Transactional Events for ML

Laura Effinger-Dean Matthew Kehrt Dan Grossman
University of Washington
{effinger, mkehrt, djg}@cs.washington.edu

Abstract

Transactional events (TE) are an approach to concurrent programming that enriches the first-class synchronous message-passing of Concurrent ML (CML) with a combinator that allows multiple messages to be passed as part of one all-or-nothing synchronization. Donnelly and Fluet (2006) designed and implemented TE as a Haskell library and demonstrated that it enables elegant solutions to programming patterns that are awkward or impossible in CML. However, both the definition and the implementation of TE relied fundamentally on the code in a synchronization not using mutable memory, an unreasonable assumption for mostly functional languages like ML where functional interfaces may have impure implementations.

We present a definition and implementation of TE that supports ML-style references and nested synchronizations, both of which were previously unnecessary due to Haskell's more restrictive type system. As in prior work, we have a high-level semantics that makes nondeterministic choices such that synchronizations succeed whenever possible and a low-level semantics that uses search to implement the high-level semantics soundly and completely. The key design trade-off in the semantics is to allow updates to mutable memory without requiring the implementation to consider all possible thread interleavings. Our solution uses first-class heaps and allows interleavings only when a message is sent or received. We have used Coq to prove the high- and low-level semantics equivalent.

We have implemented our approach by modifying the Objective Caml run-time system. By modifying the run-time system, rather than relying solely on a library, we can eliminate the potential for nonterminating computations within unsuccessful synchronizations to run forever.

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

General Terms Languages, Design

Keywords Transactional Events, Synchronous Message Passing, Concurrency

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

1. Introduction

Programming abstractions for concurrent programming are crucial for helping programmers manage the complexity inherent in supporting multiple threads of execution. Concurrent ML (CML) (Reppy 1999) is an elegant approach based on first-class synchronous message-passing. Programs build *events* that when *synchronized* on block until they can complete successfully. Events can send or receive messages on channels. Powerful event combinators, such as an event that chooses exactly one of two events to perform, let programmers build more sophisticated abstract communication protocols out of simpler ones. Section 2.1 briefly reviews CML.

By design, CML has a key limitation: Any synchronization performs at most one message send/receive. While convenient for efficient implementation, this limitation makes some communication protocols more difficult or even impossible to write. Recently, Donnelly and Fluet (2006) developed Transactional Events (TE) for Haskell, which removes this limitation. A "thenEvt" combinator lets programmers build events that contain multiple communications and that block until all the communications can succeed. Consider this example (in Caml syntax, which we use throughout):

```
thenEvt (recvEvt k1) (fun x ->
  if f x then sendEvt k2 x else neverEvt)
```

This event receives a value on `k1` and sends it on `k2`, but only if the function `f` "approves" the value. Otherwise, the entire event cannot succeed since `neverEvt` never succeeds and an event built from `thenEvt` succeeds only if both its arguments produce successful events. Therefore, *no value is received on `k1`.* Implementing `thenEvt` is nontrivial, requiring search to see if an entire synchronization can succeed while having no observable effect until the synchronization succeeds. Section 2.2 briefly reviews TE.

TE is a great step forward in providing rich combinators for synchronous message-passing and integrating them into a modern language implementation, but the definition and implementation both rely fundamentally on Haskell's purely functional nature.[1] For the example above, if we ask, "what might `f x` do?" the answer in Haskell is quite benign: Thanks to the monad to which the TE combinators belong, `f x` cannot mutate any memory references and cannot perform any message passing. It might not terminate, which in the TE implementation would hurt performance (it would consume resources forever) but would not violate the semantics.

In a mostly functional language like ML, we cannot assume `f x` is so well-behaved. Moreover, simplistic restrictions, like treating any access to mutable memory inside `f x` as a dynamic error, are fundamentally inappropriate: An essential aspect of ML is that code may use references at any time, often using abstraction to provide a functional interface. For example, if the function `f` used a memoization table internally, that should not be observable to callers. Having a memoizing `f` fail within events but a non-memoizing variant

[1] They do not rely on Haskell's laziness; in fact, eagerness slightly simplifies parts of the implementation.

succeed is a fundamental violation of ML-style abstraction. Similarly, `f` might perform a nested synchronization with additional message sends/receives, which is also impossible in the monadic setting. In short, if we cannot provide reasonable semantics and implementation for TE in the presence of side effects, then TE's utility will be restricted to purely functional languages.

This paper provides precisely such a semantics and implementation, focusing on effects from accessing mutable references. In extending TE to make sense in the presence of side effects and nested synchronizations, the most important issue is how to provide an appropriate guarantee that the implementation will find a successful communication if one exists. This completeness guarantee is essential to TE's usefulness, but extending it naïvely would require the implementation to consider all interleavings of side effects.

More specifically, if two threads are communicating and one performs `r:=1; r:=2` while the other performs `if !r = 1 then fail() else succeed()`, must TE find the successful interleaving? We decide no since doing so is intractably expensive and not useful for any reasonable idiom we have encountered. To avoid this problem while still providing a precise semantics and a completeness guarantee *with respect to communication patterns*, we define and implement a semantics in which each thread's updates to the shared-memory heap happen in "chunks" (much like transactional memory) at the point where message sends/receives occur. Section 3 provides a more complete informal description of our approach to mutation and several examples justifying the semantics.

As in the original TE work, we have two formal semantics, a high-level nondeterministic semantics for programmers in Section 4 and a low-level semantics in Section 5 that makes the search for a successful communication explicit. While the addition of a mutable heap and nested synchronizations is nontrivial, the overall structure of both semantics is reassuringly similar to prior work. To bolster our confidence in the requisite correctness proof—that the low-level semantics finds a communication if and only if the high-level semantics does (see Section 6)—we have formalized the semantics and proof in Coq, which has not been done previously.

The low-level semantics directly informs our implementation, which we achieved by modifying the Objective Caml bytecode interpreter and run-time system, as described in Section 7. Run-time support simplifies parts of the implementation, such as copying threads to perform search, and lets us ensure nontermination on unsuccessful communications is stopped in a timely fashion.

For further discussion of related work and conclusions, see Sections 8 and 9 respectively.

2. Background

2.1 Concurrent ML

Concurrent ML (Reppy 1999) is a synchronous message-passing system originally developed for Standard ML. Threads communicate by sending values over typed channels. Threads sending or receiving on a channel block until a complementary send or receive is performed by another thread on the same channel.

CML introduces *events*, which describe communications to be performed later. The simplest events are produced by the functions `sendEvt` and `recvEvt`, whose types are shown in Figure 1, along with most of the CML interface. These functions do not perform the send or receive; they produce events describing a send or receive.

To perform the communication that an event describes, a thread *synchronizes* on the event by calling `sync`, which has type `'a event -> 'a`. For send and receive events, this attempts the communication, blocking until a matching communication occurs. The synchronization *succeeds* if the communication is performed.

Once the separation between the definition and synchronization of events is articulated, the concept can be generalized with several

```
type 'a channel
type 'a event

val newChan : unit -> 'a channel

val sendEvt : 'a channel -> 'a -> unit event
val recvEvt : 'a channel -> 'a event
val alwaysEvt : 'a -> 'a event
val neverEvt : 'a event
val chooseEvt : 'a event -> 'a event -> 'a event
val wrapEvt : 'a event -> ('a -> 'b) -> 'b event
(* others *)

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

Figure 1. CML's interface.

other combinators that produce events. We will discuss just those used in this paper. One is `chooseEvt`, which produces an event from two other events.[2] When synchronized on, a `chooseEvt` synchronizes on both its subevents until exactly one succeeds. The value produced by the `chooseEvent` is the value produced by the chosen subevent.

As a simple example, the following event, when synchronized on, waits to receive on two different channels, `k1` and `k2`, and succeeds by receiving exactly one value.

```
let either = chooseEvt (recvEvt k1) (recvEvt k2)
```

Another, simpler event combinator is `alwaysEvt`, which takes a value and returns an event on which synchronization always succeeds immediately with that value. Conversely, synchronizing on a `neverEvt` will never succeed, i.e., it will block forever. `alwaysEvt` and `neverEvt` are often useful in conjunction with other event combinators.

The final event combinator we discuss is `wrapEvt`. It takes a single event and a function to apply to the result of synchronizing on that event. Synchronizing on a `wrapEvt` first synchronizes on the subevent, and, when it succeeds, applies the function to the result of synchronization to produce a value.

`wrapEvt` is often used to change the type of one subevent in a `chooseEvt`, such that both subevents have the same type. For example, the following event chooses between receiving on two channels. However, one channel returns floats and the other ints. We convert the int to a float with a `wrapEvt` before returning it.

```
let floatEvt =
  chooseEvt (wrapEvt (recvEvt intk) float_of_int)
            (recvEvt floatk)
```

The function passed to `wrapEvt` is called *after* the event passed to `wrapEvt` has synchronized. At that point, the event can no longer fail, even if the function does not terminate or raises an exception. Moreover, if the function synchronizes on a second event, the first event in a `wrapEvt` will already have synchronized even if the second synchronization fails. For example, the following code repeatedly sends on a channel. Each send happens individually, when a single receive is performed on the channel in another thread.

