Concurrent ML: Design, Application and Semantics

John H. Reppy

Department of Computer Science
Cornell University *

1 Introduction

Concurrent ML (**CML**) is a high-level, high-performance language for concurrent programming. It is an extension of Standard ML (**SML**) [MTH90], and is implemented on top of Standard ML of New Jersey (**SML/NJ**) [AM87]. **CML** is a practical language and is being used to build real systems. It demonstrates that we need not sacrifice high-level notation in order to have good performance. **CML** is also a well-defined language. In the tradition of **SML**, it has a formal semantics and its type-soundness has been proven.

Although most research in the area of concurrent language design has been motivated by the desire to improve performance by exploiting multiprocessors, I believe that concurrency is also a useful programming paradigm for certain application domains. For example, interactive systems often have a naturally concurrent structure [CP85, RG86, Pik89, Haa90, GR92]. Another example is distributed systems: most systems for distributed programming provide multi-threading at the node level (e.g., **Isis** [BCJ+90] and **Argus** [LCJS87]). Sequential programs in these application domains often must use complex and artificial control structures to schedule and interleave activities (e.g., event-loops in graphics libraries). They are, in effect, simulating concurrency. These application domains need a high-level concurrent language that provides both efficient sequential execution and efficient concurrent execution: **CML** satisfies this need.

1.1 An Overview of CML

CML is based on the sequential language **SML** [MTH90] and inherits the useful features of **SML**: functions as first-class values, strong static typing, polymorphism, datatypes and pattern matching, lexical scoping, exception handling and a state-of-the-art module facility. An introduction to **SML** can be found elsewhere in this volume [Oph92]; also see [Pau91] or [Har86]. The sequential performance of **CML** benefits from the quality of the **SML/NJ** compiler [AM87]. In addition **CML** has the following properties:

* Author's current affiliation: AT&T Bell Laboratories, Murray Hill NJ 07974, `jhr@research.att.com`.

- **CML** provides a high-level model of concurrency with dynamic creation of threads and typed channels, and *rendezvous* communication. This distributed-memory model fits well with the mostly applicative style of **SML**.
- **CML** is a higher-order concurrent language. Just as **SML** supports functions as first-class values, **CML** supports synchronous operations as first-class values [Rep88, Rep91a, Rep92]. These values, called *events*, provide the tools for building new synchronization abstractions. This is the most significant characteristic of **CML**.
- **CML** provides integrated I/O support. Potentially blocking I/O operations, such as reading from an input stream, are full-fledged synchronous operations. Low-level support is also provided, from which distributed communication abstractions can be constructed.
- **CML** provides automatic reclamation of threads and channels, once they become inaccessible. This permits a technique of speculative communication, which is not possible in other threads packages.
- **CML** uses preemptive scheduling. To guarantee interactive responsiveness, a single thread cannot be allowed to monopolize the processor. Pre-emption insures that a context switch will occur at regular intervals, which allows "off-the-shelf" code to be incorporated in a concurrent thread without destroying interactive responsiveness.
- **CML** is efficient. Thread creation, thread switching and message passing are very efficient (benchmarks results are reported in [Rep92]). Experience with **CML** has shown that it is a viable language for implementing usable interactive systems [GR91].
- **CML** is portable. It is written in **SML** and runs on essentially every system supported by **SML/NJ** (currently four different architectures and many different operating systems).
- **CML** has a formal foundation. Following the tradition of **SML** [MTH90, MT91], a formal semantics has been developed for the concurrency primitives of **CML**.

1.2 Organization of this Paper

This paper is organized into three main parts. The first, consisting of Sections 2 and 3, describes the design and rationale of **CML**. The second part focuses on the practical aspects: Section 4 describes the use of **CML** in real applications, and Section 5 briefly discusses the implementation of **CML**. The third part presents the formal underpinnings of **CML**: Section 6 gives the dynamic semantics a small concurrent language that supports the core **CML** concurrency mechanisms, and then Section 7 gives a static semantics (i.e., a type system) for this language and presents type-soundness results. The paper concludes with a discussion of related work and a summary.

