Composable Asynchronous Events

Lukasz Ziarek, KC Sivaramakrishnan, Suresh Jagannathan
Purdue University
{lziarek, chandras, suresh}@cs.purdue.edu

Abstract

Although asynchronous communication is an important feature of many concurrent systems, building *composable* abstractions that leverage asynchrony is challenging. This is because an asynchronous operation necessarily involves two distinct threads of control – the thread that initiates the operation, and the thread that discharges it. Existing attempts to marry composability with asynchrony either entail sacrificing performance (by limiting the degree of asynchrony permitted), or modularity (by forcing natural abstraction boundaries to be broken).

In this paper, we present the design and rationale for *asynchronous events*, an abstraction that enables composable construction of complex asynchronous protocols without sacrificing the benefits of abstraction or performance. Asynchronous events are realized in the context of Concurrent ML's first-class event abstraction [16]. We discuss the definition of a number of useful asynchronous abstractions that can be built on top of asynchronous events (e.g., composable callbacks) and provide a detailed case study of how asynchronous events can be used to substantially improve the modularity and performance of an I/O-intensive highly concurrent server application.

Categories and Subject Descriptors: D.1.1 [Programming Techniques] Applicative (functional) programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications - Concurrent, distributed, and parallel languages, ML; D.3.3 [*Programming Languages*]: Language Constructs and Features - Concurrent programming structures

General Terms: Design, Languages, Theory

Keywords: Asynchrony, Concurrent ML, First-Class Events, Composability, Message-Passing

1. Introduction

Software complexity is typically managed using programming abstractions that encapsulate behaviors within modules. A module's internal implementation can be changed without requiring changes to clients that use it, provided that these changes do not entail modifying its signature. For concurrent programs, the specifications that can be described by an interface are often too weak to enforce this kind of strong encapsulation, especially in the presence of communication that spans abstraction boundaries. Consequently, changing the implementation of a concurrency abstraction by adding, modifying, or removing behaviors often requires pervasive change to the users of the abstraction. Modularity is thus compromised.

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06…$5.00

This is particularly true for asynchronous behavior generated internally within a module. An asynchronous operation initiates two temporally distinct sets of actions, neither of which are exposed in its enclosing module's interface. The first set defines *post-creation* actions – these are actions that must be executed after an asynchronous operation has been initiated, without taking into account whether the effects of the operation have been witnessed by its recipients. For example, a post-creation action of an asynchronous send on a channel might initiate another operation on that same channel; the second action should take place with the guarantee that the first has already deposited its data on the channel. The second are *post-consumption* actions – these define actions that must be executed only after the effect of an asynchronous operation has been witnessed. For example, a post-consumption action might be a callback that is triggered when the client retrieves data from a channel sent asynchronously.

In this paper, we describe how to build and maintain asynchronous concurrency abstractions that give programmers the ability to express *composable* and *extensible* asynchronous protocols. By weaving protocols through the use of post-creation and post-consumption computations, we achieve composability without sacrificing modularity, enabling reasoning about loosely coupled communication partners that span logical abstraction boundaries. To do so, we enable the construction of signatures and interfaces that specify asynchronous behavior via abstract post-creation and post-consumption actions. We present our extensions in the context of the Concurrent ML's (CML) first-class communication events [16]. Just as CML's synchronous events provide a solution to composable, synchronous message-passing that could not be easily accommodated using λ-abstraction and application, the asynchronous events defined here offer a solution for composable asynchronous message-passing not easily expressible using synchronous communication and explicit threading abstractions. There are three overarching goals of our design:

1. Asynchronous combinators should permit uniform composition of pre/post-creation and consumption actions. This means that protocols should be allowed to extend the behavior of an asynchronous action both with respect to the computation performed before and after it is created and consumed.
2. Asynchronous actions should provide sensible visibility and ordering guarantees. A post-creation computation should execute with the guarantee that the asynchronous action it follows has been created, even if that action has not been fully discharged. The effects of consumed asynchronous actions should be consistent with the order in which they were created.

```
spawn     : (unit -> 'a) -> threadID
sendEvt   : 'a chan * 'a -> unit Event
recvEvt   : 'a chan -> 'a Event
alwaysEvt : 'a -> 'a Event
never     : 'a Event
sync      : 'a Event -> 'a
wrap      : 'a Event * ('a -> 'b) -> 'b Event
guard     : (unit -> 'a Event) -> 'a Event
choose    : 'a Event list -> 'a Event
```

Figure 1: CML event operators.

3. Communication protocols should be agnostic with respect to the kinds of actions they handle. Thus, both synchronous and asynchronous actions should be permitted to operate over the same set of abstractions (e.g., communication channels).

Contributions.

1. We present a comprehensive design for asynchronous events, and describe a family of combinators analogous to their synchronous variants available in CML. To the best of our knowledge, this is the first treatment to consider the meaningful integration of composable CML-style event abstractions with asynchronous functionality.
2. We provide implementations of useful asynchronous abstractions such as callbacks and mailboxes (buffered channels), along with a number of case studies extracted from realistic concurrent applications (e.g., a concurrent I/O library, concurrent file processing, etc.). Our abstractions operate over ordinary CML channels, enabling interoperability between synchronous and asynchronous protocols.
3. We discuss an implementation of asynchronous events that has been incorporated into Multi-MLton, a parallel extension of MLton [14], a whole-program optimizing compiler for Standard ML, and present a detailed case study that shows how asynchronous events can help improve the expression and performance of highly concurrent server application.

The paper is organized as follows. Sec. 2 provides a brief introduction to Concurrent ML. Additional motivation for asynchronous events is given in Sec. 3. Sec. 4 describes the asynchronous combinators and abstractions we support. Sec. 5 gives a formal operational semantics for asynchronous events. A detailed case study is presented in Sec. 6. Related work and conclusions are provided in Sec. 7 and Sec. 8.

2. Background: Concurrent ML

Context: We explore the design of asynchronous events in the context of Concurrent ML [16] (CML). CML is a concurrent extension of Standard ML that utilizes synchronous message passing to enable the construction of synchronous communication protocols. Threads perform `send` and `recv` operations on typed channels; these operations block until a matching action on the same channel is performed by another thread.

CML also provides first-class synchronous *events* that abstract synchronous message-passing operations. An event value of type `'a event` when synchronized on yields a value of type `'a`. An event value represents a potential computation, with latent effect until a thread synchronizes upon it by calling `sync`. The following equivalences thus therefore hold: `send(c, v)` $\equiv$ `sync(sendEvt(c,v))` and `recv(c)` $\equiv$ `sync(recvEvt(c))`.

Notably, thread creation is *not* encoded as an event – the thread `spawn` primitive simply takes a thunk to evaluate as a separate thread, and returns a thread identifier that allows access to the newly created thread's state.

Besides `sendEvt` and `recvEvt`, there are other base events provided by CML. The `never` event, as its name suggests, is never available for synchronization; in contrast, `alwaysEvt` is always available for synchronization. These events are typically generated based on the (un)satisfiability of conditions or invariants that can be subsequently used to influence the behavior of more complex events built from the event combinators described below.

Much of CML's expressive power derives from event combinators that construct complex event values from other events. We list some of these combinators in Fig. 1. The expression `wrap (ev, f)` creates an event that, when synchronized, applies the result of synchronizing on event `ev` to function `f`. Conversely, `guard(f)` creates an event that, when synchronized, evaluates `f()` to yield event `ev` and then synchronizes on `ev`. The `choose` event combinator takes a list of events and constructs an event value that represents the non-deterministic choice of the events in the list; for example, `sync(choose[recvEvt(a),sendEvt(b,v)])` will either receive a unit value from channel `a`, or send value `v` on channel `b`. Selective communication provided by choice motivates the need for first-class events. We cannot, for example, simply build complex event combinators using function abstraction and composition because function closures do not allow inspection of the computations they encapsulates, a necessary requirement for combinators like `choice`.

3. Motivation

Given CML synchronous events, we can envision a simple way to express asynchrony in terms of lightweight threads that encapsulate synchronous operations:

```
fun asyncEvt (evt) = wrap(alwaysEvt(),
                          fn() => spawn(sync(evt)))
```

Evaluating `asyncEvt(evt)` yields an event that, when synchronized, spawns a new thread to perform the actions defined by `evt`. (The use of `alwaysEvt` ensures that the event is always available for synchronization.) Thus,

```
fun asend(c,v) = sync(asyncEvt(sendEvt(c,v)))
```

defines a function that asynchronously sends value `v` on channel `c`, when applied. The use of explicit user-defined threads to express asynchrony in this way, however, fails to provide sensible ordering guarantees. For example, in the expression:

```
let val evt1 = asend(c,v)
    val evt2 = asend(c,v')
in recvEvt(c) end
```

the value returned when this expression is synchronized may be either `v` or `v'` (depending on the implementation of the underlying thread scheduler), even though `evt1` is synchronized before `evt2`.