```
let rec sendForever k v =
  sync (wrapEvt (sendEvt k v)
          (fun () -> sendForever k v))
```

There are many examples of elegant multithreaded code using the CML event combinators. In particular, CML easily lends itself to writing server threads, which compute some value, possibly with

[2] This is just as powerful as a variant that chooses from a list of events.

```
let refServer init =
  let setk = newChan () in
  let getk = newChan () in
  let loop value =
    sync (chooseEvt
            (wrapEvt
              (sendEvt getk value)
              (fun () -> loop value))
            (wrapEvt
              (recvEvt setk)
              (fun value' -> loop value'))) in
  ignore (Thread.create loop init);
  (setk, getk)

let get (setk, getk) = sync (recvEvt getk)

let set (setk, getk) v = sync (sendEvt setk v)
```

Figure 2. A server to simulate a mutable reference in CML

```
val thenEvt :
  'a event -> ('a -> 'b event) -> 'b event
```

Figure 3. Type of `thenEvt`

input received on channels, and send the value on another channel. For example, Figure 2 presents a *refserver* that simulates a mutable reference. The function creates a new thread that synchronizes on an event that chooses between receiving a new value on a setter channel and sending the old value on a getter channel.

2.2 Transactional Events

Transactional events are an extension of CML motivated by the observation that CML does not provide a way of sequencing multiple communications in a single event. To address this limitation, TE provides a combinator, `thenEvt`, that allows such sequencing. Its type is shown in Figure 3.[3] Specifically, `thenEvt` takes an event and a function for producing a new event from the result of the first event. Synchronizing on a `thenEvt` synchronizes on the first event and then calls the function with the resulting value. This results in an event that is then synchronized on. We can use nested `thenEvt`s to sequence an arbitrary number of events. For the synchronization of `thenEvt` to succeed, the synchronization of both sequenced events must succeed. Such events are *transactional* in that they provide an all-or-nothing semantics: the events contained in them either all succeed or all fail.

The ability to combine a sequence of events into a single, transactional event makes synchronizing on events significantly more complex than in CML. In CML, synchronizing on an event performs at most a single send or receive. In TE, synchronizing on an event may perform an arbitrary number of communications, which may be with multiple threads. For such a synchronization to succeed, all the threads with which it communicates must also have successful synchronizations, which may themselves communicate with other synchronizing threads. Moreover, all these threads may use `chooseEvent` any number of times during their sequence of communications.

Adding `thenEvt` increases the expressive power of CML. A simple example of this is the guarded-receive idiom, which is common enough to enjoy explicit support in frameworks such as Erlang (Armstrong et al. 1996) and CSP (Hoare 1978). In guarded

[3] The addition of `thenEvt` to CML makes the type constructor `event` an instance of a monad with plus, which was useful in Haskell.

```
(* 'a chan -> ('a -> bool) -> 'a event *)
let guardedRecvEvt k pred =
  thenEvt
    (recvEvt k)
    (fun x ->
      if pred x then alwaysEvt x else neverEvt)
```

Figure 4. Guarded receive in TE

receive, a thread attempts to receive on a channel by synchronizing on an event. There is some predicate that guards the receive, such that the receive (and, as always, the matching send of another thread) succeeds only if the predicate holds on the received value.

Implementing guarded receive with CML is unexpectedly complex for such a relatively straightforward operation (Donnelly and Fluet 2006). In particular, a CML-style guarded receive is inherently non-modular, because the protocol requires the participation of both the sender and the receiver. If a thread needs to guard receives on a channel, *all* other threads that send or receive on that channel must follow the guarded-receive protocol as well.

A TE implementation of this idiom, presented in Figure 4, is simple. The `thenEvent` sequences together two events. The first is a receive. The received value is checked to see if it satisfies the guard. If so, the next event is an `alwaysEvt` that returns the received value. Otherwise the next event is a `neverEvt`, which cancels the transaction. The whole event, therefore, blocks until a value is received that satisfies the predicate. The event is modular: other threads do not have to use a complex protocol to communicate with this event.

The inclusion of `thenEvt` also allows protocols that cannot be encoded in CML at all. One example is n-way thread synchronization. An event synchronization in CML performs a single send or receive, which necessarily synchronizes only two threads. One cannot use CML's event combinators to build an event that synchronizes more than two threads. With `thenEvt`, however, multiple sends and receives can be performed by synchronizing on a single event, and each individual communication will occur only if the event succeeds and all the communications occur. Donnelly and Fluet (2006) show that this can be used to implement an n-way synchronization.

2.3 Implementing Transactional Events

Transactional events have been implemented as a library for Haskell. Calling `sync` performs a search through the space of possible ways of executing the event. When a thread calls `sync`, a new *search thread* is spawned, which performs the communications described.[4] At any source of nondeterminism, such as which branch of a `chooseEvt` to take, or which of several other threads communicating on a channel to communicate with, new search threads are spawned to search all the possibilities. We formalize a similar search strategy in detail in Section 5.

The goal of this search is to produce a set of search threads that have successfully completed by communicating with each other. The set must be closed; that is, no thread in the set can have communicated with any thread outside of it. Moreover, each program thread that is synchronizing may have produced multiple search threads, representing different ways of nondeterministically executing a given event. For correctness, the search is restricted to allow the final set of threads to contain at most one search thread corresponding to each program thread. This restriction guarantees that

[4] Performing communications between threads is obviously an imperative operation, which Haskell strictly controls. TE Haskell is built on top of STM Haskell (Harris et al. 2005a,b), which allows shared memory in the STM monad.

```
Thread 1:  sync (chooseEvt
             (thenEvt (sendEvt k1 2)
               (fun () -> sendEvt k2 3))
             (thenEvt (sendEvt k2 4)
               (fun () -> sendEvt k1 5)))
Thread 2:  sync (recvEvt k1)
Thread 3:  sync (recvEvt k2)
```

Figure 5. Three threads with a nondeterministic choice

```
let mapEvt f k1 k2 =
  thenEvt (recv k1) (fun x -> sendEvt k2 (f x))
```

Figure 6. A `thenEvt` running arbitrary code

there is a globally correct view of what each program thread did during the event.

We illustrate this restriction using Figure 5. Here, thread 1 may choose to send to the other two threads in either order. Threads 2 and 3 simply receive values. However, search threads for thread 1 will take both branches of the `chooseEvt`, so search threads for 2 and 3 will have the opportunity to communicate with either branch of the `chooseEvt`. For the search to complete, 2 and 3 must communicate with the *same* branch of the `chooseEvt`. If they did not, multiple search threads spawned from thread 1 would appear in the final set of threads, and there would be no globally correct view of what thread 3 did.

However, the implementation must also consider the effects of functions passed into `thenEvt`. Such functions can contain arbitrary user code, including code with side-effects. For example, Figure 6 contains a call to a function `f` inside a `thenEvt`.

Transactional events for Haskell use the Haskell type system to prevent side effects in `thenEvt`s.[5] The Event library is not part of the IO monad, which means that any code passed to `thenEvt` is purely functional. `thenEvt` in a setting without similar purity guarantees is the major topic investigated in our work and will be discussed in the following section.

3. Adding TE to Caml

Combining transactional events with an impure language introduces unexpected complications, particularly when threads share data via reads and writes to a global heap. Section 3.1 describes three necessary properties of a semantics for mutation within transactions. Sections 3.2 and 3.3 then describe two extreme and unsuitable approaches: disallowing mutation within events and allowing threads to access the heap in any order. The benefits of each extreme and the examples they support motivate the solution we describe in Section 3.4: heap accesses are made in "chunks" delineated by sends and receives. Section 3.5 considers other effectful features, most importantly nested synchronizations.

3.1 Three Properties of Mutation within Events

In this section, we identify three properties of mutation within transactions that would lead to nonsensical behavior if violated. The approaches in Sections 3.2–3.4 all respect these properties, but Sections 3.2 and 3.3 suffer from other limitations.

[5] Nontermination and exceptions remain possible. Nontermination of code in `thenEvt`s does not prevent another path through the event from completing successfully, so only performance suffers. Uncaught exceptions in an event are considered the same as `neverEvt`, i.e., the event fails.

First, synchronizing on an unsuccessful event should have no observable effect; that is, it must remain *atomic*. Take the following example:

```
sync (thenEvt (sendEvt k1 0)
  (fun _ -> r := 47; recvEvt k2))
```

If the send succeeds but the receive does not, then neither the send nor the update to `r` may be visible to other threads.

Second, if there are multiple ways in which a given synchronization may succeed, the effects of each alternative should be *isolated* from one another. Consider this code:

```
sync (chooseEvt
  (thenEvt (recvEvt k1)
    (fun _ -> r := 43; recvEvt k2))
  (thenEvt (recvEvt k1)
    (fun _ -> r := !r + 1; recvEvt k2)))
```

In this example, if the `chooseEvt` succeeds and `r` is 17 initially, then we expect that the final value of `r` will be either 43 or 18. If the right side were to mistakenly see the left's write of 43 to `r`, we would get a final value of 44.

Third, heap reads and writes must be *consistent* with respect to thread communication order. For example:

```
Thread 1:  sync (thenEvt (recvEvt k1)
             (fun _ -> r := 45; sendEvt k2 0))

