some constituent primitive event becomes enabled. This operation returns the synchronization result of the enabled event.

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

`wrap (ev, f)` creates an event which, if `ev` is enabled and 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 enabled and selected for synchronization.

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

`never` creates an event which is never enabled.

2.1.1 Examples

As a simple example, we consider the implementations of an event value that represents communication between a client thread and two server threads. If a client wishes to send 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 () => sync (recvEvt (replyCh1))),
  wrap (sendEvt (req2, serverCh2),
        fn () => sync (recvEvt (replyCh2))) ]
```

In this example, the choice is between the communications `sendEvt (req1, serverCh1)` and `sendEvt (req2, serverCh2)`; when synchronizing on this event, the client will send a request to exactly one server, after which the client will block waiting to receive a reply.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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

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

In this example, the choice is between the communications `recvEvt (replyCh1)` and `recvEvt (replyCh2)`; when synchronizing on this event, the client will send requests to both servers (blocking until both servers receive the requests), after which the client will receive a reply from exactly one server.

Of course, there is no requirement that the client make use of the same communication pattern with both servers:

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

Here, the choice is between the communications `recvEvt (replyCh1)` and `sendEvt (req2, serverCh2)`; when synchronizing on this event, the client will send a request to the first server (blocking until the first server receives the request), after which the client will either receive a reply from the first server or will send a request to the second server and block waiting to receive a reply from the second server.

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

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

In general, one often wants to implement a protocol consisting of a sequence of communications $c_1; c_2; \ldots; c_n$. 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 terminate in order for `guard` to yield the commit point communication; likewise, all of the post-synchronous communications must terminate in order for `wrap` to yield the synchronization result. One must be careful to maintain program invariants after pre-synchronous actions, since the corresponding commit point communication and post-synchronous action may not be chosen. This motivates the need for `withNack`, as a mean to signal compensating actions in non-chosen protocols.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

2.2 Limitations of CML events

Having a single communication commit point limits the expressive power of CML events. In CML, given the operations for two-way synchronization (i.e., `sendEvt` and `recvEvt`), there is no way to implement three-way synchronization as an event abstraction (Panangaden & Reppy, 1997; Reppy, 1999). As shown in Section 5.4, TE Haskell allows for an abstract implementation of such three-way synchronous operations.

The fact that all CML events must have a single commit point also places a limit on the modularity achievable with these events. Consider the example of trying to program guarded (or conditional) 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 `m` from `ch`, but only if `g m` 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. In TE Haskell, guarded receive can be given a transparently correct implementation with just a few lines of code (see Section 5.1).

Guarded receive can be implemented in CML, but not as a modular, composable event abstraction. It requires a fairly complicated protocol in which the sender and the receiver cooperate and the use of a special guarded-receive channel abstraction.

To see that the guarded-receive communication protocol cannot be implemented as a composable event abstraction that can participate in non-deterministic choice, suppose there is a sender *S*, which wishes to send a value `m : 'a` and a receiver *R*, which wishes to receive with a guard `g : 'a -> bool`. Either *S* and *R* must interact directly in order to decide whether the event commits or there needs to be an intermediate server. Suppose an intermediate server is used. At some point in the protocol, the server would have to simultaneously commit to notifying *S* that its send was accepted and to sending `m` to *R* (or to notifying *R* that its previous receipt of `m` can be committed). However, this requires three-way synchronization, which itself is not expressible as an event abstraction. Therefore, *S* and *R* must interact directly to decide whether to commit to a communication. Clearly the first communication between *S* and *R* cannot be the commit point, because there is no way to evaluate `g m` before the first communication. So the protocol must start with an unconditional communication between *S* and *R*. But this means that if there is a sender with no receiver or a receiver with no sender then this unconditional communication will block. Consequently, synchronizing on a `choose` event that includes a guarded send or guarded receive will block, even if another event included in the `choose` is enabled and could be chosen for synchronization. This destroys the composability of the implementation.

Requiring a protocol to implement guarded receive also 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 guarded-receive channel 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.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

2.3 Concurrent Haskell

Concurrent Haskell (Peyton Jones *et al.*, 1996) is an extension of Haskell with concurrency primitives. Following the Haskell tradition, Concurrent Haskell isolates the side-effect producing concurrency primitives in the I/O monad (Peyton Jones & Wadler, 1993; Peyton Jones, 2001). We assume that the reader is familiar with the style of monadic I/O employed by Haskell, as well as with the `do`-notation used for monadic actions. In examples, we will make use of I/O actions that read and write a character, `getChar` and `putChar`, which have 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. A thread may also query its own identifier:

```
myThreadId :: IO ThreadId
```

As in Haskell, 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; a Concurrent Haskell program continues to perform the I/O actions of any concurrently spawned thread. For example, here is a program (using the `do`-notation) 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 output in a non-deterministic order, due to the non-deterministic scheduling of concurrent threads.

3 Transactional events

By introducing transactional events, we overcome the limitations of CML's single communication commit point requirement. In essence, we allow an event synchronization to take place in a transaction that has no observable effect until all of the required events are enabled and successfully complete. Transactions, an idea originally from the world of databases, have recently found use as a powerful technique to ensure atomicity in concurrent programming (Herlihy & Moss, 1993; Harris & Fraser, 2003; Welc *et al.*, 2004; Harris *et al.*, 2005b). We adapt transactions to the setting of synchronous message passing, yielding gains in modularity and expressiveness.

A transaction groups tentative actions made during a computation, which can be committed if the actions complete successfully and can be aborted otherwise. In

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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. To put it another way, all of the actions of a committed transactions are observed by the rest of the system or program to have been taken as a single atomic action.

Due to this all-or-nothing semantics, transactions provide a straightforward way of overcoming the limitation of single communication commit points in CML. We view event synchronization as a transaction where *multiple* synchronous message passing communications may be grouped as tentative actions. Since message passing is synchronous (every send must be matched by a receive), communication between threads has the effect of merging their event synchronization transactions (so that they either commit or abort together). This makes it 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. Haskell provides an ideal setting for programming with transactional events because the use of monadic I/O makes it easy to keep irrevocable side-effects out of transactional events, which may otherwise need to be aborted. Monads also turn out to be the right notion for describing the compositional structure of transactional events. We dub our extension 'TE Haskell'.

The basic interface for transactional events is given in Figure 1. An `Evt a` is a transactional event: an abstract synchronous operation that yields a result of type `a` when synchronized upon. An `SChan a` is a synchronous channel used for message passing, where `a` is the type of messages. 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 transactional event synchronizations.[3] 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 }
```

[3] This enables the useful idiom of creating reply channels that are local to a synchronization.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
-- The Evt monad
data Evt a

-- Synchronization
sync :: Evt a -> IO a

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

-- Monadic event combinators
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

-- Thread identity
myThreadIdEvt :: Evt ThreadId

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

Fig. 1. The `TxEvent` 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 `evt `thenEvt` f` is the event which tentatively synchronizes on the event `evt`, yielding the result `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

```
evt1 = do { sendEvt ch 0 ; sendEvt ch 1 }
```

This event may only successfully complete synchronization if both sends are successfully received. For example, it may synchronize if another thread is synchronizing on the following event:

```
evt2 = do { a <- recvEvt ch ; b <- recvEvt ch }
```

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

It may also synchronize if two other threads are synchronizing on the event `recvEvt ch`.

The event `alwaysEvt e` is an event that immediately yields `e` when synchronized upon. Note that `alwaysEvt` is a left and right unit of `thenEvt`; hence, `Evt a` forms a monad (Wadler, 1995), with `alwaysEvt` as the monadic unit (`return`).

Events may also be composed as non-deterministic alternatives using the `chooseEvt` combinator. The event `evt1 `chooseEvt` evt2` synchronizes as either `evt1` or `evt2`, 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

```
evt3 = (do { sendEvt ch 0 ; sendEvt ch 1 })
       `chooseEvt`
       (do { a <- 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 `evt2` or with a thread synchronizing on the following event:

```
evt4 = sendEvt ch 2
```

Note, however, that if there are threads synchronizing on both `evt2` and `evt4`, then the event with which `evt3` synchronizes is non-deterministic; hence, `chooseEvt` is a commutative combinator.

The event `neverEvt` is an event which never successfully completes when synchronized upon. This is analogous to an explicit abort action in a transaction; it can be used to indicate that the sequence of actions that led to the `neverEvt` should not be taken as a synchronization. For example, the event which receives two equal integers on the same channel is

```
evt5 = do { a <- recvEvt ch ; b <- recvEvt ch
          ; if a == b then alwaysEvt () else neverEvt }
```

Note that this event cannot synchronize with a thread synchronizing on `evt1`, but can synchonize with two threads synchronizing on `evt4`.

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 (Wadler, 1995; Hinze, 2000; Kiselyov *et al.*, 2005; MonadPlus, 2005), with `chooseEvt` as monadic plus (`mplus`) and `neverEvt` as monadic zero (`mzero`).

The event `myThreadIdEvt` is an event which immediately yields the thread identifier of the synchronizing thread. It is useful to replicate the `myThreadId` operation with an event type in order to create synchronous event abstractions that implement protocols that are sensitive to the identities of the participating threads; see Section 5.3 for an example.

The semantics of transactional events requires that a synchronization commits to a particular alternative in a `chooseEvt` only if all of the constituent events

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

in that alternative can successfully complete. The semantics also requires that a synchronization commits to a communication partner in a `sendEvt` or `recvEvt` only if 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; tentative synchronization actions are aborted when such a collection may not successfully complete. A thread performing a synchronization blocks until a collection of choice alternatives and communication partners that may successfully complete can be found.

Because synchronous message passing involves other transactional event synchronizations, 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.

We discuss issues of implementation in more detail in Section 6. For now, though, we note that the most important property we will desire of an implementation is a progress guarantee with respect to synchronizations, namely, if a collection of choice alternatives and communication partners that successfully complete exists for some groups of threads blocked performing synchronizations, then the implementation must eventually find some such collection, effect the choices and communications, successfully complete the synchronizations and unblock the threads. Note that this progress guarantee does not imply that every thread blocked performing a synchronization will eventually be unblocked, nor does it imply that every group of threads blocked performing synchronizations will eventually have some sub-group unblocked; for example, a program with a single thread that synchronizes on a `sendEvt` will never be unblocked (since, there being only one thread in the program, there will never be another thread synchronizing on a matching `recvEvt`).

On the other hand, this progress guarantee does imply that an implementation should have properties akin to deadlock-freedom and livelock-freedom with respect to synchronizing events. For example, consider two threads synchronizing on the following events:

```
evtA = (sendEvt ch1 ()) `chooseEvt` (recvEvt ch2)
evtB = (sendEvt ch2 ()) `chooseEvt` (recvEvt ch1)
```

Synchronizations on these two events may successfully complete together, either with `evtA` choosing to send and `evtB` choosing to receive or vice-versa. Hence, an implementation should neither have both events choose to send and then wait indefinitely for the other to choose to receive (i.e., deadlock), nor loop indefinitely with both events choosing to send and then both events choosing to receive (i.e., livelock).

3.1 Exceptions in transactional events

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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

is necessary to specify the semantics of exceptions in transactional events and in synchronizations. The treatment of exceptions in TE Haskell is somewhat subtle. While the issues regarding the use of exceptions in shared-memory transactions are well known, the fact that multiple threads may interact through synchronous message passing during event synchronization makes the issue deserving of further consideration. Indeed, it is precisely the fact that an exception may be thrown as a consequence of the receipt of a synchronous message (a situation that may never arise in shared-memory transactions) that adds new dimensions to the issue.

Asynchronous exceptions (Marlow *et al.*, 2001) 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 evt` 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 thread synchronizations:

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

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

If the synchronization in `t1` aborts with the exception `Foo` (caused by the tentative send of `0` on the channel `ch`), 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 ch
               ; if i == 0 then throw Foo
                           else return i })
           `chooseEvt` (sendEvt ch 0))

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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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 given 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 its 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 (with `t6` raising an exception) without `t7` completing its synchronization. Allowing exceptions that propagate to the top level of an event synchronization to be treated as a synchronization 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 threads `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 given 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 evt h` is the event that acts as `evt`, unless synchronizing on `evt` throws an exception `ex`, in which case 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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

non-local control flow they are introducing does not destroy any important mutual commitment properties.

As an aside, in Haskell, a language with lazy evaluation and imprecise exceptions (Peyton Jones *et al.*, 1999), one must be aware that `catchEvt` only catches exceptions that are raised when evaluating expressions to event values during synchronization. In particular, it does not catch exceptions that are thrown by unevaluated expressions returned as a synchronization result. For example, consider the following synchronizing thread:

```
t8 = sync (catchEvt (alwaysEvt (throw (AssertionFailed "foo")))
                    ( _ -> alwaysEvt 3))
```

This thread will synchronize, returning a lazy computation that will raise an exception if evaluated. As another example, consider the following synchronizing thread:

```
t9 = sync (catchEvt (do { i <- recvEvt ch
                        ; if i < 10
                            then throw (AssertionFailed "foo")
                            else alwaysEvt i })
                    ( _ -> alwaysEvt 10))
```

This thread will synchronize (assuming a matching sender on the channel `ch`), always returning a value greater than or equal to 10, since the exception is thrown when evaluating the `if ...` expression to an event value. Indeed, thread `t9` will synchronize with the following thread:

```
t10 = sync (sendEvt ch (throw (AssertionFailed "bar")))
```

which sends a lazy computation that raises an exception when evaluated by the `i < 10` expression in `t9`.

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 CML, 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 2 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 and applications), which we omit.

The event combinators should be familiar 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`.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Values

v	::=	c	characters
	|	θ	thread identifiers
	|	κ	channel names
	|	`alwaysEvt` e | `thenEvt` e' e_f	
	|	`neverEvt` | `chooseEvt` e_l e_r	
	|	`throwEvt` e | `catchEvt` e' e_h	
	|	`newSChan` | `recvEvt` κ | `sendEvt` κ e'	
	|	`myThreadIdEvt`	
	|	`unitIO` e | `bindIO` e' e_f	
	|	`throwIO` e | `catchIO` e' e_h	
	|	`getChar` | `putChar` c	
	|	`myThreadId` | `forkIO` e | `sync` e	
	|	`\`x`->`e | ...	functional language values

Expressions

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

Fig. 2. 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. We have in mind the call-by-need evaluation strategy employed by Haskell. However, order of evaluation for pure terms (whether call-by-value, call-by-name or call-by-need) has no real impact on the behavior of transactional events.

Although, we do not do so in this presentation, it would be straightforward to incorporate a sequential evaluation relation that indicates exceptions raised by the evaluation of functional language terms. Again, the exact nature of the evaluation (whether raised exceptions are precise or imprecise (Moran *et al.*, 1999)), has no real impact on the behavior of transactional events.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Synchronous Evaluation Contexts

$$M^{Evt} \quad ::= \quad [] \mid \texttt{thenEvt}\ M^{Evt}\ e_f \mid \texttt{catchEvt}\ M^{Evt}\ e_h$$

EvtEval

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

EvtThenAlways

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{thenEvt (alwaysEvt } e')\ e_f] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[e_f\ e'] \rangle\}$$

EvtThenThrow

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{thenEvt (throwEvt } e')\ e_f] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{throwEvt } e'] \rangle\}$$

EvtChooseLeft

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{chooseEvt } e_l\ e_r] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[e_l] \rangle\}$$

EvtChooseRight

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{chooseEvt } e_l\ e_r] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[e_r] \rangle\}$$

EvtCatchAlways

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{catchEvt (alwaysEvt } e')\ e_h] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt } e'] \rangle\}$$

EvtCatchThrow

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{catchEvt (throwEvt } e')\ e_h] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[e_h\ e'] \rangle\}$$

EvtNewSChan

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

EvtSendRecv

$$\mathcal{S} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{sendEvt } \kappa\ e'] \rangle, \langle \theta_r, M_r^{Evt}[\texttt{recvEvt } \kappa] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{alwaysEvt ()}] \rangle, \langle \theta_r, M_r^{Evt}[\texttt{alwaysEvt } e'] \rangle\}$$

EvtMyThreadId

$$\mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{myThreadIdEvt}] \rangle\} \rightsquigarrow \mathcal{S} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt } \theta] \rangle\}$$

Fig. 3. TE Haskell: Dynamic semantics – Synchronous evaluation.

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:

$$\begin{array}{lrcl} \textit{Synchronizing Event} & S & ::= & \langle \theta, e \rangle \\ \textit{Synchronization Groups} & \mathcal{S} & ::= & \{S, \ldots\} \end{array}$$

The synchronous evaluation relation is closely related to the event matching relation in the semantics of Concurrent ML (Appendix B of Reppy 1999). 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 reducing into the events $e_1',\ldots,e_k'$. The complete set of rules is given in Figure 3. All of

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

the rules include non-deterministically choosing one or more events for a step of evaluation.

The rule EvtEval implements the sequential evaluation of an expression in the active position.

The rule EvtThenAlways implements sequential composition in the `Evt` monad. The rules EvtChooseLeft and EvtChooseRight 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.[4] The rule EvtSendRecv implements the two-way rendezvous of communication along a channel; note that the transition replaces the `sendEvt` and `recvEvt` events with `alwaysEvt` events. Finally, the rule EvtMyThreadId simply returns the thread identifier of the synchronizing event.

It is worth considering the possible terminal configurations for a set of events under the synchronous evaluation relation. 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 one 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 group of concurrent threads as a set of pairs of thread identifiers and `IO` expressions:

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

To model the I/O behavior of the program, transitions are labelled with an 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 I/O behavior. In a real language, there would be many other observable I/O actions.

The complete set of rules is given in Figure 4. All of the rules include non-deterministically choosing one or more threads for a step of evaluation. Most of the rules are entirely standard for a dynamics of Haskell or Concurrent Haskell (Peyton Jones, 2001); we include them simply for completeness.

The rule IOEval implements the sequential evaluation of an expression in the active position. 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

[4] This freshness condition could be formalized by a synchronous evaluation relation of the form $\mathcal{K}; \mathcal{S} \rightsquigarrow \mathcal{K}'; \mathcal{S}'$, where $\mathcal{K}$ is a set of allocated channel names.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Concurrent Evaluation Contexts

$$M^{IO} \quad ::= \quad [] \mid \texttt{bindIO}\ M^{IO}\ e_f \mid \texttt{catchIO}\ M^{IO}\ e_h$$

IOEval

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

IOBindUnit

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{bindIO (unitIO}\ e')\ e_f] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[e_f\ e'] \rangle\}$$

IOBindThrow

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{bindIO (throwIO}\ e')\ e_f] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{throwIO}\ e'] \rangle\}$$

IOCatchUnit

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{catchIO (unitIO}\ e')\ e_h] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ e'] \rangle\}$$

IOCatchThrow

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{catchIO (throwIO}\ e')\ e_h] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[e_h\ e'] \rangle\}$$

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

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

IOMyThreadId

$$\mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{myThreadId}] \rangle\} \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta, M^{IO}[\texttt{unitIO}\ \theta] \rangle\}$$

IOFork

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

IOSync

$$\frac{\{\langle \theta_1, e_1 \rangle, \ldots, \langle \theta_k, e_k \rangle\} \rightsquigarrow^* \{\langle \theta_1, \texttt{alwaysEvt}\ e'_1 \rangle, \ldots, \langle \theta_k, \texttt{alwaysEvt}\ e'_k \rangle\}}{\begin{array}{l}\mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{sync}\ e_1] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\texttt{sync}\ e_k] \rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{T} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e'_1] \rangle, \ldots, \langle \theta_k, M_k^{IO}[\texttt{unitIO}\ e'_k] \rangle\}\end{array}}$$

Fig. 4. TE Haskell: Dynamic semantics – Concurrent evaluation.

perform the appropriately labelled transition, yielding an observable action. The IOMyThreadId rule returns the thread identifier of the active thread.

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.[5]

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

[5] This freshness condition, along with the freshness condition in the EvtNewSChan rule of the synchronous evaluation relation, could be formalized by a concurrent evaluation relation of the form $\Theta; \mathcal{K}; \mathcal{T} \xrightarrow{a} \Theta'; \mathcal{K}'; \mathcal{T}'$, where Θ is a set of allocated thread identifiers and $\mathcal{K}$ is a set of allocated channel names.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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 back 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, 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 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.

4.3.1 Monad laws

It should be clear that the `IO` type constructor of TE Haskell forms a monad and the `Evt` type constructor of TE Haskell forms a monad-with-plus.

The former follows directly from the fact that the IOBindUnit 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 EvtBindAlways 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 EvtChoose1 and EvtChoose2.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

The monad laws 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 a right identity for `chooseEvt` and that `neverEvt` is a left and a 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.

4.3.2 Spirit of Concurrent ML

We may also see that TE Haskell preserves the 'spirit' of Concurrent ML. Recall from Section 2.1 that we often want to implement a protocol consisting of a sequence of communications: $c_1; c_2; \ldots; c_n$. 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 $c_1; \ldots; c_{i-1}$ and $c_{i+1}; \ldots; c_n$).[6]

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.

4.3.3 Sophisticated transactional concepts

We may also compare TE Haskell's treatment of event synchronization as a transaction to more sophisticated transactional concepts in the literature. One such concept is that of nested transactions (Moss, 1985): a top-level transaction is divided into a number of child transactions; child transactions must commit before their parent transaction, but child transactions that abort need not abort their parent transaction. Effects of parent transactions are visible to their child transactions, but effects of child transactions are visible to their parent transaction only after the child commits. Transactional events naturally express nested transactions via the `thenEvt` and `chooseEvt` combinators: these events only synchronize if the appropriate constituent events synchronize; furthermore, the inability of one alternative in a `chooseEvt` to synchronize does not make the `chooseEvt` unable to synchronize – it may synchronize via the other alternative.

Another such concept is that of multi-threaded transactions: a top-level transaction may spawn threads within the transaction; all spawned threads must complete (and commit) before the top-level transaction may commit. The semantics of TE Haskell given in this section does not allow *any* I/O actions to take place during an event synchronization; in particular, it is not possible to fork new threads as part of

[6] Nonetheless, one may still implement a sequence of communications with a dedicated commit point; see Section 5.2.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

an event synchronization. Hence, as formulated, TE Haskell prohibits the expression of this sort of multi-threaded transaction.[7]

The separation of I/O actions from event synchronization is motivated by a desire to ensure that truly irrevocable (i.e., observable) actions cannot take place during event synchronization. Forking a thread only has an observable effect when the forked thread takes an observable action; recall that the IOFork rule has a silent action. Hence, it should suffice to ensure that threads spawned within an event synchronization cannot take observable actions until the synchronization can successfully complete. At the same time, spawned threads should interact in a non-trivial manner with the event synchronization, else there is little reason to spawn them *within* an event synchronization.

We believe that the following proposal could form the basis of a coherent account of multi-threaded transactions in the context of transactional events. We extend the set of transactional event combinators with a combinator to spawn a thread as a synchronization action:

```
forkEvt :: Evt a -> (a -> IO ()) -> Evt ThreadId
```

The idea is that `forkEvt evt f` spawns a thread to synchronize on `evt`, and, upon successful synchronization with result `r`, the spawned thread goes on to act as `f r`. The spawnee is required to synchronize in the same synchronization group as the spawner; that is, the spawnee is only allowed to persist if the spawner successfully synchronizes. We may think of this as the spawner and the spawnee having an implicit communication dependency, which ensures that the two synchronizations commit or abort together. The expression `forkEvt evt (\() -> return ())` corresponds to the special case when the spawned thread immediately terminates after synchronization, which is closer to the account of multi-threaded transactions given earlier. We hope to formalize and elaborate on this proposal in future work.

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 CML (Section 5.2). Interestingly, we may also easily encode transactional shared memory (Section 5.3). Next, we demonstrate that TE Haskell is strictly more powerful than CML by encoding an abstract three-way rendezvous operation (Section 5.4). Finally, we show that scheduling of transactional events is NP-hard by encoding boolean satisfiability (Section 5.5).

Throughout this section, we will make extensive use of Haskell's `do`-notation for monadic computations.

[7] On the other hand, a different sort of multi-threaded transaction is required: multiple threads must synchronize simultaneously in order to enable synchronous message passing along channels.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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 (or conditional) 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 (Armstrong *et al.*, 1996) and CSP (Hoare, 1978), 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 can be done neither in CML nor 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 as follows:

```
(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 guarded communication using a special guarded-receive channel abstraction and guarded send and receive operations, having the following signature:

```
signature GUARD_CHAN = sig
  type 'a gchan

  val gchan : unit -> 'a gchan
  val gsend : ('a gchan * 'a) -> unit
  val grecv : ('a -> bool) * 'a gchan -> 'a
end
```

This approach is much less modular because guarded receive can only be performed with the cooperation of the sender. Also, as discussed in Section 2.2, the guarded send and receive operations cannot be implemented as well-behaved CML events that compose with non-deterministic choice; note that the result type of `gsend` and `grecv` are `unit` and `'a`, rather than `unit event` and `'a event`.

Unlike the CML version, the TE Haskell implementation of guarded receive is also kill-safe (Flatt & Findler, 2004): if the reading thread is killed and the event

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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 a CML implementation, while possible, requires an additional layer of complexity in the implementation.

5.2 Encoding Concurrent ML

5.2.1 Simple CML

We first 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 we will shortly show that `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 (Moggi, 1991)), we note that the general idea is to translate a function with side-effects 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 CML 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[8]:

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

The encodings of the CML combinators are given in Figure 5. 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.

5.2.2 Acknowledgement variables

Before giving an encoding for CML with the `withNack` combinator, we present a simple *acknowledgement variable* abstraction (see Figure 6). An acknowledgement

[8] In the lifting function and the encodings, we make use of the `fmap` and `join` operations, which may be defined as follows, 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 ; x <- m ; return x }
```

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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

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

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

neverCMLEvt :: CMLEvt a
neverCMLEvt = lift (neverEvt)

chooseCMLEvt :: CMLEvt a -> CMLEvt a -> CMLEvt a
chooseCMLEvt io_evt_io1 io_evt_io2 =
  do { evt_io1 <- io_evt_io1
     ; evt_io2 <- io_evt_io2
     ; return (evt_io1 `chooseEvt` evt_io2) }

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 io_evt_io =
  do { evt_io <- io_evt_io
     ; io <- sync evt_io
     ; io }
```

Fig. 5. Simple CML encoding.

variable is a variable that may be asynchronously enabled and that may be synchronously queried, blocking synchronizations until the variable is enabled. Since an acknowledgement variable may not be disabled, note that there is no harm in enabling an acknowledgement variable multiple times.

The abstraction is quite simple to implement. An acknowledgement variable is simply a channel carrying `()`. Creating a new acknowledgement variable amounts to creating a new channel. Enabling an acknowledgement variable requires spawning a thread that repeatedly sends `()` along the channel. The event value for synchronizing on an acknowledgement variable is the receive event.