Besides not preserving desired ordering, there are more fundamental issues at play here. Suppose we wish to add post-creation and post-consumption actions to an asynchronous event; post-creation actions are to be performed once the thread is spawned, and post-consumptions actions must be performed once the communication action is complete. We could modify the definition of `asyncEvt` so that post-creation and post-consumption actions are supplied when the event is constructed:

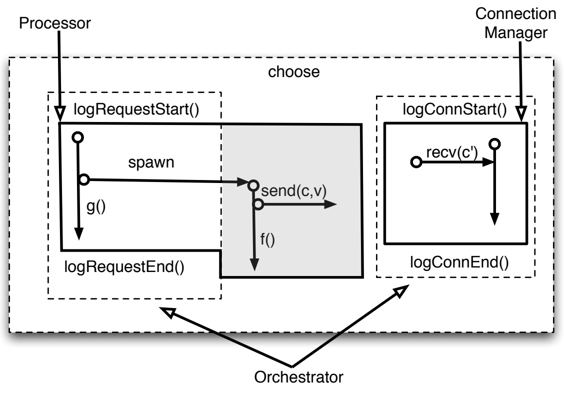

Figure 2: The figure shows the events built up by the three modules that comprise our abstract server. The events created by the processor and connection manager, depicted with thick squares, are opaque to the orchestrator and cannot be deconstructed. Notably, this means the orchestrator does not have the ability to modify or extend the behavior of the thread created by `processorEvt` (shown in grey) using only synchronous event combinators.

```
fun asyncEvt(f,g,evt) =
     wrap(alwaysEvt(),
          fn() => (spawn(fn() => sync(wrap(evt, f))); g()))
```

Given this definition, an asynchronous send event might be defined to perform a post-consumption action, here `f` (e.g., an action that waits for an acknowledgment indicating the sent data was received with no errors). Similarly, the event may be equipped with a post-creation action, here `g` (e.g., an operation that performs local clean-up or finalization).

To illustrate the composability and modularity issues that arise in using `asyncEvt` defined in this way, consider the definition of a simple resource manager (like a web-server) that consists of three modules: (1) a *connection manager* that listens for and accepts new connections, (2) a *processor* that processes and sends data based on input requests, and (3) an *orchestrator* that enables different kinds of interactions and protocols among clients, the processor, and the connection manager. If each established connection represents an independent group of actions (e.g., requests), we can leverage asynchrony to allow multiple requests to be serviced concurrently. We discuss this design in more detail in Sec. 6.

One simple definition of an orchestrator is a procedure that chooses between the communication event defined by the processor (call it `processorEvt`) and the connection manager (call it `managerEvt`). Suppose `processorEvt` internally leveraged our definition of `asyncEvt` to perform a send asynchronously and then executed some post-creation action g and post-consumption action `f`. Even if the details of the communication protocol used by the connection manager and processor were visible to the orchestrator, the orchestrator would not be able to change their internal functionality since they are encapsulated within events, which abstract the computation actions and effects they perform. Nonetheless, the orchestrator is still free to *augment* these protocols to yield additional behavior. For example, the following definition:

```
fun orchestrate(processorEvt, managerEvt) =
  sync(choose([processorEvt, managerEvt]))
```

uses choice to non-deterministically select whichever event is available to be synchronized against. This definition allows the execution of the `managerEvt` if a new connection is available, or the `processorEvt` if the current input request has been satisfied, or a non-deterministic choice among the two if both can execute.

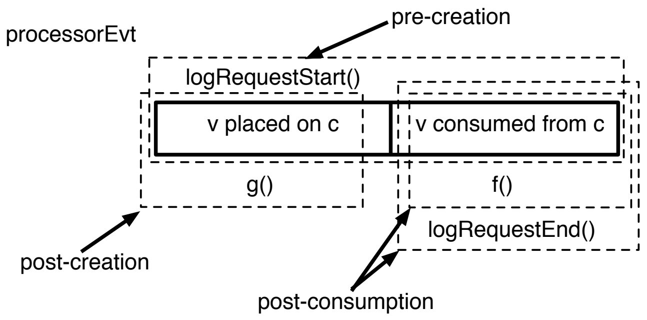

Figure 3: The figure shows how asynchronous events can alleviate the problems illustrated in Fig. 2. By making creation and consumption actions explicit in `processorEvt`'s definition, we are able to specify `logRequestEnd` as a post consumption action correctly extending the protocol in the orchestrator.

Unfortunately, composing more expressive asynchronous protocols in this way is difficult. To see why, consider a further modification to the orchestrator that incorporates logging information. We can wrap our processor event with a function that logs requests and our connection manager with one that logs connection details. The `guard` event combinator can be used to specify pre-synchronization actions. In this example, these actions would be logging functions that record the start of a connection or request. [1]

```
fun orchestrate(processorEvt, managerEvt) =
  sync(choose([guard(fn () => logRequestStart();
                        wrap(processorEvt, logRequestEnd)),
               guard(fn () => logConnStart();
                        wrap(managerEvt, logConnEnd))]))
```

This code does not provide the functionality we desire, however. Since the processor handles its internal protocols asynchronously, wrapping *logRequestEnd* around `processorEvt` specifies an action that will occur in the main thread of control after the execution of `g`, the post-creation action defined when the event was created (see Fig. 2). However, the request should only be completed after the post-consumption action `f`, which is executed by the thread created internally by the event. Since this thread is *hidden* by the event, there is no way to extend it. Moreover, there is no guarantee that the request has been successfully serviced even after `g` completes. We could recover composability by either (a) not spawning an internal thread in `asyncEvt`, (b) weaving a protocol that required `g` to *wait* for the completion of `f`, effectively yielding synchronous behavior, or (c) modifying `procesorEvt` and `managerEvt` to handle the post-consumption action explicitly. The first two approaches force `f` and `g` to be executed prior to *logRequestEnd*, but a synchronous solution of this kind would not allow multiple requests to be processed concurrently; approach (c) retains asynchrony, but at the cost of modularity, requiring server functionality to be modified anytime clients wish to express additional behaviors.

The heart of the problem is a dichotomy in language abstractions; asynchrony is fundamentally expressed using distinct threads of control, yet composablity is achieved through event abstractions that are thread-unaware. The result is that CML events cannot be directly applied to build post-consumption actions for realizing composable asynchronous communication protocols.

Fig. 3 diagrammatically shows how extensible post-creation and post-consumption actions can be leveraged to achieve our desired

[1] In CML, we could also use the `withNack` combinator to avoid firing both the `logRequestStart` and `logConnStart` during the choice [16].

behavior. In Section 4.1, we show a modified implementation of the orchestrator using the asynchronous event primitives and combinators we define in the following section that captures the behavior shown in the figure.

Putting it All Together. Although synchronous first-class events alleviate the complexity of reasoning about arbitrary thread interleavings, and enable composable synchronous communication protocols, using threads to encode asynchrony unfortunately re-introduces these complexities. Our design introduces first-class *asynchronous* events with the following properties to alleviate this drawback: *(*i) they are extensible both with respect to pre- and post-creation as well as pre- and post-consumption actions; *(*ii) they can operate over the same channels that synchronous events operate over, allowing both kinds of events to seamlessly co-exist; and, *(*iii) their visibility, ordering, and semantics is independent of the underlying runtime and scheduling infrastructure.

4. Asynchronous Events

In order to provide primitives that adhere to the desired properties outlined above, we extend CML with the following two base events: `aSendEvt` and `aRecvEvt`, to create an asynchronous send event and an asynchronous receive event, resp. The differences in their type signature from their synchronous counterparts reflect the split in the creation and consumption of the communication action they define:

```
 sendEvt : 'a chan * 'a -> unit Event
aSendEvt : 'a chan * 'a -> (unit, unit) AEvent

 recvEvt : 'a chan -> 'a Event
aRecvEvt : 'a chan -> (unit, 'a) AEvent
```

An `AEvent` value is parametrized with respect to the type of the event's post-creation and post-consumption actions. In the case of `aSendEvt`, both actions are of type `unit`: when synchronized on, the event immediately returns a `unit` value and places its `'a` argument value on the supplied channel. The post-consumption action also yields `unit`. When synchronized on, an `aRecvEvt` returns `unit`; the type of its post-consumption action is `'a` reflecting the type of value read from the channel when it is paired with a send.