Thread 2:  sync (thenEvt (recvEvt k2)
             (fun _ -> sendEvt k3 (!r)))
```

If these threads successfully synchronize together, the communication on `k2` implies that Thread 2's read of `r` comes after Thread 1's write, so Thread 2 must send 45 on `k3`.

We seek an implementation of transactional events that supports the three properties these examples illustrate.

3.2 One Extreme: Disallowing Mutation

As a first attempt to solve our mutation problems, consider a semantics in which any attempt to read from or write to a mutable heap location would result in a dynamic error (or alternatively, the enclosing event failing to synchronize). This solution seems reasonable; threads should ideally be using message-passing to communicate, so why would a programmer need mutable state within a transaction?

However, disallowing mutation within transactions would break functional abstractions. ML functions and data structures commonly make use of internal mutable state while presenting a purely functional interface. For instance, a function might memoize results of previous calls to increase efficiency.

As an example, Figure 7 gives two implementations of a simple dictionary interface. The functional implementation `FunDict` implements `lookup` using pure list lookup. `CacheDict` caches the most recently looked-up element in a reference cell. Although `CacheDict` has an impure implementation, its interface is purely functional, so clients should be able to use `CacheDict` wherever they use `FunDict`. Disallowing mutation within events breaks this abstraction, as in this example:

```
(* ('a * 'b channel) channel ->
   ('a, 'b) CacheDict.t -> unit *)
let lookupDictEvt k1 d =
  thenEvt (recv k1)
          (fun (k2, key) ->
             sendEvt k2 (CacheDict.lookup key d))
```

Mutation is useful in transactional events even when not hidden under a functional interface. The TE implementation of guarded

```
module type DICT = sig
  type ('key,'val) t
  val empty : ('key,'val) t
  val add : 'key -> 'val -> ('key,'val) t ->
              ('key,'val) t
  val lookup : 'key -> ('key,'val) t -> 'val
end

module FunDict : DICT = struct
  type ('key,'val) t = ('key * 'val) list
  let empty = []
  let add key val dict = (key, val)::dict
  let lookup key dict =
    snd (List.find (fun (k',_) -> k = k') dict)
end

module CacheDict : DICT = struct
  type ('key,'val) t =
    ('key * 'val) option ref * FunDict.t
  let empty = (ref NONE, FunDict.empty)
  let add key val (r,d) =
    (r, FunDict.add(key,val,d))
  let lookup key (r,d) =
    match r with
      (ref (SOME(k,v)) when k=key -> v
    | _ -> let ans = FunDict.lookup d in
           r := SOME(key,ans);
           ans
end
```

Figure 7. Two implementations of dictionary lookup, one functional and one using a mutable field as a cache.

receive in Figure 4 requires a user-provided predicate to approve the received value. The following example uses a reference cell within the guard to guarantee that one or more threads receive strictly increasing integer values.

```
let increasingRecvEvt k r =
  guardedRecvEvt k (fun x -> let i = !r in
                             let _ = r := x in
                             x > i)
```

In short, there are a number of practical idioms that read or write the heap within a transaction. Disallowing mutation within transactions is simply not in the mostly-functional spirit of ML.

3.3 The Other Extreme: Refservers

Consider an alternative solution in which threads in a transaction may freely read and modify the heap in any order. Again such behavior initially seems reasonable; non-transactional code is not restricted from modifying the heap at any time, so why restrict transactional code? Moreover, there is a straightforward way of extending TE to support this semantics. Recall the "refserver" interface introduced in Figure 2, in which a CML thread simulates a reference cell. We could map each heap location to a refserver thread, and translate heap reads and writes to receives and sends, respectively. Unfortunately, we shall see that this potential solution is expensive to implement and encourages ugly programming idioms.

One problem is that transactional events guarantee *completeness* as well as correctness: if a successful transaction exists, our implementation would be required to find it, potentially having to deal with a very large search space. Even without references, the size of the search space of possible transactions can be very large. However, TE is "pay-as-you-go": the size of the search space is exponential in the complexity of the program's communication protocol. For example, a pure guarded receive is a simple communication protocol, so the space needed for finding a successful transaction is relatively small. By turning reference reads and writes into communications, we would be greatly increasing the protocol complexity, and hence blowing up the search space.

Even if we could ignore the potential search-space blowup, this solution's guarantees are stronger than we believe is appropriate. For example, the following code would always succeed, because it abuses the completeness of the refserver solution by assuming that two threads' heap accesses will interleave in a specific way. We are unaware of any practical examples in which such an ugly programming idiom is necessary or useful.

Thread 1:

```
sync (thenEvt (sendEvt k 0)
  (fun _ -> r := 1; r := 0; alwaysEvt ()))
```

Thread 2:

```
sync (thenEvt (recvEvt k)
  (fun _ ->
    if !r = 1 then alwaysEvt () else neverEvt))
```

3.4 A Solution: Chunking

We have a dilemma. We cannot disallow mutation, as we do not want to break functional abstractions. Allowing any sequence of reads and writes is unacceptably expensive and encourages ugly programming idioms. We propose a compromise (explained below) between these two extremes: *chunking* heap accesses to allow mutation without sacrificing efficiency. Chunking allows mutation like the refserver solution, yet it can be implemented more efficiently and allows all useful examples we have encountered.

Our solution is to restrict the allowable interleavings of heap reads and writes to those in which threads are interleaved only when a thread reaches a communication event (`sendEvt` or `recvEvt`) or completes its sync. This restriction divides each thread's heap accesses into *chunks*, atomic blocks of code separated by sends and receives. We guarantee that if a successful transaction using chunked heap accesses exists, our implementation will find it; we also guarantee that any solution found by our implementation chunks heap accesses, as well as obeying the properties discussed in Section 3.1. The key trade-off here is that we are allowing mutation within a transaction without requiring an expensive search of all possible thread interleavings.

Chunking is both a reasonable and useful restriction on the behavior of heap accesses in transactions, as our next example shows. Suppose we have a function that generates unique identifiers by incrementing a shared counter:

```
let counter := ref 0
let getID () =
  let ans = !counter in
  let _   = counter := ans + 1 in
  ans
```

Now consider a transaction in which two threads both call `getID`:

```
Thread 1:  sync (thenEvt (sendEvt k 42)
             (fun _ -> sendEvt k1 (getID ())))

Thread 2:  sync (thenEvt (recvEvt k)
             (fun _ -> sendEvt k2 (getID ())))

Thread 3:  sync (thenEvt (recvEvt k1)
             (fun x -> thenEvt (recvEvt k2)
               (fun y -> alwaysEvt (x, y))))
```

Suppose that `counter` is 0 when the transaction begins. If we permit any thread interleaving within a transaction, we would ex-

pect that when the transaction completed, Thread 3's result could be `(0,0)`, `(0,1)` or `(1,0)`. `(0,0)` represents a transaction in which both reads of `counter` completed before either write; `(0,1)` and `(1,0)` occur if one of the threads completes its update to `counter` before the other reads `counter`. Restricting the set of correct transactions to those with "chunked" interleavings, the results `(0,1)` and `(1,0)` are legal, but `(0,0)` is not, because each thread's chunk includes both a write and a read. In this case, our semantics disallows the incorrect interleaving.

All the examples discussed in Section 3.2 run correctly under chunking. For example, any memoized function makes its updates to the memo table without using sends or receives, so the updates are executed in a contiguous block. Similarly, `CacheDict`'s lookup function reads and updates the cache within a single chunk.

3.5 Other Side Effects

The primary contribution of this paper is a semantics for mutation within transactional events, but ML has several other effectful features whose behavior in the presence of transactions deserves consideration. Our implementation supports nested synchronization, and we expect that existing techniques in the literature for exceptions, I/O, and thread creation can be incorporated.

We say that a *nested sync* occurs when a thread synchronizes on an event while executing transactional code. Consider the following code, which sequences two receives using a nested sync:

```
sync (thenEvt (recvEvt k1)
  (fun x -> let y = sync (recvEvt k2) in
            alwaysEvt (x, y)))
```

It is reasonable to expect that this code is functionally equivalent to a version using `thenEvt` to sequence the receives:

```
sync (thenEvt (recvEvt k1)
  (fun x -> thenEvt (recvEvt k2)
    (fun y -> alwaysEvt (x, y))))
```

Note, too, that library code with a purely functional interface may use events internally to perform multithreaded computation. For example, we could have implemented Section 3.4's unique-identifier generator using a server thread that repeatedly sent unique identifiers on a channel. For calls to these library functions to work correctly, we need nested synchronizations. Hence, by supporting nested sync, we avoid breaking functional abstractions, much as we did by allowing mutation within transactions. This issue does not arise in Haskell because the monadic type system ensures no call to `sync` is evaluated as part of an event.

In our semantics, nested syncs simply execute as part of the same transaction as the outer sync. That is, any sends or receives in nested syncs are part of the same all-or-nothing communication protocol begun before reaching the nested sync. We discuss the details of supporting nested sync in Sections 4 and 7.

We anticipate that existing work on exceptions and thread creation will apply to our system. Donnelly and Fluet (2006) discuss exceptions in the original TE paper; they suggest that any event that raises an exception become a `neverEvt`, with a `catchEvt` combinator to avoid aborting the transaction. Thread creation in traditional software transactions has been addressed in prior work (Jagannathan et al. 2005; Ziarek and Jagannathan 2008; Moore and Grossman 2008). Moore and Grossman propose two separate semantics for thread creation: internally parallel threads that execute within a transaction, and on-commit threads that execute at top-level after the transaction commits. We expect that it would be possible to implement both types of thread creation within TE.

Arbitrary irreversible I/O is essentially not supportable within transactions, but existing pragmatic techniques for transactional memory systems — particularly buffering of input and output, as in AtomCaml (Ringenburg and Grossman 2005) — seem reasonable for transactional events as well. Such systems buffer output until after the transaction has successfully completed. If the transaction fails, the output is discarded. Similarly, input can be buffered, with any input read put back in the buffer if the transaction failed.

Expressions:

$$\begin{array}{lcl} e & ::= & () \mid l \mid \kappa \mid x \mid \lambda x.e \mid e\ e \mid \\ & & \mathsf{ref}\ e \mid e := e \mid !e \mid \\ & & \mathsf{create}\ e \mid \mathsf{sync}\ e \mid \mathsf{newChan} \mid \\ & & \mathsf{alwaysEvt}\ e \mid \mathsf{neverEvt} \mid \mathsf{thenEvt}\ e\ e \mid \\ & & \mathsf{chooseEvt}\ e\ e \mid \mathsf{sendEvt}\ e\ e \mid \mathsf{recvEvt}\ e \end{array}$$

Values:

$$\begin{array}{lcl} v & ::= & () \mid l \mid \kappa \mid \lambda x.e \mid \\ & & \mathsf{alwaysEvt}\ v \mid \mathsf{neverEvt} \mid \mathsf{thenEvt}\ v\ v \mid \\ & & \mathsf{chooseEvt}\ v\ v \mid \mathsf{sendEvt}\ v\ v \mid \mathsf{recvEvt}\ v \end{array}$$

Mutable heap:

$$H ::= \cdot \mid H, l \mapsto v$$

Evaluation contexts:

$$\begin{array}{lcl} M & ::= & [\cdot] \mid M\ e \mid v\ M \mid \\ & & \mathsf{ref}\ M \mid !M \mid M := e \mid v := M \mid \\ & & \mathsf{sync}\ M \mid \mathsf{alwaysEvt}\ M \mid \\ & & \mathsf{thenEvt}\ M\ e \mid \mathsf{thenEvt}\ v\ M \mid \\ & & \mathsf{chooseEvt}\ M\ e \mid \mathsf{chooseEvt}\ v\ M \mid \\ & & \mathsf{sendEvt}\ M\ e \mid \mathsf{sendEvt}\ v\ M \mid \mathsf{recvEvt}\ M \\ \mathcal{M} & ::= & \cdot \mid M{:}\mathcal{M} \end{array}$$

Threads:

$$\begin{array}{lcl} T & ::= & \langle \theta, e \rangle \\ \overline{T} & ::= & \cdot \mid T \parallel \overline{T} \end{array}$$

Transaction syntax during execution:

$$\begin{array}{lcl} E & ::= & [\cdot] \mid \mathsf{thenEvt}\ E\ v \\ K & ::= & \langle \theta, \mathcal{M}, e \rangle \\ \overline{K} & ::= & \cdot \mid K \parallel \overline{K} \\ \tilde{K} & ::= & \cdot \mid K \end{array}$$

Figure 8. TE syntax.

$\boxed{H; e \hookrightarrow H'; e'}$

$$\textsc{Context}\ \frac{H; e \hookrightarrow H'; e'}{H; M[e] \hookrightarrow H'; M[e']} \qquad \textsc{App}\ \frac{}{H; (\lambda x.e)\ v \hookrightarrow H; e[v/x]}$$

$$\textsc{NewRef}\ \frac{l \text{ fresh}}{H; \mathsf{ref}\ v \hookrightarrow H, l \mapsto v; l} \qquad \textsc{GetRef}\ \frac{}{H; !l \hookrightarrow H; H(l)}$$

$$\textsc{SetRef}\ \frac{}{H; l := v \hookrightarrow H, l \mapsto v; ()} \qquad \textsc{NewChan}\ \frac{\kappa \text{ fresh}}{H; \mathsf{newChan} \hookrightarrow H; \kappa}$$

Figure 9. Single-threaded semantics.

4. High-Level Semantics

We define a high-level, nondeterministic semantics for transactional events that clarifies the design decisions discussed in Section 3. Our goal is that the high-level semantics provide a clear definition of which transactions can succeed in our implementation. The

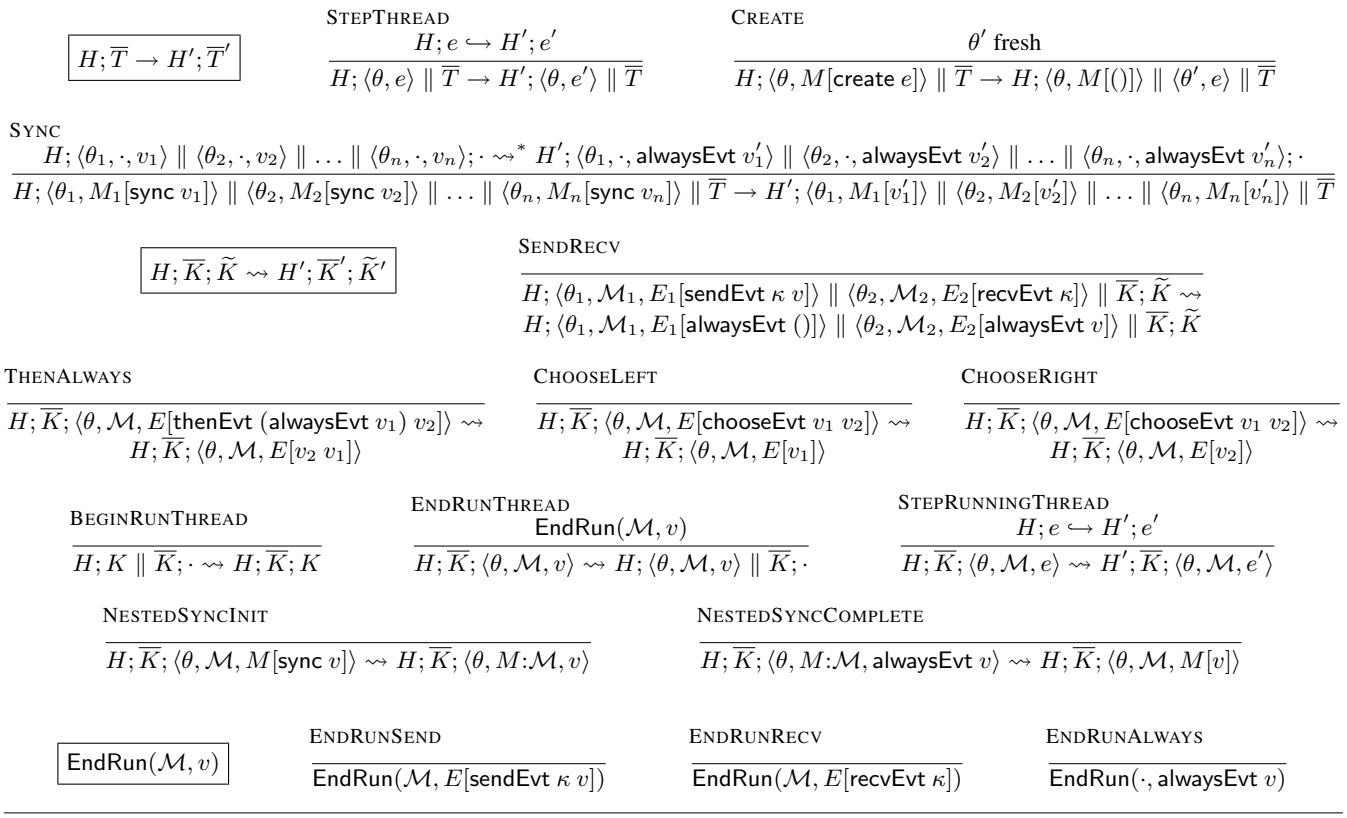

Figure 10. High-level semantics.

semantics enforces chunking of heap accesses as described in Section 3.4 and allows nested sync as described in Section 3.5.

Syntax and basic evaluation steps We model ML as a call-by-value lambda calculus with a mutable heap and threads. The syntax (Figure 8) includes primitives for mutable references, thread creation, channels, and events. We provide six composable event combinators, as well as the newChan keyword for creating a new channel and the sync keyword for synchronizing on an event. Value forms are $()$, heap locations l, channels κ, and events. We omit data types such as integers or pairs for simplicity. A program state is a mutable heap H and a pool of threads T, each of which has a thread ID θ and an expression e. We use the $\parallel$ operator, assumed to be commutative and associative, to build pools of threads. The use of stacks of evaluation contexts and run-time transaction syntax is discussed below. Our language does not have a type system; although the Caml implementation uses standard ML-style types, type safety is orthogonal to the issues being considered.

The high-level semantics is divided into three levels of execution: single-threaded, multithreaded, and synchronous. The single-threaded transition $\hookrightarrow$ (Figure 9) performs basic function application, reads and writes the heap, and creates fresh channels. Note the use of evaluation contexts M to streamline the semantics and enforce call-by-value behavior.

The multithreaded transition $\to$ (Figure 10) is nondeterministic. It may choose any thread to take a basic evaluation step or to spawn a new thread. It may also choose a subset of threads to execute a transaction (rule SYNC). SYNC nondeterministically chooses a subset of the thread pool $\overline{T}$, finds a successful transaction using the subset's threads and $\rightsquigarrow$ transitions, and updates the threads' values in $\overline{T}$ to reflect the result of the transaction.

Events The $\rightsquigarrow$ transition (Figure 10) steps between pools of *syncing threads* $\overline{K}$. Each syncing thread consists of a thread ID θ, a stack of evaluation contexts $\mathcal{M}$ ($\mathcal{M}$ is nonempty in the presence of nested sync; described below), and an expression e. The syncing program state also includes an optional *running thread* $\widetilde{K}$ that we use to enforce chunked heap accesses. Threads move to the running-thread position to execute single-threaded steps, process chooseEvt or thenEvt combinators, or begin or end nested syncs. (To enforce chunking, we could require only heap accesses take place from the running-thread position, but such a semantics is equivalent and more complicated.)

As in TE Haskell, we have four rules for processing events:

- The CHOOSELEFT and CHOOSERIGHT rules nondeterministically take the left and right branches of a chooseEvt.
- The SENDRECV rule nondeterministically takes two threads, one of which is blocked at a sendEvt on a channel κ and one of which is blocked at a recvEvt on κ, and performs a communication between the two threads.
- THENALWAYS takes the completed left side of a thenEvt and applies the thunked code on the right side to the result.

We use *event contexts* E to evaluate under the left side of a thenEvt. An event is a tree of thenEvt and chooseEvt combinators. The syntax for E isolates the first event to process by moving down the left side of each thenEvt in the tree. After a THENALWAYS step, the expression $v_2\ v_1$ must be re-evaluated to an event using basic evaluation steps. These steps may read from or write to the heap, and we require that these heap accesses occur "all at once" with chunking.

Nondeterministic choice	High-level transition	Low-level equivalent
Which threads participate	SYNC	SYNCINIT, SYNCCOMPLETE
Which side of chooseEvt	CHOOSELEFT, CHOOSERIGHT	CHOOSE
Whether to perform a communication	SENDRECV	SENDRECV
Which thread enters run mode / What heap to use	BEGINRUNTHREAD	BEGINRUNTHREAD

Figure 11. Nondeterministic choices made while a transaction is executing.