2 Basic Concurrency Primitives

We start with a discussion of the basic concurrency operations provided by **CML**. A running **CML** program consists of a collection of *threads*, which use synchronous message passing on typed channels to communicate and synchronize. In keeping with the flavor of **SML**, both threads and channels are created dynamically (initially, a program consists of a single thread). The signature of the basic thread and channel operations is given in Figure 1. The function `spawn` takes a function as an argument

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

val channel : unit -> '1a chan

val accept  : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit
```

Fig. 1. Basic concurrency primitives

and creates a new thread to evaluate the application of the function to the unit value. Channels are also created dynamically using the function `channel`, which is weakly polymorphic.[2] The functions `accept` and `send` are the synchronous communication operations. When a thread wants to communicate on a channel, it must *rendezvous* with another thread that wants to do a complementary communication on the same channel. **SML**'s lexical scoping is used to share channels between threads, and to hide channels from other threads (note, however, that channels can be passed as messages).

Most **CSP**-style languages (e.g., **occam** [Bur88] and **amber** [Car86]) provide similar rendezvous-style communication. In addition, they provide a mechanism for *selective communication*, which is necessary for threads to communicate with multiple partners. It is possible to use polling to implement selective communication, but to do so is awkward and requires busy waiting. Usually selective communication is provided as a multiplexed I/O operation. This can be a multiplexed input operation, such as **occam**'s `ALT` construct [Bur88]; or a generalized (or symmetric) select operation that multiplexes both input and output communications, such as **Pascal-m**'s `select` construct [AB86]. Implementing generalized select on a multi-processor can be difficult [Bor86], but there are situations in which generalized selective communication is necessary (an example of this is given in Section 3.1).

Unfortunately, there is a fundamental conflict between the desire for abstraction and the need for selective communication. For example, consider a server thread

[2] The "`1`" in the type variable of `channel`'s result type is the *strength* of the variable. This is a technical mechanism used to allow polymorphic use of updatable objects without creating type loopholes.

that provides a service via a *request-reply* (or *remote procedure call* (RPC) style) protocol. The server side of this protocol is something like:

```
fun serverLoop () = if serviceAvailable()
      then let
        val request = accept reqCh
        in
          send (replyCh, doit request);
          serverLoop ()
        end
      else doSomethingElse()
```

where the function `doit` actually implements the service. Note that the service is not always available. This protocol requires that clients obey the following two rules:

1. A client must send a request before trying to read a reply.
2. Following a request the client must read exactly one reply before issuing another request.

If all clients obey these rules, then we can guarantee that each request is answered with the correct reply, but if a client breaks one of these rules, then the requests and replies will be out of sync. An obvious way to improve the reliability of programs that use this service is to bundle the client-side protocol into a function that hides the details, thus ensuring that the rules are followed. The following code implements this abstraction:

```
fun clientCall x = (send(reqCh, x); accept replyCh)
```

While this insures that clients obey the protocol, it hides too much. If a client blocks on a call to `clientCall` (e.g., if the server is not available), then it cannot respond to other communications. Avoiding this situation requires using selective communication, but the client cannot do this because the function abstraction hides the synchronous aspect of the protocol. This is the fundamental conflict between selective communication and the existing forms of abstraction. If we make the operation abstract, we lose the flexibility of selective communication; but if we expose the protocol to allow selective communication, we lose the safety and ease of maintenance provided by abstraction. The next section describes our solution to this dilemma.

3 First-class Synchronous Operations

To resolve the conflict between abstraction and selective communication requires introducing a new abstraction mechanism that preserves the synchronous nature of the abstraction. First-class synchronous operations provide this abstraction mechanism [Rep88, Rep89, Rep91a].

The traditional `select` construct has four facets: the individual I/O operations, the actions associated with each operation, the nondeterministic choice, and the synchronization. In **CML**, we unbundle these facets by introducing a new type of values, called *events*, that represent synchronous operations. By starting with *base-event* values to represent the communication operations, and providing combinators to associate actions with events and to build nondeterministic choices of events, a flexible mechanism for building new synchronization and communication abstractions is realized. Event values provide a mechanism for building an abstract representation of a protocol without obscuring its synchronous aspect.

To make this concrete, consider the following loop (using an **Amber** style `select` construct [Car86]), which implements the body of an accumulator that accepts either addition or subtraction input commands and offers its contents:

```
fun accum sum = (
      select addCh?x => accum(sum+x)
          or subCh?x => accum(sum-x)
          or readCh!sum => accum sum)
```

The `select` construct consists of three I/O operations: `addCh?x`, `subCh?x`, and `readCh!sum`. For each of these operations there is an associated action on the right hand side of the `=>`. Taken together, each I/O operation and associated action define a clause in a nondeterministic synchronous choice. It is also worth noting that the input clauses define a scope: the input operation binds an identifier to the incoming message, which has the action as its scope.

Figure 2 gives the signature of the event operations corresponding to the four facets of generalized selective communication. The functions `receive` and `transmit`

```
val receive  : 'a chan -> 'a event
val transmit : ('a chan * 'a) -> unit event

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

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

Fig. 2. Basic event operations

build base-event values that represent channel I/O operations. The `wrap` combinator binds an action, represented by a function, to an event value. And the `choose` combinator composes event values into a nondeterministic choice. The last operation is `sync`, which forces synchronization on an event value. I call this set of operations "**PML** events," since they constitute the mechanism that I originally developed in **PML** [Rep88].