5.2.3 Full CML

To encode the `withNack` combinator of CML, we simply augment the 'outer' `IO` action with a list of the acknowledgement variables created for constituent `withNack`

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
type AckVar = SChan ()

newAckVar :: IO AckVar
newAckVar = sync newSChan

setAckVar :: AckVar -> IO ()
setAckVar a =
  let loopIO = do { sync (sendEvt a ()) ; loopIO } in
  do { forkIO loopIO ; return () }

getAckVarEvt :: AckVar -> Evt ()
getAckVarEvt a = recvEvt a
```

Fig. 6. Acknowledgement variable implementation.

events and augment the `Evt` action with a list of the acknowledgement variables to enable:

```
type CMLEvt a = IO ([AckVar], Evt ([AckVar], IO a))
```

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

```
lift :: Evt a -> CMLEvt a
lift evt = return ([], fmap ( x -> ([], return x)) evt)
```

which may be used to encode `alwaysCMLEvt`, `neverCMLEvt`, `recvCMLEvt` and `sendCMLEvt`. Similarly, the encoding of `wrapCMLEvt` simply augments the post-synchronous action, without changing the acknowledgement variables:

```
wrapCMLEvt :: CMLEvt a -> (a -> IO b) -> CMLEvt b
wrapCMLEvt io_evt_io f =
  fmap (\(acks, evt_io) -> (acks, fmap (\(acks, io) -> (acks, io >>= f))
                                       evt_io))
       io_evt_io
```

More interesting is the encoding of `withNackCMLEvt`:

```
withNackCMLEvt :: (CMLEvt () -> IO (CMLEvt a)) -> CMLEvt a
withNackCMLEvt f =
  do { ack <- newAckVar
     ; let io_io_evt_io = f (lift (getAckVarEvt ack))
     ; (acks, evt_io) <- join io_io_evt_io
     ; return (ack:acks, evt_io) }
```

A new acknowledgement variable is created and provided to the function `f`, which (after performing arbitrary side effects) returns an `IO` action yielding a list of acknowledgement variables and an event value. The new acknowledgement variable is added to the list of acknowledgement variables and the event value is returned unmodified.

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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
chooseCMLEvt :: CMLEvt a -> CMLEvt a -> CMLEvt a
chooseCMLEvt iei1 iei2 =
  do { (acks1, ei1) <- iei1 ; (acks2, ei2) <- iei2
     ; (fmap (\(acks, i) -> (acks2 ++ acks, i)) ei1)
       `chooseEvt`
       (fmap (\(acks, i) -> (acks1 ++ acks, i)) ei2) }
```

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

```
syncCML :: CMLEvt a -> IO a
syncCML io_evt_io =
  do { (_, evt_io) <- io_evt_io
     ; (acks, io) <- sync evt_io
     ; loopAckVar acks
     ; io }
  where loopAckVar [] = return ()
        loopAckVar (ack:acks) = do { setAckVar ack
                                   ; loopAckVar acks }
```

Note that the sequencing in the `IO` monad ensures that the 'inner' `IO` action of the event encoding is performed after all of the acknowledgement variables have been enabled.

5.2.4 Discussion

There is an interesting consequence of this encoding of Full CML. Note that the encoding only makes use of `thenEvt` through the `fmap` operation. Furthermore, note that when `fmap` (in the `Evt` monad) is applied to a pure, terminating function (as it is in all of the functions defined earlier), then CML's `wrap` combinator provides essentially the same functionality. This suggests that neither `withNack` nor `guard` need be taken as primitive operations in CML; rather they may be expressed as a stylized use of the other primitive CML operations. This simplifies both the meta-theory of CML and the implementation of CML, without changing the expressive power of CML.

This 'reverse encoding' requires recognizing that the above use of Haskell's `IO` monad for pre- and post-synchronization actions may be approximated by thunks in Standard ML. It is straightforward to apply this idea to give an implementation of the complete set of CML combinators (i.e., including `withNack` and `guard`) using only a primitive set of CML combinators (i.e., excluding `withNack` and `guard`). In 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 implementation of the primitive combinators will be simpler and possibly faster. We also note that since the primtive `wrap` combinator is only applied to pure, terminating functions, then one need not be pessimistic about their behavior; in particular, it is not strictly necessary to run the post-synchronous functions passed to the primitive `wrap` combinator outside of critical regions.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

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 CML (Sections 3.2.3 and 3.2.7 of Reppy, 1999). Since transactional events extend CML synchronizations 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 (Harris *et al.*, 2005b). 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 access 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 the `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 is 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.[9] 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 7 gives the encoding, which we discuss in some detail. The high-level view of the encoding is quite simple. Recall that each transactional variable will

[9] Using the `forkEvt` combinator proposed in Section 4.3.3, we could avoid making this change and give `newTVar` the type `a -> STM (TVar a)`.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
type STM a = Evt a

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

type 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) }
```

Fig. 7. Transactional shared memory encoding.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

be represented by a server thread. A thread wishing to read or write transactional 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 given earlier, 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, which may be obtained using `myThreadIdEvt`.

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`.

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` 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 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 `TVars`, 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.

Finally, we note that this encoding simulates the high-level operational behavior of transactional shared memory, but not the low-level run-time artifacts commonly

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

found in implementations. For example, there is no explicit transaction log and there are no explicit abort or commit operations. Rather, validation of a serializble transaction is distributed throughout the synchronization, via the comparison of thread identifiers by the server threads. Aborts are implicit in the transition of a server thread to a `neverEvt` (indicative of non-serializable accesses by different threads), which prevents a successful synchronization. Similarly, commits are implicit in a successful synchronization.

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 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). (Section 6.4 of Reppy 1999).

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 (with $n > 1$), 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 8).

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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
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) }
```

Fig. 8. The `TriSChan` abstraction.

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.

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` yields a synchronous channel for swapping among n threads. With this interface, we may easily encode first-class synchronization barriers (Turbak, 1996).

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

5.5 Boolean satisfiability

In this section, we show that the problem of deciding whether the IOSync rule can be applied to a given set of synchronizing events is NP-hard by encoding the boolean satisfiability problem. Boolean satisfiability (SAT) is the problem of determining whether there exists a satisfying assignment to a classical propositional formula in n variables and is NP-complete. Encoding of SAT using `chooseEvt` is completely straightforward. However, the problem is NP-hard for the case of three or more transactions even if we exclude the use of `chooseEvt`.

To show this, we give two encodings of boolean satisfiability. The first encoding uses three transactional events, none of which use the `chooseEvt` combinator, and is exponential in the number of possible communications over a particular channel. The second encoding requires only one transactional event, which does not use the communication combinators, and is exponential in the number of `chooseEvt` combinators. These encodings show that implementing transactional events inherently requires combinatorial search, and that there are at least two sources of such behavior: determining which branch of a `chooseEvt` to take, and determining how to match senders and receivers on a channel.

We assume an abstract data type `Formula` of boolean formulas in n variables numbered 0 through $n-1$ and a function

```
evalFormula :: [Bool] -> Formula -> Bool
```

which, given a list of booleans of length n and a formula of at most n variables, decides the truth of the formula where the ith variable is replaced by the ith boolean in the list. Each encoding of SAT is given as a function

```
sat :: Int -> Formula -> IO ()
```

that takes an integer n and a formula in at most n variables; the function prints 'Satisfiable' if there exists a satisfying assignment and prints nothing otherwise. The first encoding is given by

```
sat n formula =
  do { c <- sync newSChan
     ; forkIO (sync (mapM_ (\_ -> sendEvt c True) [1..n]))
     ; forkIO (sync (mapM_ (\_ -> sendEvt c False) [1..n]))
     ; sync (do { input <- mapM (\_ -> recvEvt c) [1..n]
                ; mapM_ (\_ -> recvEvt c) [1..n]
                ; let success = evalFormula input formula
                ; if success then alwaysEvt () else neverEvt })
     ; putStrLn "Satisfiable" }
```

The encoding works by creating a channel `c`, and spawning two threads running transactional events. One of these events sends n messages with value `True` over `c`, and the other sends n messages with value `False` over `c`. The main thread then reads n messages from `c`, saving the values in the list `input` and then reads and discards n more messages. The manner in which communications are matched determines the assignment of booleans to elements of the list `input`, and every assignment is possible. This encoding shows the following theorem.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Theorem 3 (*Communication matching is NP-hard*)
Given three synchronizing events $\mathcal{S} = \{\langle\theta_1, e_1\rangle, \langle\theta_2, e_2\rangle, \langle\theta_3, e_3\rangle\}$ such that e_1, e_2 and e_3 share a common channel and e_1, e_2 and e_3 do not use the `chooseEvt` combinator, the problem of finding a sequence of synchronous evaluation steps $\mathcal{S} \rightsquigarrow^* \{\langle\theta_1, \texttt{alwaysEvt}\ e'_1\rangle, \langle\theta_2, \texttt{alwaysEvt}\ e'_2\rangle, \langle\theta_3, \texttt{alwaysEvt}\ e'_3\rangle\}$ is NP-hard.

When we allow uses of `chooseEvt`, the problem becomes NP-hard even for just a single thread that makes no communications. In this case, we can encode boolean satisfiability with

```
sat n formula =
  do { sync (do { input <- mapM (\_ -> (alwaysEvt False)
                                       `chooseEvt`
                                       (alwaysEvt True)) [1..n]
                ; let success = evalFormula input formula
                ; if success then alwaysEvt () else neverEvt })
     ;  putStrLn "Satisfiable" }
```

This program uses `chooseEvt` to non-deterministically construct a list of booleans, and checks if the list corresponds to a satisfying assignment. This encoding implies the following, fairly unsurprising, theorem.

Theorem 4 (*Non-deterministic choice is NP-hard*)
Given a synchronizing event $\mathcal{S} = \{\langle\theta, e\rangle\}$ such that e uses no communication combinators, the problem of finding a sequence of synchronous evaluation steps $\mathcal{S} \rightsquigarrow^* \{\langle\theta, \texttt{alwaysEvt}\ e'\rangle\}$ is NP-hard.

These results show that the expressiveness of transactional events comes at a high cost; any implementation of transactional events must be able to solve NP-hard problems. On the other hand, not every event synchronization requires a search through a space of exponential size. While CML-like (i.e., single communication) transactional events can be synchronized efficiently, we cannot hope to have an implementation that efficiently synchronizes a large number of relatively long transactional events that communicate on a relatively large set of channels. Therefore, programmers ought to keep this in mind and, as much as possible, keep their transactions short and communicating on disjoint channels. Furthermore, many useful transactional-event idioms (e.g., guarded receive, triple-swap channels) seem to have these properties.

6 Implementation

In this section, we describe an implementation of transactional events.[10] 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 STM extensions of Concurrent Haskell (Harris *et al.*, 2005b) available in GHC (Glasgow Haskell Compiler, 2006).[11] Hence, the implementation

[10] The implementation may be obtained at `http://journals.cambridge.org/issue_Journalof functionalprogramming/Vol18No5-6`.

[11] Note, however, that the implementation of transactional events using STM is significantly more involved than the encoding of transactional shared memory using transactional events given in Section 5.3.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

required no changes to the Haskell compiler or run-time system. Furthermore, the implementation may take advantage of recent developments that have extended GHC to execute Haskell (with STM) on shared-memory multiprocessors (Harris *et al.*, 2005a). 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 of communicating threads. Hence, the synchronization of threads will not (unduly) impact the progress of non-synchronizing threads; similarly, the synchronization of one group of threads will not impact the progress towards the synchronization of another independent group of threads.

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 4. 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.

6.1 Unsuitable implementation strategies

Before describing our implementation, it is instructive to understand why some seemingly natural implementation strategies are unsuitable for transactional events. We have motivated the semantics of transactional events by suggesting that an event synchronization takes place in a transaction that can be aborted if it does not successfully complete; such a view aligns well with the idea of delaying the choice of threads and evaluation transitions in a synchronization. Hence, one might consider directly applying techniques for implementing transactional shared memory (Grossman, 2006) to the problem of implementing transactional events.

One approach to implementing transactional shared memory is to control the thread scheduler so that the atomicity and isolation of a transaction is ensured (Manson *et al.*, 2005; Ringenburg & Grossman, 2005). A block of code to be executed as a transaction runs without locking and accumulates a thread-local log that records every memory write. The log is used to revoke the effects of an uncompleted transaction that is interrupted by thread pre-emption; the transaction is retried when the thread is rescheduled. This approach makes the optimistic assumption that transactions will rarely be pre-empted, because transactions generally execute quickly.

Another popular approach to implementing transactional shared memory is to have each transaction compute using a private version of memory and then reflect changes back to shared memory using an optimistic synchronization protocol (Shavit & Touitou, 1997; Harris & Fraser, 2003; Harris *et al.*, 2005b). A block of code to be executed as a transaction runs without locking and accumulates a thread-local log that records every memory read and write. When the block completes, it attempts to validate the log. If the log is consistent with the current state of shared memory, the block (atomically) commits the logged changes to memory; if the log is inconsistent with the current state of shared memory (because another

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

thread has committed changes to memory read from or written to by this block), then the log is discarded and the transaction is re-executed. This approach makes the optimistic assumption that transactions will rarely have an inconsistent view of memory, because transactions generally access disjoint memory.