As an example, consider Fig 3, where the `processorEvt` event upon synchronization initiates an asynchronous action to place the value `v` on channel `c` and then executes `g`. When the send is paired with a receive, `f` is first executed, and then *logRequestEnd*. In this case, the type of the computation would be:

(*return-type-of* `g`, *return-type-of* `logRequestEnd) AEvent`

The semantics of both asynchronous send and receive guarantees that successive communication operations performed by the same thread get witnessed in the order in which they were issued. In this respect, an asynchronous send event shares functionality with a typical non-blocking send of the kind found in languages like Erlang [1] or libraries like MPI. However, an asynchronous receive does not exhibit the same behavior as a typical non-blocking receive. Indeed, CML already provides polling methods that can be used to implement polling loops found in most non-blocking receive implementations. The primary difference between the two is that an asynchronous receive places itself on the channel at the point where it is synchronized (regardless of the availability of a matching send), while a non-blocking receive typically only queries for the existence of a value to match against, but does not alter the underlying channel structure. By actually depositing itself on the channel, an asynchronous receive thus provides a convenient mechanism to implement *ordered* asynchronous buffers or streams – successive asynchronous receives are guaranteed to receive data from a matching send in the order in which they were synchronized.

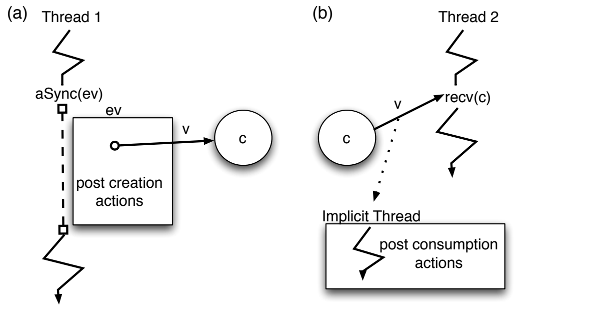

Figure 4: The figure shows a complex asynchronous event `ev`, built from a base `aSendEvt`, being executed by Thread 1. When the event is synchronized via. `aSync`, the value `v` is placed on channel `c` and post-creation actions are executed (see (a)). Afterwards, control returns to Thread 1. When Thread 2 consumes the value `v` from channel `c`, an implicit thread of control is created to execute any post-consumption actions (see (b)).

In addition to these new base events, we also introduce a new synchronization primitive: `aSync`, to synchronize asynchronous events. The `aSync` operation fires the computation encapsulated by the asynchronous event of type `('a, 'b) AEvent` and returns a value of type `'a`, corresponding to the return type of the event's post-*creation* action (see Fig. 4).

```
 sync     : 'a Event -> 'a
aSync     : ('a, 'b) AEvent -> 'a
```

Unlike their synchronous variants, asynchronous events do *not* block if no matching communication is present. For example, executing an asynchronous send event on an empty channel places the value being sent on the channel and then returns control to the executing thread (see Fig. 4(a)). In order to allow this non-blocking behavior, an *implicit* thread of control is created for the asynchronous event when the event is paired, or *consumed* as shown in Fig. 4(b). If a receiver is present on the channel, the asynchronous send event behaves similarly to a synchronous event; it passes the value to the receiver. However, it still creates a new implicit thread of control if there are any post-consumption actions to be executed.

Similarly, the synchronization of an asynchronous receive event does not yield the value received (see Fig. 5); instead, it simply enqueues the receiving action on the channel. Therefore, the thread which synchronizes on an asynchronous receive always gets the value `unit`, even if a matching send exists. The actual value consumed by the asynchronous receive can be passed back to the thread which synchronized on the event through the use of combinators that process post-consumption actions. This is particularly well suited to encode reactive programming idioms: the post-consumption actions encapsulate a reactive computation.

To illustrate the differences between synchronous and asynchronous primitive events, consider the two functions `f` and `af` shown below:

```
1. fun  f () =
2.     (spawn (fn () => sync (sendEvt(c, v)));
3.      sync (sendEvt(c, v'));
4.      sync (recvEvt(c)))

5. fun af () =
6.     (spawn (fn () => sync (sendEvt(c, v)));
7.      aSync (aSendEvt(c, v'));
8.      sync (recvEvt(c)))
```

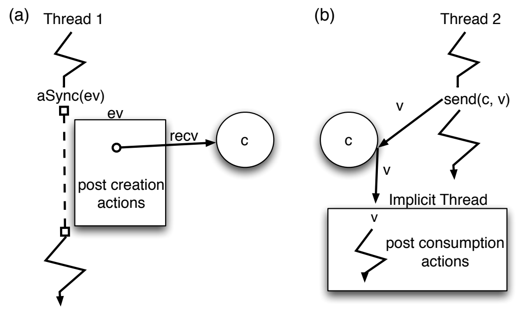

Figure 5: The figure shows a complex asynchronous event `ev`, built from a base `aRecvEvt`, being executed by Thread 1. When the event is synchronized via `aSync`, the receive action is placed on channel `c` and post-creation actions are executed (see (a)). Afterwards, control returns to Thread 1. When Thread 2 sends the value `v` to channel `c`, an implicit thread of control is created to execute any post-consumption actions passing `v` as the argument (see (b)).

The function `f`, if executed in a system with no other threads will always block because there is no recipient available for the send of `v'` on channel `c`. On the other hand, suppose there was another thread willing to accept communication on channel `c`. In this case, the only possible value that `f` could receive from `c` is `v`. This occurs because the receive will only happen after the value `v'` is consumed from the channel. Notice that if the spawned thread enqueues `v` on the channel before `v'`, the function `f` will block even if another thread is willing to receive a value from the channel, since a function cannot synchronize with itself. The function `af` will never block. The receive may see either the value `v` or `v'` since the asynchronous send event *only* asserts that the value `v'` has been placed on the channel and *not* that it has been consumed. If we swapped lines 6 and 7, the receive operation on line 8 is guaranteed to read `v'`. While asynchronous events do not block, they still enforce *ordering* constraints that reflect the order in which they were synchronized.

4.1 Combinators

In CML, the `wrap` combinator allows for the specification of a post-synchronization action. Once the event is completed the function wrapping the event is evaluated. For asynchronous events, this means the wrapped function is executed after the action the event encodes is *placed* on the channel and not necessarily after that action is consumed.

```
sWrap : ('a, 'b) AEvent * ('a -> 'c) ->  ('c, 'b) AEvent
aWrap : ('a, 'b) AEvent * ('b -> 'c) ->  ('a, 'c) AEvent
```

To allow for the specification of both post-creation and post-consumption actions for asynchronous events, we introduce two new combinators: `sWrap` and `aWrap`. `sWrap` is used to specify post-creation actions. The combinator `aWrap`, on the other hand, is used to express post-consumption actions. We can apply `sWrap` and `aWrap` to an asynchronous event in any order.

```
sWrap(aWrap(e, f) g) ≡ aWrap(sWrap(e, g), f)
```

We can use `sWrap` and `aWrap` to encode a composable variant of `asyncEvt` (presented in the motivation) which is also parameterized by a channel `c` and value `v`. We create a base asynchronous send event to send `v` on `c` and use `sWrap` and `aWrap` to specify `g` and `f` as a post-creation action and a post-consumption action, resp.:

```
fun asyncEvt(f,g,c,v) = aWrap(sWrap(aSendEvt(c,v),g), f)
```

Since post-creation actions have been studied in CML extensively (they act as post-synchronization actions in a synchronous context), we focus our discussion on `aWrap` and the specification of post-consumption actions. Consider the following program fragment:

```
fun f() =
  let val c_local = channel()
  in aSync (aWrap(aSendEvt(c, v),fn () => send(c_local, ())));
     g();
     recv(c_local);
     h()
  end
```

The function `f` first allocates a local channel c_{local} and then executes an asynchronous send `aWrap`-ed with a function that sends on the local channel. The function `f` then proceeds to execute functions `g` and `h` with a receive on the local channel between the two function calls. We use the `aWrap` primitive to encode a simple *barrier* based on the consumption of `v`. We are guaranteed that `h` executes in a context in which `v` has been consumed. The function `g`, on the other hand, can make no assumptions on the consumption of `v`. However, `g` is guaranteed that `v` is on the channel. Therefore, if `g` consumes values from `c`, it can witness `v` and, similarly, if it places values on the channel, it is guaranteed that `v` will be consumed *prior* to the values it produces. Of course, `v` could always have been consumed *prior* to `g`'s evaluation. If the same code was written with a synchronous wrap, we would have no guarantee about the consumption of `v`. In fact, the code would block, as the send encapsulated by the wrap would be executed by the *same* thread of control executing `f`. Thus, the asynchronous event implicitly creates a new evaluation context and a new thread of control; the wrapping function is evaluated in this context, not in the context associated with thread that performed the synchronization.