The simplest example of events is the implementation of the synchronous channel I/O operations that were described in the previous section. These are defined using function composition, `sync` and the channel I/O event-value constructors:

```
val accept = sync o receive
val send   = sync o transmit
```

A more substantial example is the accumulator loop from above, which is implemented as:

```
fun accum sum = sync (
      choose [
          wrap (receive addCh, fn x => accum (sum+x)),
          wrap (receive subCh, fn x => accum (sum-x)),
          wrap (transmit (readCh, sum), fn () => accum sum)
        ])
```

Notice how `wrap` is used to associate actions with communications.

The great benefit of this approach to concurrency is that it allows the programmer to create new first-class synchronization and communication abstractions. For example, we can define an event-valued function that implements the client-side of the RPC protocol given in the previous section as follows:

```
fun clientCallEvt x = wrap (transmit(reqCh, x), fn () => accept replyCh)
```

Applying `clientCallEvt` to a value v does not actually send a request to the server, rather it returns an event value that can be used to send v to the server and then accept the server's reply. This event value can be used in a `choose` expression with other communications; in which case the `transmit` base-event value is used in selecting the event. This example shows that we can use first-class synchronous operations to abstract away from the details of the client-server protocol, without hiding the synchronous nature of the protocol.

This approach to synchronization and communication leads to a new programming paradigm, which I call *higher-order concurrent programming.* To understand this the higher-order nature of this mechanism, it is helpful to draw an analogy with first-class function values. Table 1 compares these two higher-order mechanisms.

Table 1. Relating first-class functions and events

Property	**Function values**	**Event values**
Type constructor	`->`	`event`

Introduction	λ-abstraction	`receive`
		`transmit`
		etc.

Elimination	application	`sync`

Combinators	composition
	`map`
	etc.
		`choose`
		`wrap`
		etc.

3.1 An Example

An example that illustrates a number of key points is an implementation of a *buffered channel* abstraction. Buffered channels provide a mechanism for asynchronous communication, which is similar to the actor mailbox [Agh86]. The source code for this abstraction is given in Figure 3. The function `buffer` creates a new buffered

```
abstype 'a buffer_chan = BC of {
    inch : 'a chan,
    outch : 'a chan
  }
with
  fun buffer () = let
        val inCh = channel() and outCh = channel()
        fun loop ([], []) = loop([accept inCh], [])
          | loop (front as (x::r), rear) = sync (
              choose [
                  wrap (receive inCh, fn y => loop(front, y::rear)),
                  wrap (transmit(outCh, x), fn () => loop(r, rear))
                ])
          | loop ([], rear) = loop(rev rear, [])
        in
          spawn (fn () => loop([], []));
          BC{inch=inCh, outch=outCh}
        end
  fun bufferSend (BC{inch, ...}, x) = send(inch, x)
  fun bufferReceive (BC{outch, ...}) = receive outch
end (* abstype *)
```

Fig. 3. Buffered channels

channel, which consists of a buffer thread, an input channel and an output channel; the function `bufferSend` is an asynchronous send operation; and the function `bufferReceive` is an event-valued receive operation. The buffer is represented as a queue of messages, which is implemented as a pair of stacks (lists). This example illustrates several key points:

- Buffered channels are a new communication abstraction, which have first-class citizenship. A thread can use the `bufferReceive` function in any context that it could use the built-in function `receive`, such as selective communication.
- The buffer loop uses both input and output operations in its selective communication. This is an example of the necessity of generalized selective communication. If we have only a multiplexed input construct (e.g., **occam**'s `ALT`), then we must to use a request/reply protocol to implement the server side of the `bufferReceive` operation (see pp. 37-41 of [Bur88], for example). But if a re-

quest/reply protocol is used, then the `bufferReceive` operation cannot be used in a selective communication by the client.

- The buffer thread is a good example of a common **CML** programming idiom: using threads to encapsulate state. This style has the additional benefit of hiding the state of the system in the concurrency operations, which makes the sequential code cleaner. These threads serve the same role that *monitors* do in some shared-memory concurrent languages.
- This implementation exploits the fact that unreachable blocked threads are garbage collected. If the clients of this buffer discard it, then the buffer thread and channels will be reclaimed by the garbage collector. This improves the modularity of the abstraction, since clients do not have to worry about explicit termination of the buffer thread.