Chunking We enforce the chunking of heap accesses described in Section 3.4 with an additional piece of program state $\widetilde{K}$, which either holds a *running thread* K or is empty ($\cdot$). A thread enters *run mode* with the BEGINRUNTHREAD rule, which requires that $\widetilde{K}$ be empty before the step. The STEPRUNNINGTHREAD rule allows the running thread to take single-threaded steps, possibly reading or writing the heap. Rule ENDRUNTHREAD moves the running thread out of run mode only when the thread completes its sync or reaches a communication (formalized with the predicate EndRun in Figure 10). Hence, threads executing within a transaction access the heap in atomic "chunks" delineated by communications. Note that the thread's run does *not* complete when the thread reaches a chooseEvt or thenEvt.

Nested sync The stack of evaluation contexts $\mathcal{M}$ included in each syncing thread implements nested sync. If a thread executing in run mode reaches a nested sync, its evaluation context is saved and pushed onto $\mathcal{M}$ (rule NESTEDSYNCINIT). When the nested sync completes, the most recent context is popped off the stack and the thread continues execution where it left off (rule NESTEDSYNCCOMPLETE). A top-level sync cannot complete unless the stack of contexts is empty.

Example Consider the following example, which assumes that our language includes integer constants and that $e_1; e_2$ is syntactic sugar for $(\lambda\_.e_2)\ e_1$.

Thread 1:
sync (thenEvt (alwaysEvt ()) ($\lambda\_.r := 22$; recvEvt κ))

Thread 2:
sync (thenEvt (alwaysEvt ()) ($\lambda\_.r := 45$; sendEvt κ (!r)))

The SYNC rule can select both threads to participate in a transaction. We then nondeterministically choose a thread to enter run mode. Suppose Thread 1 is chosen; it writes 22 to r and exits run mode waiting to receive on κ. Thread 2 then enters run mode, writes 45 to r, reads 45 from r, and exits run mode trying to send 45 on κ. Note it is not possible for Thread 2 to read 22 from r, no matter which thread enters run mode first. The two threads then communicate over κ, leaving Thread 1 with the expression alwaysEvt 45 and Thread 2 with expression alwaysEvt (). At this point, both threads have the form alwaysEvt v for some value v, so the sync succeeds and the two threads return to the normal pool of threads.

5. Low-Level Semantics

The section presents a low-level semantics that implements Section 4's high-level semantics with an exhaustive search of interactions among syncing threads. The low-level rules "determinize" several sources of nondeterminism, such as which branch of a chooseEvt to take or whether to communicate with another thread (Figure 11).

Program execution The low-level syntax and semantics appear in Figures 12 and 13, respectively. We have closely followed TE Haskell's low-level semantics, with some key changes to allow for mutation and nested sync. Our low-level program state consists of a global heap H that may be *frozen* as $\not{H}$, a pool of threads $\overline{T}$, a pool

Search threads:	S	::=	$\langle \theta, \widetilde{R}, \mathcal{M}, e, \rho \rangle$
	$\widetilde{R}$	::=	$\cdot \mid R$
Search heaps:	R	::=	$\langle H, \eta \rangle$
Paths:	$\hat{\rho}$	::=	Left | Right | Send($\langle \theta, \rho \rangle$) | Recv($\langle \theta, \rho \rangle$) | Heap($\eta$)
	ρ	::=	$\cdot \mid \hat{\rho}{:}\rho$
	η	::=	$\cdot \mid \langle \theta, \rho \rangle{:}\eta$
Program state:	h	::=	$H \mid \not{H}$
	$\overline{S}$	::=	$\cdot \mid S \parallel \overline{S}$
	$\overline{R}$	::=	$\cdot \mid R \parallel \overline{R}$
Trails:	τ	::=	$\langle \theta, \rho \rangle \mid \langle \mathsf{heap}, \eta \rangle$

Figure 12. Low-level runtime syntax (extends Figure 8).

of *search heaps* (explained below) $\overline{R}$, and a pool of *search threads* $\overline{S}$ (also explained below). Program execution takes place in two alternating stages: normal thread execution, during which the heap is unfrozen (H) and threads in $\overline{T}$ may make progress, and synchronization, during which the heap is frozen ($\not{H}$) and search heaps and search threads attempt to find a successful transaction. The program enters the synchronization stage with rule SYNCINIT, which initializes a single search heap and a pool of search threads from the current global heap and a set of threads that are blocked at a sync in $\overline{T}$. SYNCINIT nondeterministically chooses which threads to initialize as search threads; in practice, we take all threads blocked at a sync. The SYNCCOMPLETE rule exits synchronization when a set of search threads is *committable* (explained below) with a single search heap, committing the transaction's results back to the main pool $\overline{T}$. The synchronization stage cannot be stuck, even if no successful transaction exists: it is always possible to unfreeze the heap and return to normal thread execution by committing an empty set of search threads with the initial search heap created by SYNCINIT.

Search threads During synchronization, each thread T that is at a sync corresponds to one or more *search threads* S. A search thread consists of a thread ID θ, an optional first class heap $\widetilde{R}$ (explained below), a stack $\mathcal{M}$ of evaluation contexts for nested sync, an expression, and a path ρ (also explained below). Search threads perform a search for possible transactions by speculatively executing transactional code. As in the TE Haskell semantics, we search for a successful transaction by replicating search threads for each nondeterministic step. Therefore, the pool of search threads $\overline{S}$ may represent many different partially completed transactions.

When a search thread replicates, the path ρ of each replicated search thread is updated to record what nondeterministic choice was made. For example, in the CHOOSE rule, a search thread replicates as two search threads, one taking the left branch and one taking the right. Each replicated thread updates its path with either Left or Right. In SENDRECV, the choice is between performing a communication or not, so the sender and receiver each replicate as two search threads, one of which performed the communication and

$h; \overline{T}; \overline{R}; \overline{S} \to h'; \overline{T}'; \overline{R}'; \overline{S}'$

STEPTHREAD
$$\frac{H; e \hookrightarrow H'; e'}{H; \langle \theta, e \rangle \parallel \overline{T}; \cdot; \cdot \to H'; \langle \theta, e' \rangle \parallel \overline{T}; \cdot; \cdot}$$

CREATE
$$\frac{\theta' \text{ fresh}}{H; \langle \theta, M[\mathsf{create}\ e] \rangle \parallel \overline{T}; \cdot; \cdot \to H; \langle \theta, M[()] \rangle \parallel \langle \theta', e \rangle \parallel \overline{T}; \cdot; \cdot}$$

SYNCINIT
$$\frac{}{H; \langle \theta_1, M_1[\mathsf{sync}\ v_1] \rangle \parallel \langle \theta_2, M_2[\mathsf{sync}\ v_2] \rangle \parallel \ldots \parallel \langle \theta_k, M_n[\mathsf{sync}\ v_n] \rangle \parallel \overline{T}; \cdot; \cdot \to \mathcal{H}; \langle \theta_1, M_1[\mathsf{sync}\ v_1] \rangle \parallel \langle \theta_2, M_2[\mathsf{sync}\ v_2] \rangle \parallel \ldots \parallel \langle \theta_k, M_n[\mathsf{sync}\ v_n] \rangle \parallel \overline{T}; \langle H, \cdot \rangle; \langle \theta_1, \cdot, \cdot, v_1, \cdot \rangle \parallel \ldots \parallel \langle \theta_k, \cdot, \cdot, v_k, \cdot \rangle}$$

SYNCCOMPLETE
$$\frac{\{\langle \theta_1, \rho_1 \rangle, \langle \theta_2, \rho_2 \rangle, \ldots, \langle \theta_n, \rho_n \rangle, \langle \mathsf{heap}, \eta \rangle\} \text{ committable}}{\mathcal{H}; \langle \theta_1, M_1[\mathsf{sync}\ v_1] \rangle \parallel \langle \theta_2, M_2[\mathsf{sync}\ v_2] \rangle \parallel \ldots \parallel \langle \theta_n, M_n[\mathsf{sync}\ v_n] \rangle \parallel \overline{T}; \langle H', \eta \rangle \parallel \overline{R}; \langle \theta_1, \cdot, \cdot, \mathsf{alwaysEvt}\ v_1', \rho_1 \rangle \parallel \langle \theta_2, \cdot, \cdot, \mathsf{alwaysEvt}\ v_2', \rho_2 \rangle \parallel \ldots \parallel \langle \theta_n, \cdot, \cdot, \mathsf{alwaysEvt}\ v_n', \rho_n \rangle \parallel \overline{S} \to H'; \langle \theta_1, M_1[v_1'] \rangle \parallel \langle \theta_2, M_2[v_2'] \rangle \parallel \ldots \parallel \langle \theta_n, M_n[v_n'] \rangle \parallel \overline{T}; \cdot; \cdot}$$

SENDRECV
$$\frac{\langle \theta_1, \rho_1 \rangle \text{ and } \langle \theta_2, \rho_2 \rangle \text{ coherent}}{\mathcal{H}; \overline{T}; \overline{R}; \langle \theta_1, \cdot, \mathcal{M}_1, E_1[\mathsf{sendEvt}\ \kappa\ v], \rho_1 \rangle \parallel \langle \theta_2, \cdot, \mathcal{M}_2, E_2[\mathsf{recvEvt}\ \kappa], \rho_2 \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \overline{R}; \langle \theta_1, \cdot, \mathcal{M}_1, E_1[\mathsf{sendEvt}\ \kappa\ v], \rho_1 \rangle \parallel \langle \theta_2, \cdot, \mathcal{M}_2, E_2[\mathsf{recvEvt}\ \kappa], \rho_2 \rangle \parallel \langle \theta_1, \cdot, \mathcal{M}_1, E_1[\mathsf{alwaysEvt}\ ()], \mathsf{Send}(\langle \theta_2, \rho_2 \rangle) {:} \rho_1 \rangle \parallel \langle \theta_2, \cdot, \mathcal{M}_2, E_2[\mathsf{alwaysEvt}\ v], \mathsf{Recv}(\langle \theta_1, \rho_1 \rangle) {:} \rho_2 \rangle \parallel \overline{S}}$$

CHOOSE