Yet another approach to implementing transactional shared memory is to have each transaction access shared memory using locks and to use a rollback mechanism to avoid deadlock (Adl-Tabatabai *et al.*, 2006; Harris *et al.*, 2006; Hindman & Grossman, 2006). A block of code to be executed as a transaction runs with locking for memory writes and accumulate a thread-local log that records the original value at every memory write. The log is used to revoke the effects of an aborted transaction. Locking for memory reads leads to eager conflict detection, while validating memory reads when the block completes leads to lazy conflict detection. As with the previous approach, this approach makes the optimistic assumption that transactions will rarely block trying to acquire a lock, because transactions generally access disjoint memory.

Under all of these approaches, one might imagine that a channel could be implemented as a mutable reference, so that channel sends and receives are logged (in an appropriate manner). A transactional event that sends and receives values from multiple channels would simply ensure that the reads and writes occur within the same transaction. Unfortunately, such an implementation does not suffice, since a (traditional) transaction does not encompass the execution of multiple threads, which is required to handle the *synchronous* nature of communication.

One might also imagine extending these approaches in the following manner. Upon executing a synchronization, a thread enters a transaction and accumulates a thread-local transaction log that records every memory read and write. When two synchronizing threads communicate via message passing, each of the threads must be in a transaction and the communication entails the 'merging' of their transaction logs; the two synchronizing threads now commit or abort together. Since a thread may communicate with more than one other thread during a transactional event synchronization, a single synchronization transaction log may be the result of 'merging' the transaction logs of many synchronizing threads; this single transaction log governs the entire set of synchronizing threads, so that they commit or abort together. A set of synchronizing threads commits if they all evaluate to `alwaysEvt` e' events with a consistent log, while they abort if one (or more) evaluate to a `neverEvt` event, an uncaught exception, or an unmatched communication event, or if the log is determined to be inconsistent.

While the implementation described below will share some passing resemblance to this scheme, using a single transaction log for a set of synchronizing threads does not satisfactorily suggest a means of handling the non-determinism in transactional event synchronization. Non-determinism arises from uses of `chooseEvt` and from uses of communication events, whereby a sender is non-deterministically matched with a receiver.

As was noted earlier, techniques for implementing transactional shared memory take advantage of optimistic assumptions about the nature of shared memory transactions to run without locking, eagerly reading from shared memory, and to

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

rarely re-execute transactions. One might be tempted to apply a similar optimistic assumption about the nature of transactional events. For example, one might resolve non-determinism in transactional event synchronization by randomly selecting an alternative at uses of `chooseEvt` and by eagerly matching senders with receivers.

Unfortunately, it is by no means clear that transactional events support this optimistic assumption. Consider, for example, the triple-swap channel and the `swapEvt` from Section 5.4. Recall that each thread that swaps on a channel non-deterministically chooses between acting as a client or as a leader in the exchange protocol. In order for three threads to synchronize with a `swapEvt` on the same triple-swap channel, exactly two threads must act as clients and exactly one thread must act as a leader. Hence, an implementation that resolves non-determinism at uses of `chooseEvt` by randomly choosing an alternative has only a 3/8 chance of finding a successful synchronization (without even considering possible matchings of senders and receivers). Things only get worse if more than three threads simultaneously attempt to swap on the same channel. Furthermore, an implementation must determine when to abort a synchronization due to an unmatched communication and when to allow the synchronization to remain active waiting for a matching communication.

All of these considerations suggest that we investigate an implementation that is more systematic in the exploration of `chooseEvt` alternatives and matchings of senders and receivers. While an exhaustive search may be necessary in some instances, we strive to avoid both redundant configurations and configurations that may never successfully synchronize. In particular, we formulate a necessary and sufficient condition for a search configuration to correspond to a successful synchronization; this criterion may be used to prune the search of configurations that may never successfully synchronize.

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

6.2 Refined semantics

In our refined semantics, we distinguish between the concurrent threads of Section 4.2.3 and two additional kinds of threads, namely suspended threads and search threads:

$$\begin{array}{lrcl} \textit{Concurrent Threads} & T & ::= & \langle \theta, e \rangle \\ \textit{Suspended and Search Threads} & S & ::= & \langle \theta, M^{IO}, e \rangle \mid \langle \theta, e, \rho \rangle \\ \textit{Thread Soups} & \mathcal{P} & ::= & \{T, \ldots, S, \ldots\} \end{array}$$

Concurrent threads continue to execute according to the rules given in Figure 4, except that we will shortly revise the IOSync rule. A suspended thread represents a concurrent thread waiting for the result of synchronizing on a transactional event. A suspended thread includes the thread identifier, the `IO` evaluation context and the transactional event of the original concurrent thread. A search thread represents

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

incremental progress towards synchronizing on a transactional event. A search thread includes the thread identifier of the thread on whose behalf it is searching for a synchronization, a transactional event to be evaluated and a path recording the non-deterministic actions of the search thread. A completed search thread is one where the transactional event has the form `alwaysEvt` e'. Note that, within a thread soup, concurrent and suspended threads have unique thread identifiers, but multiple search threads may share the same thread identifier and each search thread will share the same thread identifier with a suspended thread.

A path records the non-deterministic actions made during the evaluation of a transactional event; in particular, it records the alternative taken at `chooseEvt` and the communication partner at `sendEvt` and `recvEvt`. A trail, written $\langle \theta, \rho \rangle$, simply pairs a thread identifier with a path:

$$\begin{array}{llll} \textit{Path Element} & \eta & ::= & \mathsf{Left} \mid \mathsf{Right} \mid \mathsf{Send}(\langle \theta_r, \rho_r \rangle) \mid \mathsf{Recv}(\langle \theta_s, \rho_s \rangle) \\ \textit{Path} & \rho & ::= & \bullet \mid \eta{:}\rho \end{array}$$

The path element Left indicates that the first alternative in a `chooseEvt` was taken, while the path element Right indicates that the second alternative was taken. The path element $\mathsf{Send}(\langle \theta_r, \rho_r \rangle)$ indicates that a synchronous message was sent to a search thread with thread identifier θ_r and path ρ_r (θ_r is the receiver). Similarly, the path element $\mathsf{Recv}(\langle \theta_s, \rho_s \rangle)$ indicates that a synchronous message was received from a search thread with thread identifier θ_s and path ρ_s (θ_s is the sender). (Note that neither the channel nor the message is included in the path element.)

There is a natural partial order on paths.

Definition 1 (*Extends*)
The path ρ_a *extends* the path ρ_b, written $\rho_a \succeq \rho_b$, if ρ_a is an extension of ρ_b (alternatively, if ρ_b is a suffix of ρ_a). Formally,

$$\rho_a \succeq \rho_b \equiv \exists \eta_1. \cdots \exists \eta_n. \rho_a = \eta_n{:}\cdots{:}\eta_1{:}\rho_b$$

Since the path elements $\mathsf{Send}(\langle \theta_r, \rho_r \rangle)$ and $\mathsf{Recv}(\langle \theta_s, \rho_s \rangle)$ include paths, a single trail $\langle \theta, \rho \rangle$ may be seen to represent not only the non-deterministic actions of one search thread, but also the non-deterministic actions of all search threads that it communicated with, either directly or indirectly. Thus, a single trail implies a set of trails upon which it depends.

Definition 2 (*Dependencies*)
The *dependencies* of a trail $\langle \theta, \rho \rangle$, written $\mathsf{Dep}(\langle \theta, \rho \rangle)$, is the set of trails implied by the trail. Formally,

$$\begin{array}{rcl} \mathsf{Dep}(\langle \theta, \bullet \rangle) & = & \langle \theta, \bullet \rangle \\ \mathsf{Dep}(\langle \theta, \mathsf{Left}{:}\rho \rangle) & = & \langle \theta, \mathsf{Left}{:}\rho \rangle \cup \mathsf{Dep}(\langle \theta, \rho \rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Right}{:}\rho \rangle) & = & \langle \theta, \mathsf{Right}{:}\rho \rangle \cup \mathsf{Dep}(\langle \theta, \rho \rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Send}(\langle \theta_r, \rho_r \rangle){:}\rho \rangle) & = & \langle \theta, \mathsf{Send}(\langle \theta_r, \rho_r \rangle){:}\rho \rangle \cup \{\langle \theta_r, \mathsf{Recv}(\langle \theta, \rho \rangle){:}\rho_r \rangle\} \\ & & \quad \cup \mathsf{Dep}(\langle \theta, \rho \rangle) \cup \mathsf{Dep}(\langle \theta_r, \rho_r \rangle) \\ \mathsf{Dep}(\langle \theta, \mathsf{Recv}(\langle \theta_s, \rho_s \rangle){:}\rho \rangle) & = & \langle \theta, \mathsf{Recv}(\langle \theta_s, \rho_s \rangle){:}\rho \rangle \cup \{\langle \theta_s, \mathsf{Send}(\langle \theta, \rho \rangle){:}\rho_s \rangle\} \\ & & \quad \cup \mathsf{Dep}(\langle \theta, \rho \rangle) \cup \mathsf{Dep}(\langle \theta_s, \rho_s \rangle) \end{array}$$

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

The dependencies at a path element denoting a communication adds a trail in which the communication partner's path includes the matching communication. That is, a search thread θ with path $\mathsf{Send}(\langle\theta_r, \rho_r\rangle){:}\rho$ indicates that a synchronous message was sent to a search thread with thread identifier θ_r and path ρ_r. Since the communication is synchronous, then it must be the case that a search thread θ_r with path ρ_r received a synchronous message (from the search thread θ with the path ρ); hence, there must be a search thread θ_r with the path $\mathsf{Recv}(\langle\theta, \rho\rangle){:}\rho_r$.

Note that the dependencies of a trail may include multiple trails with the same thread identifier. Such dependencies may arise when one synchronizing thread communicates multiple times with another synchronizing thread, or when one synchronizing thread communicates with two synchronizing threads which have themselves communicated with each other. Since the dependencies of a trail represent a view of the history of a set of search threads, it is reasonable to be interested in trails that have a consistent view of history.

Definition 3 (*Consistent*)
The trail $\langle\theta, \rho\rangle$ is *consistent* if no thread identifier in the dependencies of $\langle\theta, \rho\rangle$ is paired with incomparable paths. Formally,

$$\begin{aligned}\mathsf{Consistent}(\langle\theta, \rho\rangle) \equiv \forall\langle\theta_1, \rho_1\rangle \in \mathsf{Dep}(\langle\theta, \rho\rangle).\, \forall\langle\theta_2, \rho_2\rangle \in \mathsf{Dep}(\langle\theta, \rho\rangle).\\ \theta_1 = \theta_2 \Rightarrow (\rho_1 \succeq \rho_2 \vee \rho_2 \succeq \rho_1)\end{aligned}$$

An inconsistent trail has a view of history where the same search thread was required to have performed different sequences of non-deterministic actions. Note that the definition of a consistent trail is equivalent to the following:

$$\begin{aligned}\mathsf{Consistent}(\langle\theta, \rho\rangle) \Leftrightarrow \forall\langle\theta_1, \_\rangle \in \mathsf{Dep}(\langle\theta, \rho\rangle).\, \exists\rho_1.\\ \langle\theta_1, \rho_1\rangle \in \mathsf{Dep}(\langle\theta, \rho\rangle)\\ \wedge\, \forall\langle\theta_2, \rho_2\rangle \in \mathsf{Dep}(\langle\theta, \rho\rangle).\, \theta_1 = \theta_2 \Rightarrow \rho_1 \succeq \rho_2\end{aligned}$$

which asserts that for every thead identifier θ_1 in the dependencies of a trail there exists a path ρ_1 that extends every path of θ_1. The path ρ_1 is maximal for θ_1, in the sense that ρ_1 extends every path of θ_1 in the dependencies. This formulation of consistency means that the dependencies of a consistent trail may be efficiently represented by a finite map from thread identifiers to their maximal paths.

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 trails.

Definition 4 (*Committable*)
A set of trails $\{\langle\theta_1, \rho_1\rangle, \ldots, \langle\theta_k, \rho_k\rangle\}$ is *committable* if each θ_i is unique and all dependencies of each trail are satisfied by the set. Formally,

$$\begin{aligned}\mathsf{Committable}(\{\langle\theta_1, \rho_1\rangle, \ldots, \langle\theta_k, \rho_k\rangle\}) \equiv\\ \forall i \in \{1, \ldots, k\}.\, \forall j \in \{1, \ldots, k\}.\, i \neq j \Rightarrow \theta_i \neq \theta_j\\ \wedge\, \forall i \in \{1, \ldots, k\}.\, \forall\langle\theta, \rho\rangle \in \mathsf{Dep}(\langle\theta_i, \rho_i\rangle).\\ \exists j \in \{1, \ldots, k\}.\, \theta_j = \theta \wedge \rho_j \succeq \rho\end{aligned}$$

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Note that a necessary, but not sufficient, condition for a set of trails to be committable is for each trail to be consistent:

Lemma 5
If a set of trails $\{\langle\theta_1, \rho_1\rangle, \ldots, \langle\theta_k, \rho_k\rangle\}$ is *committable*,
then each trail $\langle\theta_i, \rho_i\rangle$ is *consistent*.

Hence, we may avoid configurations that may never evolve to a committable set of search threads by never generating a search thread with an inconsistent trail.

Figure 9 revises the dynamic semantics of Section 4 to evaluate suspended and search threads. The single IOSync rule has been replaced by the SyncInit and SyncCommit rules, while the synchronous evaluation rules dealing with event expressions have been replaced by rules dealing with search threads.

The SyncInit rule transitions a concurrent thread at a `sync` e to a suspended thread and a search thread. The suspended thread records the `IO` evaluation context of the concurrent thread and the transactional event. The search thread is initialized with the transactional event and an empty path.