This simple example illustrates the essential ingredients of a basic callback mechanism. The code shown below performs an asynchronous receive and passes the result of the receive to its wrapped function. The value received asynchronously is passed as an argument to `h` by sending on the channel c_{local}.

```
let val c_local = channel()
in aSync (aWrap(aRecvEvt(c), fn x => send(c_local, x)));
   ... h(recv(c_local)) ...
end
```

Although this implementation suffices as a basic callback, it is not particularly abstract and cannot be composed with other asynchronous events. We can create an abstract callback mechanism using both `sWrap` and `aWrap` around an input event.

```
callbackEvt : ('a, 'c) AEvent * ('c -> 'b) ->
              ('b Event, 'c) AEvent


fun callbackEvt(ev, f) =
  let val c_local = channel()
  in sWrap(aWrap(ev,
                fn x => (aSync(aSendEvt(c_local, x)); x)),
           fn _ => wrap(recvEvt(c_local), f))
  end
```

If `ev` contains post-creation actions when the callback event is synchronized on, they are executed, followed by execution of the `sWrap` as shown in Fig. 6(a). The event returned by the `sWrap` (call it `ev'`), which when synchronized on will first receive a value on the local channel (c_{local}) and then apply the function `f` to this value. Synchronizing on this event (which need not happen at the point where the callback event itself is synchronized) will subsequently block until the event `ev` is discharged. Once `ev` completes, its post-consumption actions are executed in a new thread of control since `ev` is asynchronous (see Fig. 6(b)). The body of

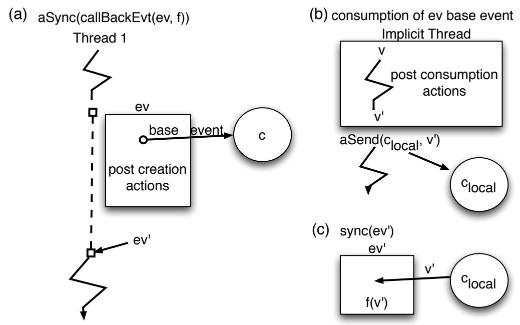

Figure 6: The figure shows a callback event constructed from a complex asynchronous event `ev` and a callback function `f`. When the callback event is synchronized via `aSync`, the action associated with the event `ev` is placed on channel `c` and post-creation actions are executed. A new event `ev'` is created and passed to Thread 1 (see (a)). An implicit thread of control is created after the base event of `ev` is consumed. Post-consumption actions are executed passing `v`, the result of consuming the base event for `ev`, as an argument (see (b)). The result of the post-consumption actions, `v'` is sent on c_{local}. When `ev'` is synchronized upon, `f` is called with `v'` (see (c)).

the `aWrap`-ed function simply sends the result of synchronizing on `ev` (call it `v'`) on c_{local} and then passes the value `v'` to any further post-consumption actions. This is done asynchronously because the complex event returned by `callbackEvt` can be further extended with additional post consumption actions. Those actions should not be blocked if there is no thread willing to synchronize on `ev'`. Thus, synchronizing on a callback event executes the base event associated with `ev` *and* creates a new event as a post-creation action, which when synchronized on, executes the callback function synchronously.

We can think of the difference between a callback and an `aWrap` of an asynchronous event in terms of the thread of control which executes them. Both specify a *post*-consumption action for the asynchronous event, but the callback, when synchronized upon, is executed potentially by an *arbitrary thread* whereas the `aWrap` is *always* executed in the implicit thread created when the asynchronous event is consumed. Another difference is that the callback can be *postponed* and only executes when two conditions are satisfied: *(*i) the asynchronous event has completed and *(*ii) the callback is synchronized on. An `aWrap` returns once it has been synchronized on, and does not need to wait for other asynchronous events or post-consumption actions it encapsulates to complete.

A guard of an asynchronous event behaves much the same as a guard of a synchronous event does; it specifies pre-synchronization actions (i.e. pre-creation computation):

```
aGuard : (unit -> ('a, 'b) AEvent) -> ('a, 'b) AEvent
```

To see how we might use asynchronous guards, notice that our definition of `callbackEvt` has the drawback that it allocates a new local channel regardless of whether or not the event is ever synchronized upon. The code below uses an `aGuard` to specify the allocation of the local channel only when the event is synchronized on:

```
fun callbackEvt(ev, f) =
  aGuard(fn () =>
    let val c_local = channel()
    in sWrap(aWrap(ev,
                   fn x => (aSync(aSendEvt(c_local, x));x)),
             fn _ => wrap(recvEvt(c_local), f))
    end)
```

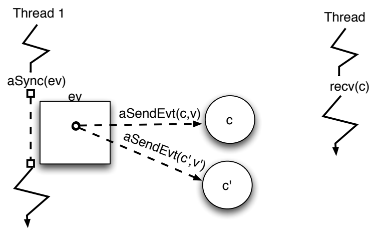

Figure 7: The figure shows Thread 1 synchronizing on a complex asynchronous event `ev`, built from a choice between two base asynchronous send events; one sending `v` on channel `c` and the other `v'` on `c'`. Thread 2 is willing to receive from channel `c`.

One of the most powerful combinators provided by CML is a non-deterministic choice over events. The combinator `choose` picks an *active* event from a list of events. If no events are active, it waits until one becomes active. An active event is an event which is available for synchronization. We define an asynchronous version of the choice combinator, `aChoose`, that operates over asynchronous events. Since asynchronous events are non-blocking, all events in the list are considered *active*. Therefore, the asynchronous choice always non-deterministically chooses from the list of available asynchronous events. We also provide a synchronous version of the asynchronous choice, `sChoose`, which blocks until one of the asynchronous base events has been consumed. Post-creation actions are not executed until the choice has been made. [2]

```
 choose : 'a Event list -> 'a Event
aChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent
sChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent
```

To illustrate the difference between `aChoose` and `sChoose`, consider a complex event `ev` defined as follows:

```
val ev = aChoose[aSendEvt(c, v), aSendEvt(c',v')]
```

If there exists a thread only willing to receive from channel `c`, `aChoose` will, with equal probability, execute the asynchronous send on `c` and `c'` (see Fig. 7). However, if we redefined `ev` to utilize `sChoose` instead, the behavior of the choice changes:

```
val ev = sChoose[aSendEvt(c, v), aSendEvt(c',v')]
```

Since `sChoose` blocks until one of the base asynchronous events is satisfiable, if there is only a thread willing to accept communication on `c` (see Fig. 7), the choice will only select the event encoding the asynchronous send on `c`.

We have thus far provided a mechanism to choose between sets of synchronous events and sets of asynchronous events. However, we would like to allow programmers to choose between both synchronous and asynchronous events. Currently, their different type structure would prevent such a formulation. Notice, however, that an asynchronous event with type `('a, 'b) AEvent` and a synchronous event with type `'a Event` both yield `'a` in the thread which synchronizes on them. Therefore, it is sensible to allow choice to operate over *both* asynchronous and synchronous events provided the type of the asynchronous event's post-creation action is the same as the type encapsulated by the synchronous event. To facilitate this interoperability, we provide combinators to transform asynchronous event types to synchronous event types and *vice-versa*:

```
aTrans : ('a, 'b) AEvent -> 'a Event
```

[2] This behavior is equivalent to a scheduler not executing the thread which created the asynchronous action until it has been consumed.

```
sTrans : 'a Event -> (unit, 'a) AEvent
```

The `aTrans` combinator takes an asynchronous event and creates a synchronous version by *dropping* the asynchronous portion of the event from the type (i.e. encapsulating it). As a result, we can no longer specify post-consumption actions for the event. However, we can still apply `wrap` to specify post-creation actions to the resulting synchronous portion exposed by the `'a Event`. Asynchronous events that have been transformed and are part of a larger `choose` event are only selected if their base event is satisfiable. Therefore, the following equivalence holds for two asynchronous events, `aEvt1` and `aEvt2`:

```
choose[aTrans(aEvt1), aTrans(aEvt2)] ≡
aChoose[aEvt1, aEvt2]
```

The `sTrans` combinator takes a synchronous event and changes it into an asynchronous event with no post-creation actions. The wrapped computation of the original event occurs now as a post-consumption action. We can encode an asynchronous version of `alwaysEvt` from its synchronous counterpart. Similarly, we can encode an asynchronous variant of `never`.

```
aAlwaysEvt : 'a -> (unit, 'a) AEvent
aNever  : (unit, 'a) AEvent
aAlwaysEvt(v) =  sTrans alwaysEvt(v)
aNever = sTrans never
```

Orchestrator revisited: Armed with asynchronous events and the combinators discussed above, we can now implement a composable orchestrator module from our simple abstract server example given in Sec. 3. We use `aGuard` to specify pre-creation actions and `aWrap` for asynchronous post-consumption actions. If `managerEvt` defines a synchronous protocol (since it merely listens for and accepts new connects), and `processorEvt` defines an asynchronous one (since it can process and communicate data concurrently with other ongoing requests), we can use `aTrans` to hide its post-consumption actions from the orchestrator. This allows us to freely choose between the asynchronous `processorEvt` and the synchronous `managerEvt`.

```
fun orchestrate(processorEvt, managerEvt) =
 sync(choose([aTrans
              aGuard(fn () => logRequestStart();
                     aWrap(processorEvt, logRequestEnd)),
              guard(fn () => logConnStart();
                    wrap(managerEvt, logConnEnd))]))
```

Mailboxes and Multicast: Using asynchronous events we can encode other CML structures such as mailboxes (i.e., buffered channels with asynchronous send and synchronous receive semantics) and multicasts channels, reducing code size and complexity. Asynchronous events provide the components from which a mailbox structure can be defined, allowing the construction of mailboxes from regular CML channels (a facility not available in CML), and providing a mechanism to define *asynchronous* send events on the mailbox trivially using the base asynchronous send event. Having an asynchronous send event operation defined for mailboxes allows for their use in selective communication. Additionally, asynchronous events now provide the ability for programmers to specify post-creation *and* post-consumption actions. Using asynchronous events, we reduced the original CML mailbox implementation from 240 LOC to 70 LOC, with a corresponding 52% improvement in performance on synthetic stress tests exercising various producer/-consumer configurations. Similarly, we were able to express multicast channels in 60 LOC, compared to 87 LOC in CML, with a roughly 19% improvement in performance.