$$\frac{}{\mathcal{H}; \overline{T}; \overline{R}; \langle \theta, R, \mathcal{M}, E[\mathsf{chooseEvt}\ v_1\ v_2], \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \overline{R}; \langle \theta, R, \mathcal{M}, E[v_1], \mathsf{Left}{:}\rho \rangle \parallel \langle \theta, R, \mathcal{M}, E[v_2], \mathsf{Right}{:}\rho \rangle \parallel \overline{S}}$$

THENALWAYS
$$\frac{}{\mathcal{H}; \overline{T}; \overline{R}; \langle \theta, R, \mathcal{M}, E[\mathsf{thenEvt}\ (\mathsf{alwaysEvt}\ v_1)\ v_2], \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \overline{R}; \langle \theta, R, \mathcal{M}, E[v_2\ v_1], \rho \rangle \parallel \overline{S}}$$

BEGINRUNTHREAD
$$\frac{\langle \mathsf{heap}, \eta \rangle \text{ and } \langle \theta, \rho \rangle \text{ coherent}}{\mathcal{H}; \overline{T}; \langle H', \eta \rangle \parallel \overline{R}; \langle \theta, \cdot, \mathcal{M}, e, \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \langle H', \eta \rangle \parallel \overline{R}; \langle \theta, \cdot, \mathcal{M}, e, \rho \rangle \parallel \langle \theta, \langle H', \eta \rangle, \mathcal{M}, e, \rho \rangle \parallel \overline{S}}$$

ENDRUNTHREAD
$$\frac{\mathsf{EndRun}(\mathcal{M}, v)}{\mathcal{H}; \overline{T}; \overline{R}; \langle \theta, \langle H', \eta \rangle, \mathcal{M}, v, \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \langle H', \langle \theta, \rho \rangle {:} \eta \rangle \parallel \overline{R}; \langle \theta, \cdot, \mathcal{M}, v, \mathsf{Heap}(\eta) {:} \rho \rangle \parallel \overline{S}}$$

STEPRUNNINGTHREAD
$$\frac{H'; e \hookrightarrow H''; e'}{\mathcal{H}; \overline{T}; \overline{R}; \langle \theta, \langle H', \eta \rangle, \mathcal{M}, e, \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \overline{R}; \langle \theta, \langle H'', \eta \rangle, \mathcal{M}, e', \rho \rangle \parallel \overline{S}}$$

NESTEDSYNCINIT
$$\frac{}{\mathcal{H}; \overline{T}; \langle \theta, R, \mathcal{M}, M[\mathsf{sync}\ v], \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \langle \theta, R, M{:}\mathcal{M}, v, \rho \rangle \parallel \overline{S}}$$

NESTEDSYNCCOMPLETE
$$\frac{}{\mathcal{H}; \overline{T}; \langle \theta, R, M{:}\mathcal{M}, \mathsf{alwaysEvt}\ v, \rho \rangle \parallel \overline{S} \to \mathcal{H}; \overline{T}; \langle \theta, R, \mathcal{M}, M[v], \rho \rangle \parallel \overline{S}}$$

Figure 13. Low-level semantics (*coherent* and *committable* defined in text; EndRun unchanged from Figure 10).

one of which did not. The communicating pair both update their paths with Send or Recv. The final type of path element, Heap, indicates that the thread entered run mode (explained below).

Search heaps Search heaps implement the chunking semantics discussed in the previous section. Unlike the high-level semantics, in which syncing threads had direct access to the global heap, the search threads work with local first-class heaps, stored in search heaps R. Each R includes a heap path η, which records the search threads that contributed toward producing that first-class heap. Every successful transaction produces one search heap that represents the final state of the global heap for that transaction.

During a transaction, search heaps move among search threads by moving in and out of the pool of available search heaps $\overline{R}$. The BEGINRUNTHREAD rule copies a search heap from $\overline{R}$ into a search thread's local state. When a search thread exits run mode, ENDRUNTHREAD adds the updated search heap to the pool. ENDRUNTHREAD updates the paths in the search thread and search heap.

Nested sync Each search thread stores a list of evaluation contexts ($\mathcal{M}$) for nested sync. The details of nested sync ($\mathcal{M}$, NESTEDSYNCINIT and NESTEDSYNCCOMPLETE) are identical to their high-level equivalents.

Committability and coherency The low-level semantics takes a SENDRECV or BEGINRUNTHREAD step only if the step might lead to a successful transaction. The communicating threads (or in the case of BEGINRUNTHREAD, the thread entering run mode and its new heap) must have a consistent view of what has happened thus far in the transaction. We formalize this requirement by defining when two *trails* are *coherent*, where a trail is either a "thread trail" $\langle \theta, \rho \rangle$ or a "heap trail" $\langle \mathsf{heap}, \eta \rangle$. In addition, the pools of search threads and search heaps may correspond to many different transactions, and it is important that the threads and heap used in the final successful transaction have observed the same transaction in progress. To that end, we define the notion of *committability*, which checks that the paths in a set are all consistent with one another.

To explain coherence and committability, we first define the $\succeq$ (*extends*) operator for paths and the *dependency set* for a trail.

DEFINITION 1 (Extends).
A path ρ extends *a path* ρ' *(*$\rho \succeq \rho'$*) if*
- $\rho = \rho'$*, or*
- *there exist* $\hat{\rho}$ *and* ρ'' *such that* $\rho = \hat{\rho}{:}\rho''$ *and* $\rho'' \succeq \rho$.

The $\succeq$ operator applies similarly to heap paths η.

A trail's dependency set is the set of trails with which it has interacted, directly or indirectly.

DEFINITION 2 (Dependency set).
For all trails τ, we define $\mathsf{Dep}(\tau)$ *as follows:*

$$\mathsf{Dep}(\tau) \quad = \quad \{\tau\} \cup \mathsf{DepAux}(\tau)$$

where we define $\mathsf{DepAux}(\tau)$ *as:*

$$\begin{array}{l}
\mathsf{DepAux}(\langle\theta, \cdot\rangle) = \{\} \\
\mathsf{DepAux}(\langle\theta, \mathsf{Left}{:}\rho\rangle) = \mathsf{DepAux}(\langle\theta, \rho\rangle) \\
\mathsf{DepAux}(\langle\theta, \mathsf{Right}{:}\rho\rangle) = \mathsf{DepAux}(\langle\theta, \rho\rangle) \\
\mathsf{DepAux}(\langle\theta, \mathsf{Send}(\langle\theta', \rho'\rangle){:}\rho\rangle) = \{\langle\theta', \mathsf{Recv}(\langle\theta, \rho\rangle){:}\rho'\rangle\} \cup \\
\quad \mathsf{DepAux}(\langle\theta, \rho\rangle) \cup \mathsf{DepAux}(\langle\theta', \rho'\rangle) \\
\mathsf{DepAux}(\langle\theta, \mathsf{Recv}(\langle\theta', \rho'\rangle){:}\rho\rangle) = \{\langle\theta', \mathsf{Send}(\langle\theta, \rho\rangle){:}\rho'\rangle\} \cup \\
\quad \mathsf{DepAux}(\langle\theta, \rho\rangle) \cup \mathsf{DepAux}(\langle\theta', \rho'\rangle) \\
\mathsf{DepAux}(\langle\theta, \mathsf{Heap}(\eta){:}\rho\rangle) = \{\langle\mathsf{heap}, \langle\theta, \rho\rangle{:}\eta\rangle\} \cup \\
\quad \mathsf{DepAux}(\langle\theta, \rho\rangle) \cup \mathsf{DepAux}(\langle\mathsf{heap}, \eta\rangle) \\
\mathsf{DepAux}(\langle\mathsf{heap}, \cdot\rangle) = \{\} \\
\mathsf{DepAux}(\langle\mathsf{heap}, \langle\theta, \rho\rangle{:}\eta\rangle) = \{\langle\theta, \mathsf{Heap}(\eta){:}\rho\rangle\} \cup \\
\quad \mathsf{DepAux}(\langle\theta, \rho\rangle) \cup \mathsf{DepAux}(\langle\mathsf{heap}, \eta\rangle)
\end{array}$$

In practice, we maintain the dependency set incrementally.

We define *coherency* to guarantee that two threads communicate only when they have observed the same transaction thus far.

DEFINITION 3 (Coherent).
$\langle\theta_1, \rho_1\rangle$ and $\langle\theta_2, \rho_2\rangle$ are coherent *if:*

- $\theta_1 \neq \theta_2$*;*
- *For all* $\langle\theta_1, \rho_1'\rangle \in \mathsf{Dep}(\langle\theta_2, \rho_2\rangle)$*,* $\rho_1 \succeq \rho_1'$*;*
- *For all* $\langle\theta_2, \rho_2'\rangle \in \mathsf{Dep}(\langle\theta_1, \rho_1\rangle)$*,* $\rho_2 \succeq \rho_2'$*;*
- *For all* $\langle\theta, \rho\rangle \in \mathsf{Dep}(\langle\theta_1, \rho_1\rangle)$ *and* $\langle\theta, \rho'\rangle \in \mathsf{Dep}(\langle\theta_2, \rho_2\rangle)$*,* $\rho \succeq \rho'$ *or* $\rho' \succeq \rho$*; and*
- *For all* $\langle\mathsf{heap}, \eta\rangle \in \mathsf{Dep}(\langle\theta_1, \rho_1\rangle)$ *and* $\langle\mathsf{heap}, \eta'\rangle \in \mathsf{Dep}(\langle\theta_2, \rho_2\rangle)$*,* $\eta \succeq \eta'$ *or* $\eta' \succeq \eta$*.*

A similar definition applies when one trail is a thread trail and the other a heap trail, as in BEGINRUNTHREAD.

The SYNCCOMPLETE rule takes a set of complete search threads and a search heap and checks that the trails of each is *committable*. Committability formally defines the intuition that all of the threads have participated in the same transaction, and that no other search threads participated in that transaction.

DEFINITION 4 (Committable).
A set of trails C is committable *if:*

1. *The thread IDs θ of the trails in C are distinct from one another;*
2. *For all $\tau \in C$, if $\langle\theta, \rho\rangle \in \mathsf{Dep}(\tau)$, then there exists ρ' such that $\langle\theta, \rho'\rangle \in C$ and $\rho' \succeq \rho$; and*
3. *There exists a unique heap path η such that $\langle\mathsf{heap}, \eta\rangle \in C$. Moreover, for all $\tau \in C$, if $\langle\mathsf{heap}, \eta'\rangle \in \mathsf{Dep}(\tau)$, then $\eta \succeq \eta'$.*

In Section 6, we shall show that the low-level semantics is both correct and complete — that is, any transaction that succeeds in the low-level semantics could have succeeded in the high-level semantics, and vice versa. And unlike the high-level semantics, the low-level semantics is reasonable to implement in a real system, as we shall see in Section 7.