The SyncCommit rule transitions a set of committable suspended and 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 $\mathcal{P}$.) The concurrent threads are formed out of the `IO` evaluation context from the suspended threads and the synchronization results from the search threads.

The remaining rules deal with the evaluation of search threads, making incremental progress towards synchronizing on a transactional event. There are straightforward adaptations of the rules for sequential evaluation of pure terms in a search thread (EvtEval), propagation of `alwaysEvt` and `throwEvt` (EvtThenAlways, EvtThenThrow, EvtCatchAlways, EvtCatchThrow), channel allocation (EvtNewSChan) and thread identity (EvtMyThreadId). The EvtChoose rule transitions a single search thread evaluating a `chooseEvt` to two search threads evaluating each of the choice alternatives; note that the paths of the search threads are extended to record the non-deterministic choice. The EvtNever rule terminates a search thread attempting to synchronize on a `neverEvt`. Similarly, the EvtThrow rule terminates a search thread that evaluates to an uncaught exception.

The EvtSendRecv 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.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

[IOEval, IOBindUnit, IOBindThrow, IOCatchUnit, IOCatchThrow, IOGetChar, IOPutChar, IOFork, and IOMyThreadId as in Figure 4, replacing the thread soup $\mathcal{T}$ with the thread soup $\mathcal{P}$.]

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

SyncCommit
$$\frac{\mathsf{Committable}(\{\langle \theta_1, \rho_1\rangle, \ldots, \langle \theta_k, \rho_k\rangle\})}{\mathcal{P} \uplus \{\langle \theta_1, M_1^{IO}, e_1\rangle, \langle \theta_1, \texttt{alwaysEvt}\ e_1', \rho_1\rangle, \ldots, \langle \theta_k, M_k^{IO}, e_k\rangle, \langle \theta_k, \texttt{alwaysEvt}\ e_k', \rho_k\rangle\} \xrightarrow{\epsilon} \mathcal{P}\backslash_{\{\theta_1,\ldots,\theta_k\}} \uplus \{\langle \theta_1, M_1^{IO}[\texttt{unitIO}\ e_1']\rangle, \ldots, \langle \theta_k, M_k^{IO}[\texttt{unitIO}\ e_k']\rangle\}}$$

EvtEval
$$\frac{e \hookrightarrow e'}{\mathcal{P} \uplus \{\langle \theta, M^{Evt}[e], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[e'], \rho\rangle\}}$$

EvtThenAlways
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{thenEvt (alwaysEvt}\ e')\ e_f], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[e_f\ e'], \rho\rangle\}$$

EvtThenThrow
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{thenEvt (throwEvt}\ e')\ e_f], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{throwEvt}\ e'], \rho\rangle\}$$

EvtNever
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{neverEvt}], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P}$$

EvtChoose
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{chooseEvt}\ e_l\ e_r], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[e_l], \mathsf{Left}{:}\rho\rangle, \langle \theta, M^{Evt}[e_r], \mathsf{Right}{:}\rho\rangle\}$$

EvtThrow
$$\mathcal{P} \uplus \{\langle \theta, \texttt{throwEvt}\ e', \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P}$$

EvtCatchAlways
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{catchEvt (alwaysEvt}\ e')\ e_h], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt}\ e'], \rho\rangle\}$$

EvtCatchThrow
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{catchEvt (throwEvt}\ e')\ e_h], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[e_h\ e'], \rho\rangle\}$$

EvtNewSChan
$$\frac{\kappa' \text{ fresh}}{\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{newSChan}], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt}\ \kappa'], \rho\rangle\}}$$

EvtSendRecv
$$\frac{\mathsf{Coherent}(\langle \theta_s, \rho_s\rangle, \langle \theta_r, \rho_r\rangle)}{\begin{array}{l}\mathcal{P} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{sendEvt}\ \kappa\ e], \rho_s\rangle, \langle \theta_r, M_r^{Evt}[\texttt{recvEvt}\ \kappa], \rho_r\rangle\} \\ \quad \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{sendEvt}\ \kappa\ e], \rho_s\rangle, \langle \theta_r, M_r^{Evt}[\texttt{recvEvt}\ \kappa], \rho_r\rangle, \\ \qquad \langle \theta_s, M_s^{Evt}[\texttt{alwaysEvt ()}], \mathsf{Send}(\langle \theta_r, \rho_r\rangle){:}\rho_s\rangle, \\ \qquad \langle \theta_r, M_r^{Evt}[\texttt{alwaysEvt}\ e], \mathsf{Recv}(\langle \theta_s, \rho_s\rangle){:}\rho_r\rangle\}\end{array}}$$

EvtMyThreadId
$$\mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{myThreadIdEvt}], \rho\rangle\} \xrightarrow{\epsilon} \mathcal{P} \uplus \{\langle \theta, M^{Evt}[\texttt{alwaysEvt}\ \theta], \rho\rangle\}$$

Fig. 9. TE Haskell: Dynamic semantics – Refined.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Definition 5 (*Coherent*)
The trails $\langle \theta_s, \rho_s \rangle$ and $\langle \theta_r, \rho_r \rangle$ are *coherent* if the trails are an acceptable sender/receiver pair. Formally,

$$\begin{aligned}
&\mathsf{Coherent}(\langle \theta_s, \rho_s \rangle, \langle \theta_r, \rho_r \rangle) \equiv \\
&\quad \theta_s \neq \theta_r \\
&\quad \wedge \forall \langle \theta, \rho \rangle \in \mathsf{Dep}(\langle \theta_r, \rho_r \rangle).\, \theta_s = \theta \Rightarrow \rho_s \succeq \rho \\
&\quad \wedge \forall \langle \theta, \rho \rangle \in \mathsf{Dep}(\langle \theta_s, \rho_s \rangle).\, \theta_r = \theta \Rightarrow \rho_r \succeq \rho \\
&\quad \wedge \forall \langle \theta_1, \rho_1 \rangle \in \mathsf{Dep}(\langle \theta_s, \rho_s \rangle).\, \forall \langle \theta_2, \rho_2 \rangle \in \mathsf{Dep}(\langle \theta_r, \rho_r \rangle). \\
&\qquad \theta_1 = \theta_2 \Rightarrow (\rho_1 \succeq \rho_2 \vee \rho_2 \succeq \rho_1)
\end{aligned}$$

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

Although coherency is used in the dynamic semantics to limit communication, any two trails may be judged coherent or incoherent. Indeed, a necessary, but not sufficient, condition for a set of trails to be committable is for each pair of distinct trails to be coherent:

Lemma 6
If a set of trails $\{\langle \theta_1, \rho_1 \rangle, \ldots, \langle \theta_k, \rho_k \rangle\}$ is *committable*,
then each pair of distinct trails $\langle \theta_s, \rho_s \rangle$ and $\langle \theta_r, \rho_r \rangle$ is *coherent*.

Another important property of coherent communication is that it preserves the consistency and coherency of trails:

Lemma 7
If the trails $\langle \theta_s, \rho_s \rangle$ and $\langle \theta_r, \rho_r \rangle$ are *consistent* and *coherent*,
then the trails $\langle \theta_s, \mathsf{Send}(\langle \theta_r, \rho_r \rangle) {:} \rho_s \rangle$ and $\langle \theta_r, \mathsf{Recv}(\langle \theta_s, \rho_s \rangle) {:} \rho_r \rangle$ are *consistent* and *coherent*.

As an immediate corollary, we have the property that evaluation under the refined semantics preserves the consistency of search threads:

Lemma 8
Suppose $\mathcal{P} \xrightarrow{a} \mathcal{P}'$ according to the semantics of Figure 9.
If, for each search thread $\langle \theta, e, \rho \rangle \in \mathcal{P}$, the trail $\langle \theta, \rho \rangle$ is *consistent*,
then, for each search thread $\langle \theta', e', \rho' \rangle \in \mathcal{P}'$, the trail $\langle \theta', \rho' \rangle$ is *consistent*.

The fact that the refined semantics never gives rise to inconsistent search threads shows that the semantics does not explore configurations that may never evolve to a committable set of search threads.

Figure 10 shows a small number of search threads that arise during the evaluation of three concurrent threads exchanging values on a triple-swap channel from Section 5.4. Figure 10(a) shows the initial configuration, where each search thread

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

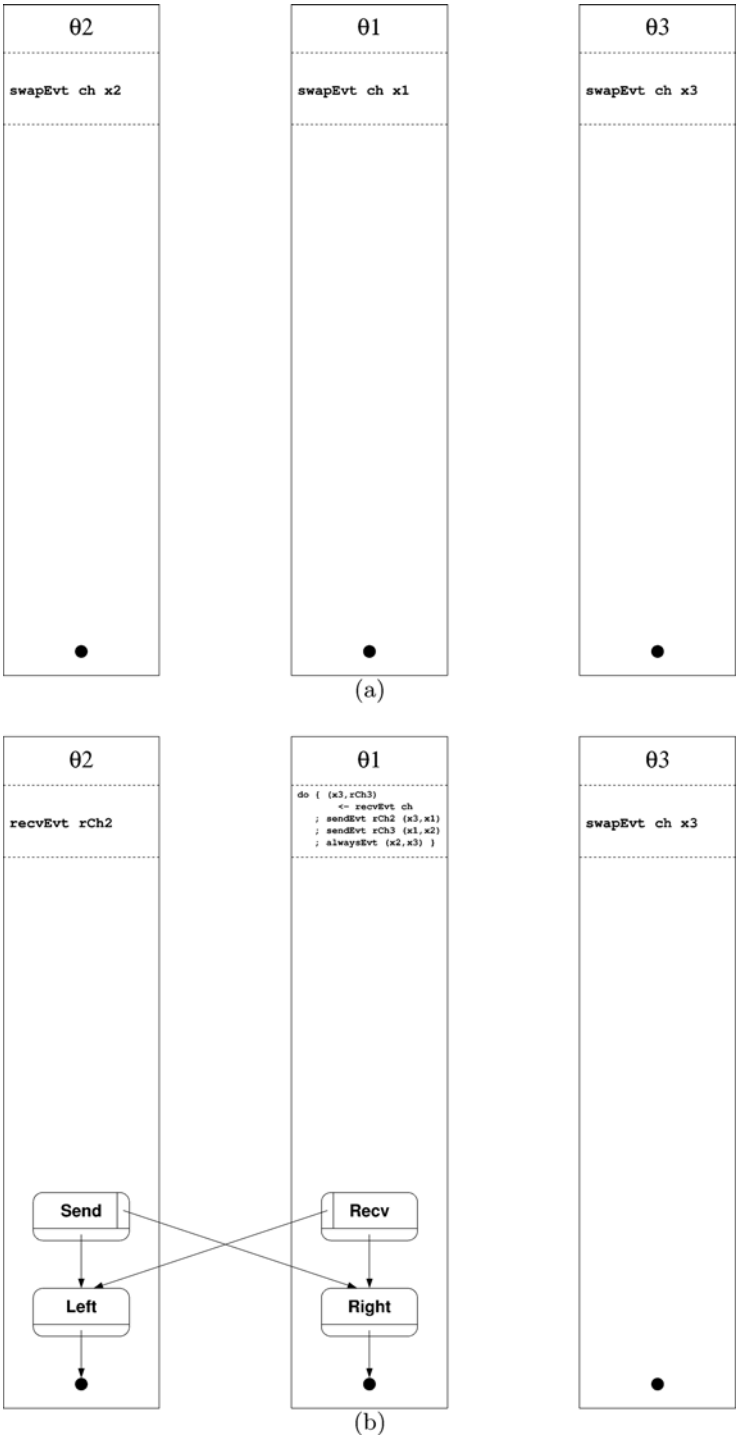

Fig. 10. See caption on next page.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