5. Semantics

Our semantics (see Fig. 8) is defined in terms of a core call-by-value functional language with threading and communication primitives. Communication between threads is achieved using synchronous channels and events. Our language extends a synchronous-event core language with asynchronous constructs. For perspicuity, the language omits many useful event combinators such as `choose` (and its variants); a semantics formalizing the full set of combinators discussed in this paper is available in an accompanying technical report [19].

In our syntax, v ranges over values, p over primitive event constructors, and e over expressions. Besides abstractions, a value can be a message identifier, used to track communication actions, a channel identifier, or an event context. An event context ($\varepsilon[]$) demarcates event expressions that are built from asynchronous events and their combinators [3] that are eventually supplied as an argument to a synchronization action. The rules use function composition $f \circ g \equiv \lambda x.f(g(x))$ to sequence event actions and computations.

The semantics also includes a new expression form, $\{e_1, e_2\}$ to denote asynchronous communication actions; the expression e_1 corresponds to the creation (and post-creation) of an asynchronous event, while e_2 corresponds to the consumption (and post-consumption) of an asynchronous event. For convenience, both synchronous and asynchronous events are expressed in this form. For a synchronous event, e_2 simply corresponds to an uninteresting action. We refer to e_1 as the synchronous portion of the event, the expression which is executed by the current thread, and e_2 as the asynchronous portion of the event, the expression which is executed by a newly created thread (see rule SYNCEVENT).

A program state consists of a set of threads ($\overline{T}$), a communication map (Δ), and a channel map ($\mathcal{C}$). The communication map is used to track the state of an asynchronous action, while the channel map records the state of channels with respect to waiting (blocked) actions. Evaluation is specified via a relation ($\longrightarrow$) that maps one program state to another. Evaluation rules are applied up to commutativity of parallel composition ($\|$).

Encoding Communication: A communication action is split into two message parts: one corresponding to a sender and the other to a receiver. A send message part is, in turn, composed of two conceptual primitive actions: a *send act* ($\texttt{sendAct}(c,v)$) and a *send wait* (`sendWait`):

$$\texttt{sendAct}: (\mathit{ChannelId} \times \mathit{Val}) \rightarrow \mathit{MessageId} \rightarrow \mathit{MessageId}$$
$$\texttt{sendWait}: \mathit{MessageId} \rightarrow \mathit{Val}$$

The *send act* primitive, when applied to a message identifier, places the value (v) on the channel (c), while the *send wait*, when applied to a message identifier, blocks until the value has been consumed off of the channel, returning `unit` when the message has been consumed.

The message identifier m, generated for each base event (see rule `SyncEvent`) is used to correctly pair the "act" and "wait". Similarly, a receive message part is composed of *receive act* ($\texttt{recvAct}(c)$) and a *receive wait* (`recvWait`) primitives:

$$\texttt{recvAct}: \mathit{ChannelId} \rightarrow \mathit{MessageId} \rightarrow \mathit{MessageId}$$
$$\texttt{recvWait}: \mathit{MessageId} \rightarrow \mathit{Val}$$

A *receive wait* behaves as its send counterpart. A *receive act* removes a value from the channel if a matching send action exists;

[3] We describe the necessity of a guarded event context when we introduce the combinators later in this section.

$$\begin{array}{rcl}
e \in \mathit{Exp} & := & v \mid x \mid p\ e \mid e\ e \\
 & \mid & \{e,\ e'\} \mid \texttt{spawn}\ e \mid \texttt{sync}\ e \mid \texttt{ch}() \\
 & \mid & \texttt{sendEvt}(e,e) \mid \texttt{recvEvt}(e) \\
 & \mid & \texttt{aSendEvt}(e,e) \mid \texttt{aRecvEvt}(e) \\
 & \mid & \texttt{aWrap}(e,e) \mid \texttt{sWrap}(e,e) \mid \texttt{aGuard}(e) \\
v \in \mathit{Val} & := & \texttt{unit} \mid c \mid m \mid \lambda x.e \mid \varepsilon[e] \\
p \in \mathit{Prim} & := & \texttt{sendAct}(c,v) \mid \texttt{sendWait} \mid \texttt{recvAct}(c) \mid \texttt{recvWait}
\end{array}$$

$$\begin{array}{rcl}
E & := & \bullet \mid E\ e \mid v\ E \mid p\ E \mid \texttt{sync}\ E \\
 & \mid & \texttt{sendEvt}(E,e) \mid \texttt{sendEvt}(c,E) \\
 & \mid & \texttt{aSendEvt}(E,e) \mid \texttt{aSendEvt}(c,E) \\
 & \mid & \texttt{recvEvt}(E) \mid \texttt{aRecvEvt}(E) \\
 & \mid & \texttt{aWrap}(E,e) \mid \texttt{sWrap}(E,e) \mid \texttt{aWrap}(v,E) \mid \texttt{sWrap}(v,E) \\
 & \mid & \texttt{aGuard}(E)
\end{array}$$

$$\begin{array}{rcll}
m & \in & \mathit{MessageId} & \\
c & \in & \mathit{ChannelId} & \\
\varepsilon[e], \varepsilon[e]^g & \in & \mathit{Event} & \\
\mathcal{A} & \in & \mathit{Action} & := \mathcal{A}_r \mid \mathcal{A}_s \\
\mathcal{A}_r & \in & \mathit{ReceiveAct} & := \mathcal{R}_c^m \\
\mathcal{A}_s & \in & \mathit{SendAct} & := \mathcal{S}_{c,v}^m
\end{array}$$

$$\begin{array}{rcrcl}
T & \in & \mathit{Thread} & := & (\texttt{t}, e) \\
\overline{T} & \in & \mathit{ThreadCollection} & := & \emptyset \mid T \mid T \mid\mid \overline{T} \\
\Delta & \in & \mathit{CommMap} & := & \mathit{MessageId} \rightarrow \mathit{Val} \\
\mathcal{C} & \in & \mathit{ChanMap} & := & \mathit{ChannelId} \rightarrow \overline{\mathit{Action}} \\
\langle \overline{T} \rangle_{\Delta,\mathcal{C}} & \in & \mathit{State} & := & \langle \overline{T}, \mathit{CommMap}, \mathit{ChanMap} \rangle
\end{array}$$

App

$$\frac{}{\begin{array}{c}\langle (\texttt{t}, E[(\lambda x.e)\ v]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \\ \langle (\texttt{t}, E[e[v/x]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

Channel

$$\frac{c\ \mathit{fresh}}{\langle (\texttt{t}, E[\texttt{ch}()]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[c]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}[c \mapsto \emptyset]}}$$

Spawn

$$\frac{\texttt{t}'\ \mathit{fresh}}{\begin{array}{c}\langle (\texttt{t}, E[\texttt{spawn}\ e]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \\ \langle (\texttt{t}', e) \mid\mid (\texttt{t}, E[\texttt{unit}]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

SendEvent

$$\frac{}{\begin{array}{c}\langle (\texttt{t}, E[\texttt{sendEvt}(c,v)]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \\ \langle (\texttt{t}, E[\varepsilon[\{\texttt{sendWait} \circ \texttt{sendAct}(c,v),\ \lambda x.\texttt{unit}\}]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

ASendEvent

$$\frac{}{\begin{array}{c}\langle (\texttt{t}, E[\texttt{aSendEvt}(c,v)]) \mid\mid \overline{T} \rangle_{\Delta} \rightarrow \\ \langle (\texttt{t}, E[\varepsilon[\{\texttt{sendAct}(c,v),\ \texttt{sendWait}\}]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