6. Proof of Equivalence

In Sections 4 and 5, we presented two distinct semantics for transactional events in an ML-like language: a high-level, nondeterministic semantics that defines the set of correct transactions, and a search-based low-level semantics, representing our Caml implementation. This section gives a very brief overview of our formal proof of equivalence for these two semantics.

Our proof establishes two core facts, Theorems 1 and 2.

THEOREM 1 (High-level to low-level).
If $H; \overline{T} \rightarrow H'; \overline{T}'$ in the high-level semantics, then $H; \overline{T}; \cdot; \cdot \rightarrow^* H'; \overline{T}'; \cdot; \cdot$ in the low-level semantics.

Theorem 1 states that the low-level semantics permits all possible program executions allowed by the high-level semantics. In particular, if a set of syncing threads successfully completes a transaction in the high-level semantics, then there exists a sequence of steps such that the same threads successfully synchronize in the low-level semantics. The proof maintains a correspondence between high-level and low-level search threads and demonstrates that at each step the set of low-level search threads is committable. Hence, when the high-level transaction completes, the low-level transaction may commit with SYNCCOMPLETE.

THEOREM 2 (Low-level to high-level).
If $h; \overline{T}; \overline{R}; \overline{S} \rightarrow h'; \overline{T}'; \overline{R}'; \overline{S}'$ in the low-level semantics and $\mathsf{SyncSim}(h; \overline{T}; \overline{R}; \overline{S})$*, then* $\mathsf{SyncSim}(h'; \overline{T}'; \overline{R}'; \overline{S}')$ *and* $\mathsf{unfreeze}(h); \overline{T} \rightarrow^* \mathsf{unfreeze}(h'); \overline{T}'$ *in the high-level semantics, where* $\mathsf{unfreeze}(H) = H$ *and* $\mathsf{unfreeze}(\not{H}) = H$.

Theorem 2 states that the low-level semantics does not introduce new possible program executions. At each step in the low-level semantics, we erase any incomplete transactions to translate to the high-level state. The theorem uses a key invariant SyncSim (the formal definition of which is omitted here) which holds when low-level program state is *synchronously simulable*: the pools of search threads and search heaps represent valid evolutions of the high-level semantics. Hence, when a low-level transaction commits with SYNCCOMPLETE, the high-level semantics may take a single SYNC step to "catch up."

Together, these two theorems demonstrate that our two semantics are equivalent. Both proofs rely on the notion of a *committable set* by mapping between a pool of high-level syncing threads and a committable set of low-level search threads. There are many subtleties and corner cases in showing that committability is preserved with each step; hence, we found it useful to mechanize our proof in the Coq proof assistant so as to avoid missing crucial details. Readers are encouraged to peruse our Coq code at `http://wasp.cs.washington.edu/tecaml` to get an idea of the proof techniques used for Theorems 1 and 2. The proof is large (some 15,000 lines of Coq code) but reassuringly similar to TE Haskell's paper proof of correctness (Donnelly and Fluet 2008).

7. Implementation

We have created a prototype implementation of TE Caml for Objective Caml. Our implementation consists of modifications to the thread scheduler and bytecode interpreter and a Caml library that uses these to implement TE. All the examples in this paper and some other small programs produce the correct answer and generate the expected set of search threads and search heaps. The development and evaluation of larger benchmarks remains future work.

7.1 OCaml Library

The core of our prototype is an OCaml library that uses the runtime changes described in Sections 7.2–7.4 to implement a system that closely follows our low-level semantics. A thread syncing on an event blocks while search threads speculatively execute all possible nondeterministic executions of an event. If a search thread encounters a nondeterministic choice, such as which branch in a `chooseEvt`, or what thread to communicate with, it spawns new search threads for all the choices. A search thread carries a path, as in the low-level semantics, to record the choices that identify that search thread's execution of the event. This lets us check that the views of the past seen by communicating threads are coherent.

Once a transaction completes, the library kills all search threads. We do not wait for a search thread to reach some sort of event before killing it because an infinite loop involving no such events could mean this never occurs. Therefore, run-time support for killing (search) threads is important. It is an improvement over the Haskell implementation, which can never kill search threads that do not return from the function call caused by `thenEvent`.

An event is implemented as a function that takes a state, containing a path and first class heap, and returns a new state and a value, which is the value the event returns when synchronized on. `sync` calls this function with an initial state; events containing other events call the functions representing their subevents.

Channels are implemented as two lists of threads: those waiting for sends and those waiting for receives. A communicating thread wakes all the threads in one list, supplying them with a value if it is sending, and then adds itself to the other list and sleeps. An awoken thread spawns a new copy of itself to run, possibly with a newly received value if it is receiving, and then returns to sleep. Note that unlike a Concurrent ML implementation, completeness requires trying all matching communications, not just one.

7.2 First Class Heaps

Our semantics prevent effects in events from being seen before events succeed through the use of multiple, first class heaps. Part of the state of a running search thread is the current heap that should be used in code run in a `thenEvt`.

We modified the bytecode interpreter to write to different locations in the actual heap depending on which first class heap was currently associated with the currently running search thread. This required checking on each access of the heap to discover if there is currently a first class heap that should be used.

While other work has investigated sophisticated techniques for representing first class heaps, our proof-of-concept implementation uses very simple heaps. We represent first class heaps as Caml lists of tuples to take advantage of the functional nature of ML data structures. A heap write simply adds a tuple to the head of the list, which lets us exploit tail sharing for immutable lists.

Our first class heaps are implemented lazily. Until client code writes to a heap location, the ordinary OCaml heap is used instead. Once a write to a location is encountered for a given first class heap, that location is added to the first class heap and all further reads or writes to that location use the first class heap.

When a committable set of search threads is found, the resulting heap is copied back into the global OCaml heap. This copying must occur atomically, which is simple because OCaml runs only one thread at a time.

7.3 Stack Copying Thread Creation

While speculatively executing events, our implementation forks off search threads to continue executing with different nondeterministic choices. Such execution may include computation higher up the call stack than the location of the thread creation.

To do this, we added stack copying thread creation to the OCaml thread library, as a function called `fork`.[6] This function is called by a thread and creates a copy of the calling thread, like Unix's process-level fork. It returns twice, once in the parent thread and once in the new child thread, where each thread is the same except for the return value of the function. Implementing `fork` requires copying the thread data structure in the OCaml scheduler. In addition, we copy the entire thread stack for the duplicated thread.

TE Haskell solved a similar problem without adding new thread creation primitives to the runtime. Instead, they use a continuation passing style (CPS) search, where explicit continuations are used rather than the program call stack. This allows them to spawn search threads using Haskell's `forkIO`, which is not stack copying.

Such a CPS search is not possible for us due to the complications presented by nested calls to `sync`, which is discussed in Sections 3.5 and 4. A CPS implementation must be able to get an explicit continuation for every call to an event inside a call to `sync`. However, nested `sync`s are in arbitrary code in `thenEvt`s. On returning from the call to `sync`, this code should continue running: the continuation that would be needed is the current Caml continuation. Stack copying fork allows the desired behavior. (First-class continuations would also suffice.)

7.4 Thread Scheduling