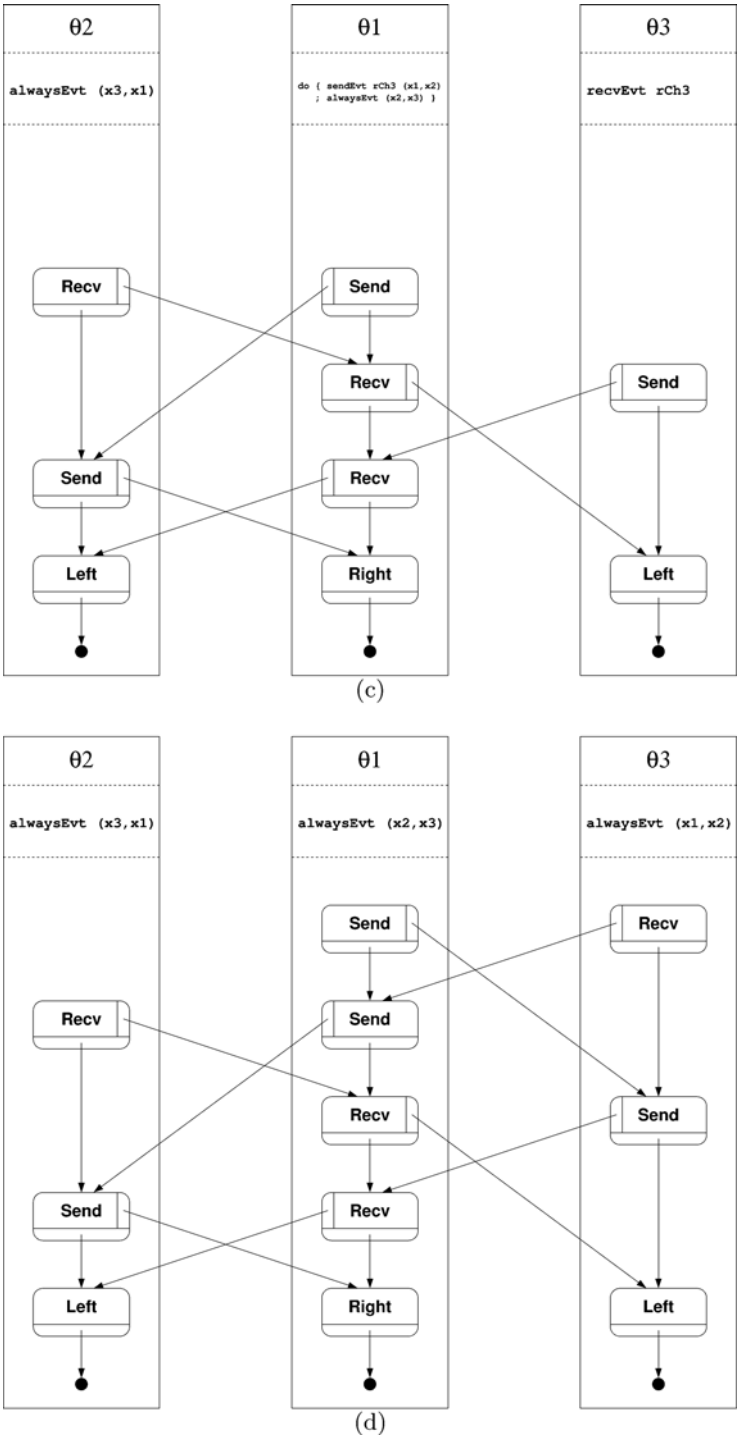

Fig. 10. Search threads for a committable triple-swap synchronization.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

is evaluating a transactional event of the form `swapEvt ch` $\mathtt{x}_i$ with an empty path. Figure 10(b) shows a subsequent configuration, where a search thread for thread θ_1 has non-deterministically chosen to act as a leader and a search thread for thread θ_2 has non-deterministically chosen to act as a client and a send of a reply channel by the second search thread has been non-deterministically matched with a receive of a reply channel by the first search thread. Figure 10(d) shows a final configuration, where each search thread has evaluated to a transactional event of the form `alwaysEvt (`$\mathtt{x}_{i+1 \bmod 3}$`,`$\mathtt{x}_{i+2 \bmod 3}$`)` with a consistent path; furthermore, the set of search threads forms a committable set. Note that the paths reachable from any search thread are included in the committable set.

6.2.1 Equivalence of the original and refined semantics

Finally, we wish to state that the refined semantics given in this section is equivalent to the semantics given in Section 4, in the sense that programs evaluated under the two semantics have the same observable behavior.

In one direction, we require that every observable behavior admitted by the semantics of Section 4 is also admitted by the semantics of this section:

Theorem 9
If $\mathcal{T} \xrightarrow{a_1;\cdots;a_n}{}^{*} \mathcal{T}'$ according to the semantics of Figures 3 and 4,
then $\mathcal{T} \xrightarrow{a'_1;\cdots;a'_m}{}^{*} \mathcal{T}'$ according to the semantics of Figure 9,
where $a_1;\cdots;a_n$ equals $a'_1;\cdots;a'_m$ modulo the insertion of ϵ actions.

In the other direction, we require that every behavior admitted by the semantics of this section is also admitted by the semantics of Section 4. In order to state this fact precisely, we introduce a simple translation from the thread soups of this section to the thread soups of Section 4:

$$\begin{aligned} |\{\}| &= \{\} \\ |\mathcal{P} \uplus \{\langle \theta, e\rangle\}| &= |\mathcal{P}| \uplus \{\langle \theta, e\rangle\} \\ |\mathcal{P} \uplus \{\langle \theta, M^{IO}, e\rangle\}| &= |\mathcal{P}| \uplus \{\langle \theta, M^{IO}[\mathtt{sync}\ e]\rangle\} \\ |\mathcal{P} \uplus \{\langle \theta, e, \rho\rangle\}| &= |\mathcal{P}| \end{aligned}$$

Note that this translation simply erases any search thread and translates a suspended thread to a synchronizing concurrent thread.

Theorem 10
If $\mathcal{T} \xrightarrow{a_1;\cdots;a_n}{}^{*} \mathcal{P}'$ according to the semantics of Figure 9,
then $\mathcal{T} \xrightarrow{a'_1;\cdots;a'_m}{}^{*} |\mathcal{P}'|$ according to the semantics of Figures 3 and 4,
where $a_1;\cdots;a_n$ equals $a'_1;\cdots;a'_m$ modulo the deletion of ϵ actions.

The non-determinism in both the original semantics and the refined semantics means that these theorems assert the *existence*, but not *uniqueness*, of a sequence of transitions in the corresponding semantics. Proofs of both theorems require supporting lemmas that relate the synchronous evaluation relation of Section 4.2.2 and the evaluation of committable search threads.[12]

[12] A technical appendix with a detailed proof of the equivalence of the original and refined semantics may be obtained at `http://journals.cambridge.org/issue_Journaloffunctionalprogramming/Vol18No5-6`.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

6.2.2 Discussion

It should be clear that the refined semantics given in this section has removed a major impediment to implementing TE Haskell. By interleaving the evaluation of many search threads with the evaluation of concurrent threads, an implementation based on the refined semantics does not need to omniciently choose a set of synchronizing threads and a sequence of synchronous evaluation transitions, which may be arbitrarily long. Although there remain a number of impediments to an efficient implementation, before we address them, it is worth noting that the refined semantics may be seen as starting point for a wide variety of implementations.

To see this, we must recognize three important aspects of the refined semantics, all related to the nature of search threads. First, the collection of search threads in the thread soup of a program state represents a frontier in the exploration of the space of possible synchronous evaluations. Second, the refined semantics places no restrictions on the strategy used to choose search threads for evaluation; different strategies correspond to different methods of extending the frontier represented by search threads. Finally, the refined semantics places few restrictions on how search threads are represented in an implementation; in particular, search threads need not be represented by threads at all (although they are in our implementation). Different design choices made regarding these aspects of the refined semantics will impact the practical behavior (time and space overhead, completeness of the search for synchronizations, etc.) of a TE Haskell program.

Perhaps the most obvious design choice (and the one adopted by our implementation) is to represent each search thread by a Concurrent Haskell thread (which maintains the search thread's identifier, event and path as part of the concurrent thread's evaluation); this design choice implicitly adopts the (preemptive and fair) concurrent thread scheduler as the strategy used to choose search threads for evaluation. The EvtChoose and EvtSendRecv transition rules correspond to thread creation; the EvtNever and EvtThrow transition rules correspond to thread termination; the remaining Evt transition rules correspond to sequential thread evaluation. The preemptive and fair nature of the concurrent thread scheduler ensures that the frontier in the space of possible synchronous evaluations represented by search threads will be extended on all fronts, yielding a complete exploration of the space that will find a synchronization if one exists. However, the nondeterministic nature of the concurrent thread scheduler means that the frontier will be extended in an unpredictable manner. Furthermore, a large number of choice and communication events may result in a long frontier made up of a large number of threads. Although Concurrent Haskell threads are lightweight, a significant number of search threads represented by concurrent threads could place a non-trivial burden on the run-time system; in particular, the space overhead of these threads could lead to more garbage collections and the time spent evaluating these search threads could leave less computational resources for non-synchronizing concurrent threads.

Another design choice is to represent each search thread as a simple data structure (which contains the search thread's identifier, event and path) and maintain a global set of these search thread structures; a single Concurrent Haskell thread

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

is dedicated to evaluating search threads, by removing a search thread from the set, taking an evaluation step appropriate for the search thread and inserting some number of search threads into the set. EvtChoose and EvtSendRecv transition rules correspond to inserting multiple new search threads; the EvtNever and EvtThrow transition rules correspond to inserting no search threads; the remaining Evt transitions rules correspond to inserting one updated search thread. Implementing the set of search threads using a LIFO stack roughly corresponds to a depth-first search through the space of possible synchronous evaluations, while using a FIFO queue roughly corresponds to a breadth-first search. The existence of infinite sequences of synchronous evaluation steps (arising from recursive events[13] and divergent sequential terms) makes the LIFO stack implementation an incomplete search; on the other hand, the FIFO queue implementation yields a complete exploration of the space that will find a synchronization if one exists. Representing a search thread as a simple data structure is likely to be more space efficient than representing a search thread by a Concurrent Haskell thread; similarly, the time required to construct a new data structure is likely to be significantly less than the time required to fork a new thread. Nonetheless, a large number of choice and communication events from independent synchronizations may result in a large set of search threads; dedicating a single Concurrent Haskell thread to the evaluation of search threads means that a 'short' synchronization by one group of theads may be slowed down by a 'long' synchronization by an independent group of threads.

An attractive design choice is to mix the above implementations: represent each search thread as a simple data structure, but maintain multiple sets of search threads; each Concurrent Haskell thread that synchronizes on a transactional event dedicates its computational resources to the evaluation of its own search threads. This design combines the space efficient representation of search threads with the concurrent evaluation of independent synchronizations. It also has the nice property that the computational resources dedicated to search for synchronizations is proportional to the number of concurrent threads blocked at synchronizations. While we have not yet had the opportunity to explore this design, we believe that it is likely to yield a well-balanced implementation.

Finally, all of these designs make use of the concepts introduced in the refined semantics (i.e., paths, dependencies, consistency, coherency and committability) to avoid exploring portions of the space of possible synchronizations that may never evolve to a committable set of search threads. Similarly, the Evt transition rules of the refined semantics serve to enumerate the space of possible synchronizations.

6.3 Representing channels and scanning for committable sets

While the refined semantics in Section 6.2 interleaves the evaluation of many search threads with the evaluation of concurrent threads, the EvtSendRecv and

[13] For example,

```
loopEvt x = ((alwaysEvt x) `thenEvt` (\ y -> loopEvt y))
            `chooseEvt` (alwaysEvt x)
```

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

SyncCommit and EvtSendRecv rules remain difficult to implement in a direct manner.

6.3.1 Representing channels

Recall the EvtSendRecv rule from Figure 9. A direct interpretation of this rule raises two issues. First, it requires matching two search 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 may be dealt with by representing a channel as a mutable reference containing two sets, one set of willing senders and one set of willing receivers:

$$\begin{array}{llll} \textit{Senders} & \textit{Sends} & ::= & \{\langle \theta, M^{Evt}, \rho, e\rangle, \ldots\} \\ \textit{Receivers} & \textit{Recvs} & ::= & \{\langle \theta, M^{Evt}, \rho\rangle, \ldots\} \\ \textit{Channel Store} & \mathbb{K} & ::= & \{\kappa \mapsto \langle \textit{Sends}, \textit{Recvs}\rangle, \ldots\} \end{array}$$

A channel name serves as an index into a channel store, where it maps to the pair of senders and receivers. A sender represents a search thread (with thread identifier θ and path ρ) willing to send the message e along a channel and continue executing according to M^{Evt}. Similarly, a receiver represents a search thread (with thread identifier θ and path ρ) willing to receive a message along a channel and continue executing according to M^{Evt}. The single EvtSendRecv rule may be replaced by separate EvtSend and EvtRecv rules[14]:

EvtSend

$$\frac{\mathbb{K}(\kappa) = \langle \textit{Sends}, \textit{Recvs}\rangle \qquad \mathcal{P}' = \bigcup \left\{ \begin{array}{l} \{\langle \theta_s, M_s^{Evt}[\texttt{alwaysEvt ()}], \mathsf{Send}(\langle \theta_r, \rho_r\rangle){:}\rho_s\rangle, \\ \langle \theta_r, M_r^{Evt}[\texttt{alwaysEvt } e], \mathsf{Recv}(\langle \theta_s, \rho_s\rangle){:}\rho_r\rangle\} \\ \quad \mid \langle \theta_r, M_r^{Evt}, \rho_r\rangle \in \textit{Recvs} \wedge \mathsf{Coherent}(\langle \theta_s, \rho_s\rangle, \langle \theta_r, \rho_r\rangle) \end{array} \right\}}{\begin{array}{l} \mathbb{K}; \mathcal{P} \uplus \{\langle \theta_s, M_s^{Evt}[\texttt{sendEvt } \kappa\ e], \rho_s\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K}[\kappa \mapsto \langle \textit{Sends} \cup \{\langle \theta_s, M_s^{Evt}, \rho_s, e\rangle\}, \textit{Recvs}\rangle]; \mathcal{P} \uplus \mathcal{P}' \end{array}}$$

EvtRecv

$$\frac{\mathbb{K}(\kappa) = \langle \textit{Sends}, \textit{Recvs}\rangle \qquad \mathcal{P}' = \bigcup \left\{ \begin{array}{l} \{\langle \theta_r, M_r^{Evt}[\texttt{alwaysEvt } e], \mathsf{Recv}(\langle \theta_s, \rho_s\rangle){:}\rho_r\rangle, \\ \langle \theta_s, M_s^{Evt}[\texttt{alwaysEvt ()}], \mathsf{Send}(\langle \theta_r, \rho_r\rangle){:}\rho_s\rangle\} \\ \quad \mid \langle \theta_s, M_s^{Evt}, \rho_s, e\rangle \in \textit{Sends} \wedge \mathsf{Coherent}(\langle \theta_s, \rho_s\rangle, \langle \theta_r, \rho_r\rangle) \end{array} \right\}}{\begin{array}{l} \mathbb{K}; \mathcal{P} \uplus \{\langle \theta_r, M^{Evt}[\texttt{recvEvt } \kappa], \rho_r\rangle\} \\ \quad \xrightarrow{\epsilon} \mathbb{K}[\kappa \mapsto \langle \textit{Sends}, \textit{Recvs} \cup \{\langle \theta_r, M_r^{Evt}, \rho_r\rangle\}\rangle]; \mathcal{P} \uplus \mathcal{P}' \end{array}}$$

Consider the EvtSend rule. A search thread that wishes to send a message along a channel spawns a pair of search threads for every coherent suspended receiver in the

[14] Most of the other rules may be simply extended with the channel store $\mathbb{K}$, which is otherwise ignored. The two exceptions are the EvtNewSChan rule, which must extend the channel store with a fresh channel mapped to empty sets of senders and receivers, and the SyncCommit rule, which must remove all senders and receivers corresponding to the synchronized threads from the sets in the channel store.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

channel store; the spawned search threads evaluate according to the event contexts of the sender and the receiver and have appropriately extended paths. Furthermore, the sender is added to the channel store. Note that by manipulating the channel store in this manner, the EvtSend rule guarantees that a sending search thread will send to all receivers already in the set of receivers and be in the set of senders for all future receivers.