RecvEvent

$$\frac{}{\begin{array}{c}\langle (\texttt{t}, E[\texttt{recvEvt}(c)]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \\ \langle (\texttt{t}, E[\varepsilon[\{\texttt{recvWait} \circ \texttt{recvAct}(c),\ \lambda x.\texttt{unit}\}]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

ARecvEvent

$$\frac{}{\begin{array}{c}\langle (\texttt{t}, E[\texttt{aRecvEvt}(c)]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \\ \langle (\texttt{t}, E[\varepsilon[\{\texttt{recvAct}(c),\ \texttt{recvWait}\}]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}\end{array}}$$

SyncEvent

$$\frac{m\ \mathit{fresh} \quad \texttt{t}'\ \mathit{fresh}}{\langle (\texttt{t}, E[\texttt{sync}\ \varepsilon[\{e,\ e'\}]]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[e\ m]) \mid\mid (\texttt{t}', e'\ m) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}}$$

Message

$$\frac{}{\Delta, \mathcal{S}_{c,v}^m \Rightarrow \Delta[m \mapsto \texttt{unit}]} \qquad \frac{}{\Delta, \mathcal{R}_c^m, v \Rightarrow \Delta[m \mapsto v]}$$

SendMatch

$$\frac{\begin{array}{c}\mathcal{C}(c) = \mathcal{R}_c^{m'} : \overline{\mathcal{A}_r} \\ \Delta, \mathcal{S}_{c,v}^m \Rightarrow \Delta' \quad \Delta', \mathcal{R}_c^{m'}, v \Rightarrow \Delta''\end{array}}{\langle (\texttt{t}, E[(\texttt{sendAct}(c,v))\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[m]) \mid\mid \overline{T} \rangle_{\Delta'',\mathcal{C}[c \mapsto \overline{\mathcal{A}_r}]}}$$

RecvMatch

$$\frac{\begin{array}{c}\mathcal{C}(c) = \mathcal{S}_{c,v}^{m'} : \overline{\mathcal{A}_s} \\ \Delta, \mathcal{S}_{c,v}^{m'} \Rightarrow \Delta' \quad \Delta', \mathcal{R}_c^{m}, v \Rightarrow \Delta''\end{array}}{\langle (\texttt{t}, E[(\texttt{recvAct}(c))\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[m]) \mid\mid \overline{T} \rangle_{\Delta'',\mathcal{C}[c \mapsto \overline{\mathcal{A}_s}]}}$$

SendBlock

$$\frac{\mathcal{C}(c) = \overline{\mathcal{A}_s} \quad \mathcal{C}' = \mathcal{C}[c \mapsto \overline{\mathcal{A}_s} : \mathcal{S}_{c,v}^m]}{\langle (\texttt{t}, E[(\texttt{sendAct}(c,v))\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}'}}$$

RecvBlock

$$\frac{\mathcal{C}(c) = \overline{\mathcal{A}_r} \quad \mathcal{C}' = \mathcal{C}[c \mapsto \overline{\mathcal{A}_r} : \mathcal{R}_c^m]}{\langle (\texttt{t}, E[(\texttt{recvAct}(c))\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}'}}$$

SendWait

$$\frac{\Delta(m) = \texttt{unit}}{\langle (\texttt{t}, E[\texttt{sendWait}\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[\texttt{unit}]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}}$$

ReceiveWait

$$\frac{\Delta(m) = v}{\langle (\texttt{t}, E[\texttt{recvWait}\ m]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}} \rightarrow \langle (\texttt{t}, E[v]) \mid\mid \overline{T} \rangle_{\Delta,\mathcal{C}}}$$

Figure 8: A core language for asynchronous events.

if none exists, it simply records the *intention* of performing the receive on the channel queue. We can think of computations occurring after an act as post-creation actions and those occurring after a wait as post-consumption actions. Splitting a communication message part into an "act" and a "wait" primitive functions allows for the expression of many types of message passing. For instance, a traditional synchronous send is simply the sequencing of a *send act* followed by a *send wait*: `sendWait` $\circ$ `sendAct`(c, v). This encoding immediately causes the thread executing the operation to block after the value has been deposited on a channel, unless there is a matching *receive act* currently available. A synchronous receive is encoded in much the same manner.

We use the global communication map (Δ) to track act and wait actions for a given message identifier. A message id is created at a synchronization point, ensuring a unique message identifier for each synchronized event. Once a send or receive act occurs, Δ is updated to reflect the value yielded by the act (see Rule MESSAGE) through an auxiliary relation ($\Rightarrow$). When a send act occurs the communication map will hold a binding to `unit` for the corresponding message, but when a receive act occurs the communication map binds the corresponding message to the value received. The values stored in the communication map are passed to the wait actions corresponding to the message (Rules SEND WAIT and RECV WAIT).

Base Events: There are four rules for creating base events, (SENDEVENT) and (RECVEVENT) for synchronous events, and (ASENDEVENT) and (ARECVEVENT) for their asynchronous counterparts. From base act and wait actions, we define asynchronous events ($\varepsilon[\{$`sendAct`(c, v), `sendWait`$\}]$). The first component of an asynchronous event is executed in the thread in which the expression evaluates, and is the target of synchronization (`sync`), while the second component defines the actual asynchronous computation. For asynchronous events we split the act from the wait. Synchronous events can also be encoded using this notation: $\varepsilon[\{$`sendWait` $\circ$ `sendAct`(c, v), $\lambda x.$`unit`$\}]$. In a synchronous event both the act and its corresponding wait occur in the synchronous portion of the event. The base asynchronous portion is simply a lambda that yields a unit value.

Event Evaluation: As mentioned above, events are deconstructed by the `sync` operator in rule (SYNCEVENT). It strips the event context ($\varepsilon[]$), generates a new message identifier for the base event, creates a new thread of control, and triggers the evaluation of the internal expressions. The asynchronous portion of the event is wrapped in a new thread of control and placed in the regular pool of threads. If the event abstraction being synchronized was generated by a base synchronous event, the asynchronous portion is an uninteresting value (e.g. , $\lambda x.$`unit`). The newly created thread, in the case of an asynchronous event, will not be able to be evaluated further as it blocks until the corresponding act for the base event comprising the complex asynchronous event is discharged.

Communication and Ordering: There are four rules for communicating over channels (SENDMATCH, RECVMATCH, SENDBLOCK, and RECVBLOCK. The channel map ($\mathcal{C}$) encodes abstract channel states mapping a channel to a sequence of actions ($\overline{\mathcal{A}}$). This sequence encodes a FIFO queue and provides ordering between actions on the channel. The channel will have either a sequence of send acts ($\mathcal{A}_s$) or receive acts ($\mathcal{A}_r$), but never both at the same time. This is because if there are, for example, send acts enqueued on it, a receive action will immediately match the send, instead of needing to be enqueued and *vice versa* (rules SENDMATCH and RECVMATCH). If a channel already has send acts enqueued on it, any thread wishing to send on the channel will enqueue its act and *vice versa* (rules SENDBLOCK) and (RECVBLOCK). After enqueueing its action, a thread can proceed with its evaluation.

Ordering for asynchronous acts and their post consumption actions as well as blocking of synchronous events is achieved by rules (SENDWAIT) and (RECVWAIT). Both rules block the evaluation of a thread until the corresponding act has been evaluated. In the case of synchronous events, this thread is the one that initiated the act; in the case of an asynchronous event, the thread that creates the act is different from the one that waits on it, and the blocking rules only block the implicitly created thread. For example, the condition $\Delta(m) = unit$ in rule SENDWAIT is established either by rule SENDMATCH, in the case of a synchronous action (created by SENDEVENT), or rules SENDBLOCK and RECVMATCH for an asynchronous one (created by ASENDEVENT).

Combinators: Complex events are built from the combinators described earlier; their definitions are shown in Figure 9. We define two variants of wrap, SWRAP for specifying extensions to the *synchronous* portion of the event and AWRAP for specifying extension to the *asynchronous* portion of the event. In the case of a synchronous event, we have SWRAP extend the event with post-consumption actions as the base event will perform both the *act* and *wait* in the synchronous portion of the event. Similarly, leveraging AWRAP on a synchronous event allows for the specification of general asynchronous actions. If the base event is asynchronous, SWRAP expresses post creation actions and AWRAP post consumption actions.

The specification of the guard combinator is a bit more complex. Since a guard builds an event expression out of a function, that when executed yields an event, the concrete event is only generated at the synchronization point. This occurs because the guard is only executed when synchronized upon. The rule GUARD simply places the function applied to a unit value (the function is really a thunk) in a specialized guarded event context ($\varepsilon[(\lambda x.e)$`unit`$]^g$). The rule SYNC GUARDED EVENT simply strips the guarded event context and synchronizes on the encapsulated expression. This expression, when evaluated, will yield an event. Guarded events cannot be immediately extended with an SWRAP or AWRAP as the expression contained within a guarded event context is a function. Instead, wrapping an event in a guarded context simply moves the wrap expression into the event context.