While a transaction is accessing and modifying a first-class heap, a regular program thread could potentially modify the regular OCaml heap in a conflicting way. To prevent this, we do not interleave program threads and search threads. A call to `sync` creates an initial search thread, which is scheduled; henceforth, only search threads are scheduled until a committable set is found or all search threads have blocked. We can then kill all search threads and schedule program threads again. The next time a `sync` is called, any initial search threads created by calls to `sync`s that have not yet completed are recreated using the current heap, along with the new search thread created by `sync`. This scheduling method ensures that all calls to `sync` that participate in the same communication see the same initial heap, and no program threads modify the heap while a transaction executes.

However, such a solution allows search threads to starve program threads if a function in a `thenEvt` does not terminate. In this case, the search for a committable set of threads would never fail, but would not terminate. To avoid this, we only let the set of search threads run for a given total time before killing the search. In this case, the implementation will re-run the search later for more time, exponentially increasing the time on each iteration. To maintain fairness, we also run program threads for an increased amount of time before re-running the search threads.

7.5 Future Work

Several aspects of our implementation could be optimized.

First, it is important to minimize how many search heaps get created. Because most chunks do not actually mutate memory, it should be possible to avoid increasing the size of the search-heap pool in the common case.

Second, our prototype checks on every read in a transaction whether the value should be read from a first-class heap or the regular heap. However, only mutable values may be written to, so, since our heaps are lazy, only mutable values will ever be in a first-class heap. Mutability information is known statically via the type system. By propagating this information to the code generator, we could use different bytecodes for reading mutable data. Only these reads would need to check the first-class heap.

Third, we could use a representation for first-class heaps more sophisticated than simple association lists.

Finally, our stack copying thread creation is inefficient. It is unnecessary to copy the entire stack. Search threads never return past the call to `sync` that generated them. We could copy only that far back, essentially using a delimited continuation.

In addition to these efficiency improvements, we could add support for more side effects, particularly I/O, in transactional events. As mentioned in Section 3.5, some transactional memory systems allow some forms of I/O via buffering. While similar techniques should work with TE, a crucial difference of our system is that a successful transaction consists of multiple threads. We believe that adding I/O to our TE implementation would involve treating I/O

[6] The existing OCaml function `Thread.create` creates threads with an empty stack and so is inappropriate for our needs.

buffers the same way we treat first-class heaps: chunking accesses to buffers and passing first-class buffers among search threads.

8. Related Work

While our work has extended transactional events (Donnelly and Fluet 2006) to support a mutable heap and nested synchronizations, it is also related to the design and implementation of other CML-style systems. Moreover, our approach of starting with message-passing and giving semantics to shared-memory accesses is in contrast to work that starts with shared-memory transactions and allows communication within transactions.

CML-style systems In addition to Reppy's original implementation (Reppy 1999) for Standard ML, CML or similar synchronous message-passing systems have been used in Haskell (Russell 2001), Scheme (Flatt and Findler 2004) and Caml (Leroy 2007), with the Scheme system supporting a modular way to kill threads. Recent work on implementing CML more efficiently has focused on static analysis of communication protocols (Reppy and Xiao 2007) and support for multiprocessors (Reppy and Xiao 2008). Stabilizers (Ziarek et al. 2006) provide checkpointing for CML programs even in the presence of inter-thread communication and updates to shared memory.

Our formal semantics follows the TE work (Donnelly and Fluet 2008) with the contributions of supporting ML's impurities and using Coq to formalize the correctness proof. One pleasing aspect of the TE combinators is that they form a monad-with-plus whereas prior work on semantics for CML-style languages (Panangaden and Reppy 1997; Jeffrey 1995; Russell 2001) demonstrated that CML does not obey the monad laws.

Transactions Starting with Harris and Fraser (2003), much recent work has considered making shared-memory concurrency easier by enriching programming languages with transactions, i.e., a primitive for executing multiple memory operations atomically. Designs and implementations for Caml (Ringenburg and Grossman 2005), Haskell (Harris et al. 2005a,b), and Scheme (Kimball and Grossman 2007) have been proposed. Our chunking mechanism is powerful enough to encode a transaction though it is not an efficient approach. It is also possible to build a shared-memory transactions library on top of TE (Donnelly and Fluet 2006).

While shared-memory and message-passing systems are difficult to compare, some recent work allowing inter-thread communication within shared-memory transactions is relevant because, like our work, it combines the two paradigms. One approach breaks the isolation of transactions to allow communication for idioms such as barriers (Smaragdakis et al. 2007). Another approach allows parallelism *within* shared-memory transactions by letting multiple threads communicate via CML while still forbidding communication outside the transactions (Ziarek and Jagannathan 2008).

9. Conclusion

We designed, formally specified, and implemented transactional events for ML. This work brings the power of extending CML with the `thenEvt` combinator to a language with side effects. The key to the semantics is allowing thread interleavings only when messages are exchanged. We proved our search-based low-level semantics correct with respect to our high-level semantics and implemented it with a few key extensions to the Objective Caml run-time system.

Acknowledgments

The anonymous reviewers and the WASP group at the University of Washington provided excellent feedback on this paper's presentation. Matthew Fluet provided invaluable help, particularly for understanding the equivalence proof for TE Haskell.

References

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

Kevin Donnelly and Matthew Fluet. Transactional events. *The Journal of Functional Programming*, 2008. To appear.

Kevin Donnelly and Matthew Fluet. Transactional events. In *11th ACM International Conference on Functional Programming*, 2006.

Matthew Flatt and Robert Bruce Findler. Kill-safe synchronization abstractions. In *ACM Conference on Programming Language Design and Implementation*, 2004.

Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2005a.

Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a shared-memory multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, 2005b.

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

Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony L. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2), 2005.

Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *The Symposium on Logic in Computer Science*, 1995.

Aaron Kimball and Dan Grossman. Software transactions meet first-class continuations. In *8th Annual Workshop on Scheme and Functional Programming*, 2007.

Xavier Leroy. The Objective Caml system release 3.10, Event module, 2007. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Event.html.

Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *35th ACM Symposium on Principles of Programming Languages*, 2008.

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

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

John Reppy and Yingqi Xiao. Specialization of CML message-passing primitives. In *34th ACM Symposium on Principles of Programming Languages*, 2007.

John Reppy and Yinqi Xiao. Toward a parallel implementation of Concurrent ML. In *Workshop on Declarative Aspects of Multicore Programming*, 2008.

Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *10th ACM International Conference on Functional Programming*, 2005.

George Russell. Events in Haskell, and how to implement them. In *6th ACM International Conference on Functional Programming*, 2001.

Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

Lukasz Ziarek and Suresh Jagannathan. Memoizing multi-threaded transactions. In *Workshop on Declarative Aspects of Multicore Programming*, 2008.

Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: A modular checkpointing abstraction for concurrent functional programs. In *11th ACM International Conference on Functional Programming*, 2006.