6.3.2 Scanning for committable sets

Recall the SyncCommit rule from Figure 9. A direct interpretation of this rule raises a number of issues. First, it requires finding a committable set of completed search threads in the thread soup. Second, it requires finding the suspended threads in the thread soup that correspond to the completed search threads. Third, it requires removing from the thread soup all other search threads that were searching on behalf of the now synchronized concurrent threads.

We discuss the latter two issues in the next section. At the present time, we focus on a technique for finding a committable set of completed search threads in the thread soup.

Note that a search thread that transitions to a completed search thread may determine a minimal set of thread identifiers and paths required for commitment by consulting its dependencies. For example, in Figure 10(c), a search thread for thread θ_2 has transitioned to a completed search thread. Hence, from its dependencies, the completed search thread for thread θ_2 may determine that any committable set in which it participates must include some completed search thread for thread θ_1 with a path that extends

$$\begin{aligned}&\mathsf{Send}(\langle\theta_2, \mathsf{Send}(\langle\theta_1, \mathsf{Right}{:}\bullet\rangle){:}\mathsf{Left}{:}\bullet\rangle)\\ &{:}\mathsf{Recv}(\langle\theta_3, \mathsf{Left}{:}\bullet\rangle){:}\mathsf{Recv}(\langle\theta_2, \mathsf{Left}{:}\bullet\rangle){:}\mathsf{Right}{:}\bullet\end{aligned}$$

and some completed search thread for thread θ_3 with a path that extends

$$\mathsf{Send}(\langle\theta_1, \mathsf{Left}{:}\bullet\rangle){:}\mathsf{Left}{:}\bullet$$

However, as currently structured, the completed search thread for thread θ_2 may not determine if any such completed search threads exist.

We deal with this issue by introducing *completion references*, mutable references that maintain sets of completed search threads:

$$\begin{array}{lrcl}\textit{Completed Search Threads} & \mathcal{C} & = & \langle\theta, \texttt{alwaysEvt}\ e', \rho\rangle\\ \textit{Completion Store} & \mathbb{C} & ::= & \{\mathbb{c} \mapsto \{\mathcal{C}, \ldots\}, \ldots\}\end{array}$$

Two completion references are allocated at each communication and are stored in the Send and Recv path elements of the search threads that are spawned at a communication:

$$\begin{array}{lrcl}\textit{Path Element} & \eta & ::= & \mathsf{Left} \mid \mathsf{Right}\\ & & & \mid \mathsf{Send}(\langle\theta_r, \rho_r\rangle, \mathbb{c}_r, \mathbb{c}_s) \mid \mathsf{Recv}(\langle\theta_s, \rho_s\rangle, \mathbb{c}_s, \mathbb{c}_r)\end{array}$$

In the trail $\langle\theta_s, \mathsf{Send}(\langle\theta_r, \rho_r\rangle, \mathbb{c}_r, \mathbb{c}_s){:}\rho_s\rangle$, the completion reference $\mathbb{c}_r$ maintains sets of completed search threads that extend the path ρ_r of the receiver, and the completion

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

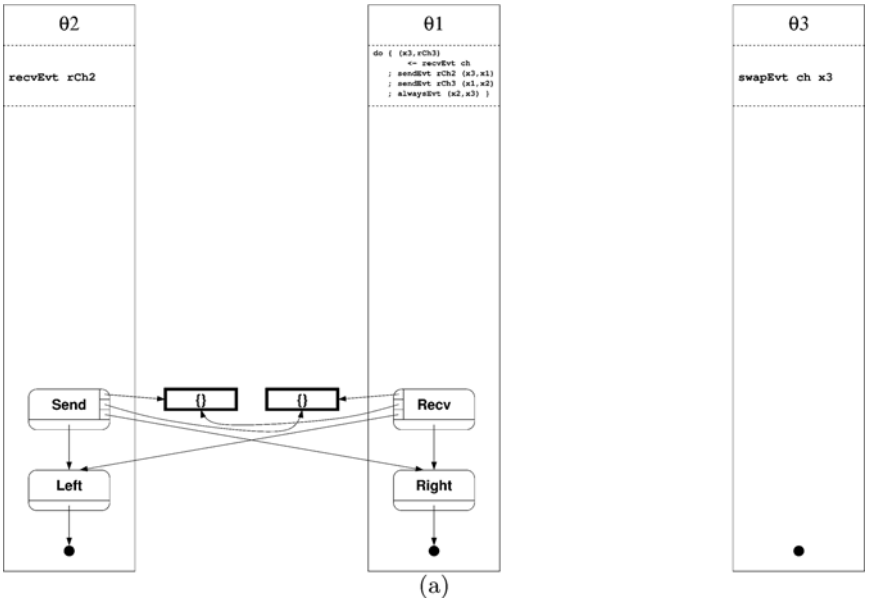

Fig. 11. See caption on next page.

reference $\mathbb{c}_s$ maintains sets of completed search threads that extend the path ρ of the sender. Note that the matching trail of the receiver, $\langle \theta_r, \mathsf{Recv}(\langle \theta_s, \rho_s \rangle, \mathbb{c}_s, \mathbb{c}_r) : \rho_r \rangle$, includes the *same* completion references, but in the reversed order.

When a search thread transitions to a completed search thread, it adds itself to the appropriate sets of completed search threads on its path, thereby making itself available for commitment to all of the search threads with which it has communicated. Finally, a completed search thread performs one scan of the sets of completed search threads of its communication partners, attempting to find a committable set of completed search threads. If such a set exists, then the search thread may commit the synchronization, on behalf of the entire committable set of completed search threads. If no such set exists, then the completed search thread halts and remains in the sets of completed search threads, awaiting commitment to be initiated by another completed search thread.

Figure 11 illustrates this strategy. Like Figure 10, Figure 11 shows a small number of search threads that arise during the evaluation of three concurrent threads exchanging values on a triple-swap channel from Section 5.4. Figure 11(a) replicates the configuration from Figure 10(b), where a send of a reply channel by the second search thread has been non-deterministically matched with a receive of a reply channel by the first search thread. Note that two completion references, initialized with the empty set $\{\}$, have been allocated for the communication; both references are reachable through the paths of both the sender and the receiver. Figure 11(b) shows a subsequent configuration, where additional communications have occurred and a search thread for thread θ_2 has completed. Note that the completed search thread has been added to the sets of completed search threads

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

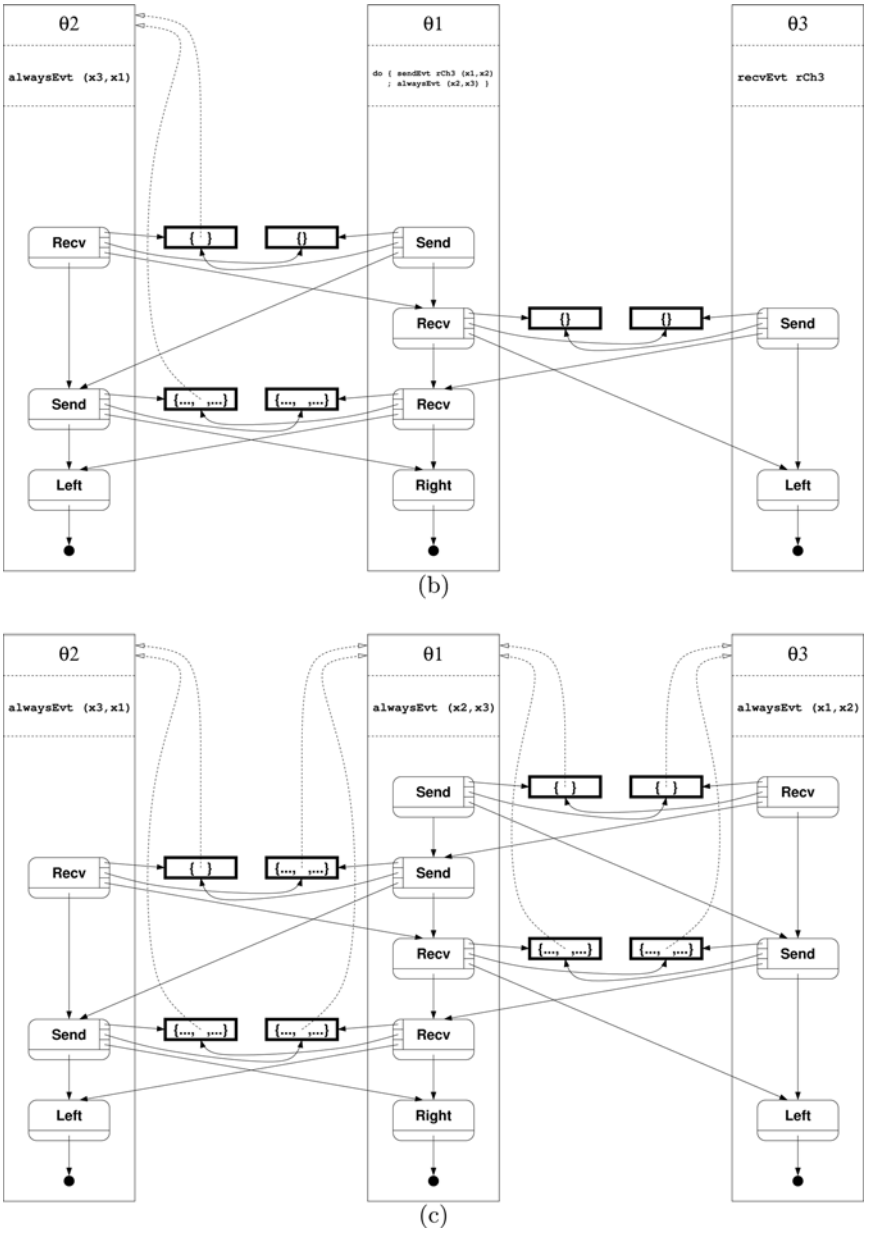

Fig. 11. Search threads with completion references.

on its path. However, the completed search thread may not reach a committable set of completed search threads, since no search threads for threads θ_1 and θ_3 have completed. Figure 11(c) shows a final configuration, where search threads for threads θ_1 and θ_3 have completed. By adding those completed search threads to the sets of completed search threads, now *any* completed search thread may reach a committable set of completed search threads.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Space precludes us from giving a complete formalization of this technique, but we remark on a few of the key details. An implementation that makes use of this strategy will require two functions, AddComplete and CommitScan. The first function updates the completion store by adding a completed search thread to the appropriate sets on its path. The second function uses the completion store to scan for committable sets of completed search threads. At any given time, there may be more than one committable set of completed search threads, so CommitScan returns a set of committable sets. (The next section will demonstrate how these functions are used in our implementation.) Finally, an important characteristic of this strategy is that any completed search thread needs only perform one scan for committable sets of completed search threads via CommitScan; if no such set exists at the time that a search thread transitions to a completed search thread, then it suffices for the completed search thread to halt and await commitment to be initiated by another search thread.

As a final note, it should be clear that both of the techniques discussed in this section (representing channels and scanning for committable sets) may be employed by each of the implementation designs discussed in Section 6.2.2.

6.4 Implementation details

We conclude this section with a number of additional details about our implementation of transactional events in Haskell.

At the beginning of this section, we noted that our implementation is written in Haskell, using the STM extensions of Concurrent Haskell available in GHC. As might be expected, the (global and mutable) channel and completion stores introduced in Section 6.3 are amenable to implementation via STM. Operations that manipulate the stores may be implemented as atomic sections that read and write transactional variables. A channel reference is implemented as a `(TVar [Sender a], TVar [Recver a])` (where `a` is the type of messages passed on the channel), and a completion reference is implemented as a `TVar [CompletedSearchThread]`.

We may also use transactional variables to address the remaining issues with the SyncCommit rule. Recall that the SyncCommit rule requires finding the suspended threads in the thread soup that correspond to the completed search threads and it requires removing from the thread soup all other search threads that were searching on behalf of the now synchronized concurrent threads. In our implementation, we ensure that all search threads arising from the same synchronization share a boolean reference implemented as a `TVar Bool`, and share a result reference implemented as a `TVar (Maybe a)` (where `a` is the type of the synchronization result). The boolean reference is allocated with the value `True` and the result reference is allocated with the value `Nothing` at the initialization of thread synchronization. When a set of search threads commit, all of their boolean references are set to `False` and all of their result references are set to `Just` e_i, where e_i is the synchronization result. Hence, our implementation does not require finding suspended search threads in the thread soup; rather, a blocked

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

concurrent thread resumes concurrent evaluation with the synchronization result when it discovers that the result reference has been set to `Just` e_i. Similarly, our implementation does not require removing search threads from the thread soup; rather, a search thread terminates (without evaluating to a completed search thread) when it discovers that its boolean reference has been set to `False`. As an optimization, a search thread also terminates if the boolean reference of one of its past communication partners has been set to `False` (since, once a communication partner has committed to a different synchronization, the search thread may never commit).