6. Case Study: A Parallel Web-server

We have implemented asynchronous events in Multi-MLton, an open source, multi-core aware implementation of MLton [14]. Our implementation closely follows the semantics given in Section 5, and comprises roughly 4K LOC wholly written in ML.

`Swerve` [14] is an open-source, third-party, multithreaded web-server wholly written in CML and is roughly 16K lines of CML code. We briefly touch upon three aspects of Swerve's design that were amenable to using asynchronous events, and show how these changes lead to substantial improvement in throughput and performance.

To better understand the utility of asynchronous events, we consider the interactions of four of `Swerve`'s modules: the `Listener`, the `File Processor`, the `Network Processor`, and the `Timeout Manager`. The `Listener` module receives incoming `HTTP` requests and delegates file serving requirements to concurrently executing processing threads. For each new connection, a new listener is spawned; thus, each connection has one main governing entity. The `File Processor` module handles access to the underlying file system. Each file that will be hosted is read by a file processor thread that chunks the file and sends it via message-sWrap

$$\frac{}{\begin{array}{c}\langle(\mathtt{t}, E[\mathtt{sWrap}(\varepsilon[\{e,\ e'\}], \lambda x.e'')]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \\ \langle(\mathtt{t}, E[\varepsilon[\{\lambda x.e'' \circ e,\ e'\}]]) \mid\mid \overline{T}\rangle_{\Delta,C}\end{array}}$$

aWrap

$$\frac{}{\begin{array}{c}\langle(\mathtt{t}, E[\mathtt{aWrap}(\varepsilon[\{e,\ e'\}], \lambda x.e'')]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \\ \langle(\mathtt{t}, E[\varepsilon[\{e,\ \lambda x.e'' \circ e'\}]]) \mid\mid \overline{T}\rangle_{\Delta,C}\end{array}}$$

Guard

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{aGuard}(\lambda x.e)]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \langle(\mathtt{t}, E[\varepsilon[(\lambda x.e)\ \mathtt{unit}]^{g}]) \mid\mid \overline{T}\rangle_{\Delta,C}}$$

Sync Guarded Event

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{sync}\ \varepsilon[e]^{g}]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \langle(\mathtt{t}, E[\mathtt{sync}\ e]) \mid\mid \overline{T}\rangle_{\Delta,C}}$$

sWrap Guarded Event

$$\frac{}{\begin{array}{c}\langle(\mathtt{t}, E[\mathtt{sWrap}(\varepsilon[e]^{g}, \lambda x.e')]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \\ \langle(\mathtt{t}, E[\varepsilon[\mathtt{sWrap}(e, \lambda x.e')]^{g}]) \mid\mid \overline{T}\rangle_{\Delta,C}\end{array}}$$

aWrap Guarded Event

$$\frac{}{\begin{array}{c}\langle(\mathtt{t}, E[\mathtt{aWrap}(\varepsilon[e]^{g}, \lambda x.e')]) \mid\mid \overline{T}\rangle_{\Delta,C} \rightarrow \\ \langle(\mathtt{t}, E[\varepsilon[\mathtt{aWrap}(e, \lambda x.e')]^{g}]) \mid\mid \overline{T}\rangle_{\Delta,C}\end{array}}$$

Figure 9: Combinator extension for a core language for asynchronous events.

passing to the `Network Processor`. The `Network Processor`, like the `File Processor`, handles access to the network. The `File Processor` and `Network Processor` execute in lock-step, requiring the `Network Processor` to have completed sending a chunk before the next one is read from disk. Timeouts are processed by the `Timeout Manager` through the use of timed events.

Lock-step File and Network I/O: Swerve was engineered assuming lock-step file and network I/O. While adequate under low request loads, this design has poor scalability characteristics. This is because (a) file descriptors, a bounded resource, can remain open for potentially long periods of time, as many different requests are multiplexed among a set of compute threads, and (b) for a given request, a file chunk is read only after the network processor has sent the previous chunk. Asynchronous events can be used to alleviate both bottlenecks.

To solve the problem of lockstep transfer of file chunks, we might consider using simple asynchronous sends. However, Swerve was engineered to require the file processor to be responsible for detecting timeouts. If a timeout occurs, the file processor sends a notification to the network processor on the same channel used to send file chunks. Therefore, if asynchrony was used to simply buffer the file chunks, a timeout would not be detected by the network processor until *all* the chunks were processed. Changing the communication structure to send timeout notifications on a separate channel would entail substantial structural modifications to the code base.

The code shown in Fig. 10 is a simplified version of the file processing module modified to use asynchronous events. It uses an arbitrator defined within the file processor to manage the file chunks produced by the `fileReader`. Now, the `fileReader` sends file chunks asynchronously to the arbitrator on the channel `arIn` (line 12) as a post-consumption action. Each such asynchronous send acts as an arbitrator for the next asynchronous send (lines 18-20). The `arbitrator` accepts file chunks from the `fileReader` on this channel and synchronously sends the file chunks to the consumer as long as a timeout has not been detected. This is accomplished by choosing between an `abortEvt` (used by the `Timeout` manager to signal a timeout) and receiving a chunk from file processing loop (lines 13-20). When a timeout is detected, an asynchronous message is sent on channel `arOut` to notify the file processing loop of this fact (line 9); subsequent file processing then stops. This loop synchronously chooses between accepting a timeout notification (line 17), or asynchronously processing the next chunk (lines 11 - 12).

```
datatype Xfr = TIMEOUT | DONE | X of chunk
1. fun fileReader name abortEvt consumer =
2. let
3.  val (arIn, arOut) = (channel(), channel())
4.  fun arbitrator() = sync
5.      (choose [
6.        wrap (recvEvt arIn,
7.              fn chunk => send (consumer, chunk)),
8.        wrap (abortEvt, fn () =>
9.              (aSync(aSendEvt(arOut, ()));
10.             send(consumer, TIMEOUT)))])
11. fun sendChunk(chunk) =
12.        aSync(aWrap(aSendEvt(arIn, X(chunk)),arbitrator))
13. fun loop strm =
14.      case BinIO.read (strm, size)
15.      of SOME chunk => sync
16.              (choose [
17.                recvEvt arOut,
18.                wrap(alwaysEvt,
19.                      fn () => (sendChunk(chunk);
20.                                loop strm))])
21.       | NONE => aSync(aSendEvt(arIn, DONE))
22. in
23. case BinIO.openIt name of
24.      NONE => ()
25.    | SOME strm => (loop strm; BinIO.closeIt strm)
26. end
```

Figure 10: A simplified version of the file processing module in Swerve.

Since asynchronous events operate over regular CML channels, we were able to modify the file processor to utilize asynchrony without having to change any of the other modules or the communication patterns and protocols they expect. Being able to choose between synchronous and asynchronous events in the `fileReader` function also allowed us to create a buffer of file chunks, but stop file processing file if a timeout was detected by the `arbitrator`.

Parallel Packetizing: In CML, channels are often used to implement shared input/output buffered streams. For example, in Swerve, the network processor uses a buffered stream to collect concurrently-processed data chunks generated by the file processor. These chunks are subsequently transformed into packets amenable for transmission back to the client. Asynchronous receives allow parallel processing of these chunks that *automatically* preserves the order in which these chunks were generated. Associated with each element in the buffered stream is a thread that asynchronously waits for the element to be deposited, processes the chunk into a packet, and sends it on a dedicated local channel. This functionality is encapsulated within an event (that leverages an asynchronous receive) that is eventually synchronized by a collator thread which waits for packets to be generated before sending the result back to the client:

```
fun packetEvt(is) =
  aGuard(fn () =>
    let val c = channel()
    in sWrap(aWrap(aRecvEvt(is),
             fn x => send(c, packetize(parse(x))),
       fn () => recvEvt(c)))
    end)
```

When the event returned by `packetEvt` is synchronized, an asynchronous receive event is deposited on the input stream (`is`), and a new event is returned, which, when synchronized in turn, will yield the final packet to be sent on the network.

Given a parameter, `bufferSize` , of how many packets we wish to processes in parallel, we can express the `collate` function as follows:

```
fun collate(bufferSize) =
  let fun createEvents(0, evts) = evts
        | createEvents(x, evts) =
          createEvents(x-1, evts@[sync(packetEvt(is))])
      fun sendPackets([]) = ()
        | sendPackets(e::evts) =
            (networkIO.send(socket,sync(e));
             sendPackets(evts))
  in sendPackets(createEvents(bufferSize, []))
  end
```

The auxiliary function `createEvents` synchronizes on `bufferSize` number of `parseAndPacketEvts` . This results in a list of events which, when synchronized, will yield the final packets. This list of events consists of the synchronous receive events over local channels returned by `parseAndPacketEvts` .