Additionally, a synchronization's boolean reference is used to periodically filter the lists of senders and receivers on a channel; doing so prevents space leaks and limits the number of potential partners that need to be considered at each communication. Similarly, a synchronization's boolean reference is used to determine when a completed search thread is available to participate in a committable set.

As was noted in Section 6.2.2, our implementation evaluates each search thread by a separate Concurrent Haskell thread. In Sections 6.2 and 6.3, we used explicit evaluation contexts (e.g., in the transition rules and in the sets of senders and receivers in the channel store). In the Haskell implementation, we use a CPS-like encoding of the `Evt` monad in order to fork new search threads during the evaluation of a transactional event, to propagate exceptions to an appropriate `catchEvt` handler, and to represent the context of senders and receivers.

The most interesting components of the implementation are the functions that initiate and commit an event synchronization. Initiation is handled by `sync`:

```
sync :: Evt a -> IO a
sync evt =
  do { tid <- myThreadId
     ; b <- atomically (newTVar True)
     ; r <- atomically (newTVar Nothing)
     ; forkIO (initSearchThread tid b r evt)
     ; atomically (do { mx <- readTVar r
                      ; case mx of
                          Nothing -> retry
                          Just x -> return x }) }
```

A concurrent thread that executes `sync evt` allocates two transactional variables (a boolean reference `b` and a result reference `r`) and then forks an initial search thread to evaluate the event. The concurrent thread then reads the contents of the result reference and retries until it sees a value other than `Nothing`; the implementation of STM in GHC ensures that the concurrent thread blocks until the result reference is written to by another thread, which signals that the synchronization has successfully completed.

Commitment of an event synchronization is handled by `finiCompletedSearch Thread`, which is invoked when a search thread transitions to a completed search thread.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

```
finiCompletedSearchThread :: CompletedSearchThread -> IO ()
finiCompletedSearchThread thrd =
  do { atomically (addComplete thrd)
     ; z <- atomically (do { zs <- commitScan thrd
                           ; case zs of
                               z:_ -> do { mapM_ setBoolRefInCompleted z
                                         ; return z }
                               _ -> return [] })
     ; atomically (mapM_ setResultRefInCompleted z)
     ; return () }
```

We use separate atomic sections to shorten the length of the transactions and to reduce the contention on transactional variables. The first atomic section adds the completed search thread to the appropriate sets on its path. The second atomic section performs a commit scan, and, upon finding a set of committable search threads, commits to the synchronization (by setting all of the boolean references in the completed search threads to `False`). The final atomic section sets all of the result references, unblocking the synchronizing threads.

Our implementation also takes advantage of the observation made in Section 6.2 that the dependencies of a consistent trail may be efficiently represented by a finite map from thread identifiers to trails with maximal paths. This improves the efficiency of the implementation (e.g., in the implementation of the Coherent predicate and the `commitScan` computation).

Although we have not written significant applications to benchmark our implementation, it has proven satisfactory for experimentation. Indeed, the implementation gracefully supports hundreds of threads simultaneously executing a triple-swap synchronization on the same triple-swap channel. As our implementation is written as a Haskell library, requiring no changes to the compiler or run-time system, we make few assumptions about the execution environment. Indeed, our only requirement is that the Concurrent Haskell thread scheduler is a preemptive and fair scheduler; this ensures that the implementation will (eventually) find a synchronization if one exists. Although we do require the STM extensions of Concurrent Haskell, we do not use all of the features provided by the STM extensions; in particular, we never make use of the `orElse` or `catchSTM` combinators. We do not employ any sophisticated garbage-collector features (e.g., weak pointers or finalizers), nor do we require specific garbage-collector options. We support, but do not require, execution on multiprocessors, thereby exploiting the high degree of parallelism in the evaluation of search threads.

For additional details, we encourage the interested reader to consult and experiment with the implementation.

7 Related work

The UniForM Workbench (Karlsen, 1997; Russell, 2001) 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

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

that of CML. Russell (2001) 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 CML 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 given above is closely related to the encoding of CML given in Section 5.2. The `computeEvent` operator is equivalent to the `guardCMLEvt` operator, providing pre-synchronous actions. The `>>>=` operator is equivalent to the `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 (1997) 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.

Jeffrey (1995a, b) has given a denotational semantics of CML using (variants) of the ideas from Moggi's computational monad program (Moggi, 1989, 1991). 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.

The STM extension of Concurrent Haskell (Harris *et al.*, 2005b), as well as the AtomCaml extension of OCaml (Ringenburg & Grossman, 2005), provide similar atomic transactions, but do not allow for synchronous message passing. A significant advantage of STM is that it has a more efficient implementation, based on the optimistic assumption that concurrently running transactions will seldom access the same memory. We showed in Section 5.3 that software transactions similar, but not identical, to those of STM Haskell can be encoded as transactional events.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

The stabilizers extension of CML (Ziarek *et al*., 2006) is perhaps the most closely related work. Stabilizers provide programmer-inserted checkpoints to which a concurrent program can roll back if a fault occurs. Each thread has its own checkpoints, and when a rollback occurs and synchronous communications are undone, both sides of the communication must roll back to consistent checkpoints. This is similar to the conceptual 'merging' of transactional event synchronizations of communicating threads in TE Haskell. The stabilizers implementation is designed primarily for correcting transient faults, and as such makes no attempt to systematically search for mutually consistent choices among threads as TE Haskell does.

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 CML. 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 directly 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 (Pike, 1989; Gasner & Reppy, 1993; Russell, 2001)). 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 in ways that compiler and run-time support may improve the efficiency of an implementation of TE Haskell. As discussed in Section 6, there are limits to what can be accomplished without modifying the compiler and/or run-time system. Exploring a native implementation of transactional events may yield new opportunities to optimize the implementation.

Finally, on the theoretical side, there are interesting questions about the relationship between transactional events and other concurrency calculi (e.g., CSP (Hoare, 1978), π-calculus (Sangiorgi & Walker, 2001)), about the right notions of behavioral equivalence, and about progress and fairness in an implementation.

Acknowledgments

We would like to thank Norman Ramsey for teaching the excellent course on Programming with Concurrency (Harvard University CS257, Fall 2005) that was the genesis of this work, as well as for helpful feedback on earlier versions. In addition we would like to thank Greg Morrisett, Avi Shinnar, Riccardo Pucella, John Reppy, the anonymous reviewers of ICFP'06 (one of whom pointed out the ease of encoding boolean satisfiability with transactional events), and the anonymous reviewers of JFP for their constructive and insightful comments.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

References

Adl-Tabatabai, Ali-Reza, Lewis, Brian T., Menon, Vijay, Murphy, Brian R., Saha, Bratin & Shpeisman, Tatiana. (2006) Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. New York: ACM Press, pp. 26–37.

Armstrong, Joe, Virding, Robert, Wikström, Claes & Williams, Mike. (1996) *Concurrent Programming in Erlang*, 2nd ed. Hertfordshire, UK: Prentice Hall International (UK) Ltd.

Donnelly, Kevin & Fluet, Matthew. (2006) Transactional events. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. New York: ACM Press, pp. 124–135.

Flatt, Matthew & Findler, Robert Bruce. (2004) Kill-safe synchronization abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*. New York: ACM Press, pp. 47–58.

Gasner, Emden R. & Reppy, John H. (1993) A multi-threaded high-order user interface toolkit. In *User Interface Software*, Bass, Len & Dewan, Prasun (eds). Software Trends, Vol. 1. New York: John Wiley & Sons, Chap. 4, pp. 61–80.

Glasgow Haskell Compiler (version 6.6). (2006) `http://www.haskell.org/ghc`.

Grossman, Dan. (April 2006) *Software Transactions are to Concurrency as Garbage Collection is to Memory Management*. Tech. Rept. 2006-04-01. University of Washington, Department of Computer Science & Engineering.

Harris, Tim & Fraser, Keir. (2003) Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'03)*. New York: ACM Press, pp. 388–402.

Harris, Tim, Marlow, Simon & Peyton Jones, Simon. (2005a) Haskell on a shared-memory multiprocessor. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. New York: ACM Press, pp. 49–61.

Harris, Tim, Marlow, Simon, Peyton Jones, Simon & Herlihy, Maurice. (2005b) Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. New York: ACM Press, pp. 48–60.

Harris, Tim, Plesko, Mark, Shinnar, Avraham & Tarditi, David. (2006) Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. New York: ACM Press, pp. 14–25.

Herlihy, Maurice & Moss, J. Eliot B. (1993) Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. New York: ACM Press, pp. 289–300.

Hindman, Benjamin & Grossman, Dan. (2006) Atomicity via source-to-source translation. In *Proceedings of the Workshop on Memory System Performance and Correctness (MSPC'06)*. New York: ACM Press, pp. 82–91.

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

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

Jeffrey, Alan. (1995a) A fully abstract semantics for a concurrent functional language with monadic types. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS'95)*. Los Alamitos, CA: IEEE Computer Society Press, pp. 255–265.

Jeffrey, Alan. (1995b) A fully abstract semantics for a nondeterministic functional language with monadic types. In *Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XI)*. Electronic Notes in Theoretical Computer Science, Vol. 1. New York: Elsevier.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Karlsen, Einar. (1997) The UniForM Concurrency ToolKit and its extensions to Concurrent Haskell. In *The Glasgow Functional Programming Workshop (GFPW)*.

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

Manson, Jeremy, Baker, Jason, Cunei, Antonio, Jagannathan, Suresh, Prochazka, Marek, Xin, Bin & Vitek, Jan. (2005) Preemptible atomic regions for real-time Java. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. Los Alamitos, CA: IEEE Computer Society, pp. 62–71.

Marlow, Simon, Peyton Jones, Simon, Moran, Andrew & Reppy, John. (2001) Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. New York: ACM Press, pp. 274–285.

Moggi, Eugino. (1989) Computational lambda calculus and monads. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS'89)*. Los Alamitos, CA: IEEE Computer Society Press, pp. 14–23.

Moggi, Eugino. (1991) Notions of computation and monads. *Info. Comput.* **93**(1), 55–92.

MonadPlus. (2005) `http://www.haskell.org/hawiki/MonadPlus`.

Moran, Andrew, Lassen, Soren B. & Peyton Jones, Simon. (1999) Imprecise exceptions, co-inductively. In *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*. Electronic Notes in Theoretical Computer Science, Vol. 26. New York: Elsevier, pp. 122–141.

Moss, J. Eliot B. (1985) *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA: MIT Press.

Panangaden, Prakash & Reppy, John. (1997) The essence of Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation, and Application*, Nielson, Flemming (ed). Monographs in Computer Science. New York: Springer-Verlag, Chap. 2, pp. 5–30.

Peyton Jones, Simon. (2001) Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, Hoare, Tony, Broy, Manfred & Steinbrüggen, Ralf (eds). NATO Science Series: Computer & Systems Sciences, Vol. 180. Amsterdam, The Netherlands: IOS Press, pp. 47–96.

Peyton Jones, Simon, Gordon, Andrew & Finne, Sigbjorn. (1996) Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. New York: ACM Press, pp. 295–308.

Peyton Jones, Simon, Reid, Alastair, Henderson, Fergus, Hoare, Tony & Marlow, Simon. (1999) A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99)*. New York: ACM Press, pp. 25–36.

Peyton Jones, Simon, & Wadler, Philip. (1993) Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*. New York: ACM Press, pp. 71–84.

Pike, Rob. (1989) A concurrent window system. *Comput. Syst.* **2**(2), 133–153.

Reppy, John. (1999) *Concurrent Programming in ML*. Cambridge, UK: Cambridge University Press.

Ringenburg, Michael F. & Grossman, Dan. (2005) AtomCaml: First-class atomicity via rollback. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*. New York: ACM Press, pp. 92–104.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press

Russell, George. (2001) Events in Haskell, and how to implement them. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. New York: ACM Press, pp. 157–168.

Sangiorgi, David & Walker, David. (2001) *The π-Calculus: A Theory of Mobile Processes*. Cambridge, UK: Cambridge University Press.

Shavit, Nir & Touitou, Dan. (1997) Software transactional memory. *Distribut. Comput.* **10**(2), 99–116.

Turbak, Franklyn. (1996) First-class synchronization barriers. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. New York: ACM Press, pp. 157–168.

Wadler, Philip. (1995) Monads for functional programming. In *Advanced Functional Programming*, Jeuring, Johan & Meijer, Erik (eds). Lecture Notes in Computer Science, Vol. 925. New York: Springer-Verlag.

Welc, Adam, Jagannathan, Suresh & Hosking, Antony L. (2004) Transactional monitors for concurrent objects. In *Proceedings of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*. Lecture Notes in Computer Science, Vol. 3086. New York: Springer-Verlag, pp. 519–542.

Ziarek, Lukasz, Schatz, Philip & Jagannathan, Suresh. (2006) Stabilizers: A modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. New York: ACM Press, pp. 136–147.

https://doi.org/10.1017/S0956796808006916 Published online by Cambridge University Press