Without asynchronous events, the concurrency afforded by this implementation could be realized by having a collection of explicit packet-izing threads all contending on the stream, waiting for a new element to be deposited. However, because these threads can process the chunks out-of-order, additional metadata such as a sequence number must be provided in the deposited chunks. This requires modifying the module which is responsible for the input stream to embed relevant metadata, as well as augmenting the collator to make sure to stitch things back into correct order using these sequence numbers. Asynchronous receives *implicitly* provide these ordering guarantees, alleviating the burden of weaving this metadata management in the protocol, resulting in cleaner, more modular, code.

Underlying I/O and Logging: To improve scalability and responsiveness, we also implemented a non-blocking I/O library composed of a language-level interface and associated runtime support. The library implements all MLton I/O interfaces, but internally utilizes asynchronous events. The library is structured around callback events as defined in Sec. 4.1 operating over I/O resource servers. Internally, all I/O requests are translated into a potential series of callback events.

Web-servers utilize logging for administrative purposes. For long running servers, logs tend to grow quickly. Some web-servers (like Apache) solve this problem by using a *rolling log*, which automatically opens a new log file after a set time period (usually a day). In Swerve, all logging functions were done asynchronously. Using asynchronous events, we were able to easily change the logging infrastructure to use rolling logs. Post consumption actions were utilized to implement the rolling log functionality, by closing old logs and opening new logs after the appropriate time quantum.

In addition, Swerve's logging infrastructure is tasked with exiting the system if a fatal error is detected. The log notates that the occurrence of the error, flushes the log to disk, and then exits the system. This ensure that the log contains a record of the error prior to the system's exit. Unfortunately, for the modules that utilize logging, this poses additional complexity and breaks modularity. Instead of logging the error at the point which it occurred, the error must be logged *after* the module has performed any clean up actions because of the synchronous communication protocol between the module and the log. Thus, if the module logs any actions during the clean up phase, they will appear in the log *prior* to the error. We can leverage asynchronous callback events to extend the module without changing the communication protocol to the log.

```
let val logEvt = aSendEvt(log, fatalErr)
    val logEvt' = callbackEvt(logEvt,
                    fn () => (Log.flush();
                              System.exit()))
    val exitEvt = aSync(logEvt')
in ( clean up; sync(exitEvt))
end
```

In the code above, `logEvt` corresponds to an event that encapsulates the communication protocol the log expects: a simple asynchronous send on the log's input channel `log`. The event `logEvt'` defines a callback. This event, when synchronized, will execute an asynchronous send to the log and will create a new event that becomes bound to `exitEvt`. When `exitEvt` is synchronized upon, we are guaranteed that the log has received the notification of the fatal error. With this simplification we can also simplify the log by removing checks to see if a logged message corresponds to a fatal error and the exit mechanism; logging and system exit are now no longer conflated.

6.1 Results

To measure the efficiency of our changes in Swerve, we leveraged the server's internal timing and profiling output for per-module accounting. The benchmarks were run on an AMD Opteron 865 server with 8 processors, each containing two symmetric cores, and 32 GB of total memory, with each CPU having its own local memory of 4 GB. The results as well as the changes to the largest modules are summarized in Table 1. Translating the implementation to use asynchronous events leads to a 4.7X performance improvement as well as a 15X reduction in client-side observed latency over the original, with only 103 lines of code changed out of 16KLOC.

Not surprisingly, the results show that asynchronous communication, when carefully applied, can yield substantial performance gains. More significantly, however, is that these gains were achieved with only small changes to the overall structure of the application. These changes were almost always mechanical, often just involving the replacement of a synchronous event combinator with an asynchronous one. No changes were required to module interfaces or the program's overall logical structure.

To put the performance gains in perspective, our modified version of Swerve with asynchronous events has a throughput within 10% of Apache 2.2.15 on workloads that establish up to 1000 concurrent connections and process small/medium files at a total rate of 2000 requests per second. For server performance measurements and workload generation we used httperf – a tool for measuring web-server performance.

Module	LOC	LOC modified	improvement
Listener	1188	11	2.15 X
File Processor	2519	35	19.23 X
Network Processor	2456	25	24.8 X
Timeout Manager	360	15	4.25 X
Swerve	16,000	103	4.7 X

Table 1: Per module performance numbers for Swerve.

7. Related Work

Many functional programming languages such as Erlang [1], JoCaml [10], and F# [18] provide intrinsic support for asynchronous programming. In Erlang, message sends are inherently asynchronous. In JoCaml, complex asynchronous protocols are defined using join patterns [2, 11] that define synchronization protocols over asynchronous and synchronous channels. In F#, asynchronous behavior is defined using asynchronous work flows that permit asynchronous objects to be created and synchronized. Convenient monadic-style `let!` -syntax permits callbacks, represented as continuations, to be created within an asynchronous computation. While these different techniques provide expressive ways to define asynchronous computations, they do not focus on issues of composability (our primary interest in this paper), especially with respect to asynchronous post-consumption actions. There have also been efforts to simplify asynchronous programming in imperative languages [3] by providing new primitives that are amenable to compiler analysis; here again, the primary focus is not on composability or modularity of asynchronous event-based protocols.

Reactive programming [12] is an important programing style often found in systems programming that uses event loops to react to outside events (typically related to I/O). In this context, events do not define abstract communication protocols (as they do in CML), but typically represent I/O actions delivered asynchronously by the underlying operating system. While understanding how reactive events and threads can co-exist is an important one, we believe such efforts are orthogonal to the focus of this work.Indeed we can encode reactive style programming idioms in ACML through the use of asynchronous receive events and/or lightweight servers.

Asynchronous exceptions as discussed in [13] provide abstractions that concurrent applications can use to allow one thread to seamlessly and asynchronously signal another. Kill-safe abstractions [8] provide a related solution to safely terminate a cooperative user-level thread without violating sensible invariants on shared objects. While asynchronous events are a general mechanism for composable and modular asynchronous programming, and thus were not designed specifically with these purposes in mind, we believe they can be used to serve such roles effectively as well, as described in the logging infrastructure example given in Sec. 6.

There have been incarnations of CML in languages and systems other than ML (e.g., Haskell [4, 17], Scheme [8], and MPI [5]). There has also been much recent interest in extending CML with transactional support [6, 7] and other flavors of parallelism [9]. We believe transactional events [6, 7] provide an interesting platform upon which to implement a non-blocking version of `sChoose` that retains the same semantics. Additionally, we expect that previous work on specialization of CML primitives [15] can be applied to improve the performance of asynchronous primitives.

8. Concluding Remarks

This paper presents the design, rationale, and implementation for asynchronous events, a concurrency abstraction that generalizes the behavior of CML-based synchronous events to enable composable construction of asynchronous computations. Our experiments indicate that asynchronous events can seamlessly co-exist with other CML primitives, and can be effectively leveraged to improve performance of realistic highly-concurrent applications.

Acknowledgements. This work is supported by the National Science Foundation under grants CCF-0701832 and CCF-0811631, and a gift from Samsung Corporation.

References

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

[2] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Commun. ACM*, 36(1), 1993.

[3] Prakash Chandrasekaran, Christopher L. Conway, Joseph M. Joy, and Sriram K. Rajamani. Programming asynchronous layers with clarity. In *FSE*, pages 65–74, 2007.

[4] Avik Chaudhuri. A Concurrent Ml Library in Concurrent Haskell. In *ICFP*, pages 269–280, 2009.

[5] Erik Demaine. First-Class Communication in MPI. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, 1996.

[6] Kevin Donnelly and Matthew Fluet. Transactional Events. *The Journal of Functional Programming*, pages 649–706, 2008.

[7] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional Events for ML. In *ICFP*, pages 103–114, 2008.

[8] Matthew Flatt and Robert Bruse Findler. Kill-safe Synchronization Abstractions. In *PLDI*, pages 47–58, 2004.

[9] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-Threaded Parallelism in Manticore. In *ICFP*, pages 119–130, 2008.

[10] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmidt. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *Advanced Functional Programming*, pages 129–158. 2002.

[11] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL*, pages 372–385, 1996.

[12] Peng Li and Steve Zdancewic. Combining Events and Threads for Scalable Network Services, and Evaluation of Monadic, Application-Level Concurrency Primitives. In *PLDI*, pages 189–199, 2007.

[13] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous Exceptions in Haskell. In *PLDI*, pages 274–285, 2001.

[14] MLton. http://www.mlton.org.

[15] John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *POPL*, pages 315–326, 2007.

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

[17] George Russell. Events in Haskell, and How to Implement Them. In *ICFP*, pages 157–168, 2001.

[18] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[19] Lukasz Ziarek, K.C. Sivaramakrishnan, and Suresh Jagannathan. Composable Asynchronous Events. Technical Report TR-11-09, Dept. of Computer Science, Purdue University, 2011.