3.2 Other Synchronous Operations

The event type provides a natural framework for accommodating other primitive synchronous operations.[3] There are three examples of this in **CML**: synchronization on thread termination (sometimes called *process join*), low-level I/O support and time-outs. Figure 4 gives the signature of the **CML** base-event constructors for these other synchronous operations. The function `wait` produces an event for syn-

```
val wait : thread_id -> unit event

val syncOnInput  : int -> unit event
val syncOnOutput : int -> unit event

val waitUntil : time -> unit event
val timeout   : time -> unit event
```

Fig. 4. Other primitive synchronous operations

chronizing on the termination of another thread. This is often used by servers that need to release resources allocated to a client in the case that the client terminates unexpectedly. Support for low-level I/O is provided by the functions `syncOnInput` and `syncOnOutput`, which allow threads to synchronize on the status of file descriptors [UNI86]. These operations are used in **CML** to implement a multi-threaded I/O stream library. There are two functions for synchronizing with the clock: `waitUntil` and `timeout`. The function `waitUntil` returns an event that synchronizes on an absolute time, while `timeout` implements a relative delay. The function `timeout` can be used to implement a timeout in a choice. The following code, for example, defines an event that waits for up to a second for a message on a channel:

[3] This is the reason that the I use the term "event" to refer to first-class synchronous operations instead of using "communication."

```
choose [
    wrap (receive ch, SOME),
    wrap (timeout(TIME{sec=1, usec=0}), fn () => NONE)
  ]
```

By having a uniform mechanism for combining synchronous operations, **CML** provides a great deal of flexibility with a fairly terse mechanism. As a comparison, **Ada** has two different timeout mechanisms: a time entry call for clients and delay statement that servers can include in a select.

CML also provides a polling mechanism. The operation

```
val poll : 'a event -> 'a option
```

is a non-blocking form of the `sync` operator. It returns `NONE` is the case where `sync` would block.

3.3 Extending PML events

Thus far, I have described the **PML** subset of first-class synchronous operations. In this section, I motivate and describe two significant extensions to **PML** events that are provided in **CML**.

Consider a protocol consisting of a sequence of communications: $c_1; c_2; \cdots; c_n$. When this protocol is packaged up in an event value, one of the c_i is designated as the *commit point*, the communication by which this event is chosen in a selective communication (e.g., the message send operation in the `clientCallEvt` abstraction above). In **PML** events, the only possible commit point is c_1. The `wrap` construct allows one to tack on $c_2; \cdots; c_n$ after c_1 is chosen, but there is no way to make any of the other c_i the commit point. This asymmetry is a serious limitation.

A good illustration of this problem is a server that implements an input-stream abstraction. Since this abstraction should be smoothly integrated into the concurrency model, the input operations should be event-valued. For example, the function

```
val input : instream -> string event
```

is used to read a single character. In addition, there are other input operations such as `input_line`. Let us assume that the implementation of these operations uses a request-reply protocol; thus, a successful `input` operation involves the communication sequence

`send` $(ch_{req}$, `REQ_INPUT`$)$; `accept`(ch_{reply})

Packaging this up as an event (as we did in Section 3) will make the `send` communication be the commit point, which is the wrong semantics. To illustrate the problem with this, consider the case where a client thread wants to synchronize on the choice of reading a character and a five second timeout:

```
sync (choose [
    wrap (timeout(TIME{sec=5, usec=0}), fn () => raise Timeout),
    input instream
  ])
```

The server might accept the request within the five second limit, even though the wait for input might be indefinite. The right semantics for the `input` operation requires making the `accept` be the commit point, which is not possible using only the **PML** subset of events. To address this limitation, **CML** provides the `guard` combinator.

Guards. The `guard` combinator is the dual of `wrap`; it bundles code to be executed *before* the commit point; this code can include communications. It has the type

```
val guard : (unit -> 'a event) -> 'a event
```

A guard event is essentially a suspension that is forced when `sync` is applied to it. As a simple example of the use of `guard`, the `timeout` function, described in Section 3.2, is actually implemented using `waitUntil` and a guard:

```
fun timeout t = guard (fn () => waitUntil (add_time (t, currentTime()))
```

where `currentTime` returns the current time.

Some languages support guarded clauses in selective communication, where the guards are boolean expressions that must evaluate to true in order that the communication be enabled. **CML** guards can be used for this purpose too, as illustrated by the following code skeleton:

```
sync (choose [
    ...
    guard (fn () => if pred then evt else choose[])
    ...
  ])
```

Here *evt* is part of the choice only if *pred* evaluates to `true`. Note that the evaluation of *pred* occurs each time the guard function is evaluated (i.e., each time the `sync` is applied).

Returning to the RPC example from above, we can now build an abstract RPC operation with the reply as the commit point. The two different versions are:

```
fun clientCallEvt1 x = wrap (transmit(reqCh, x), fn () => accept replyCh)

fun clientCallEvt2 x = guard (fn () => (send(reqCh, x); receive replyCh)
```

where the `clientCallEvt1` version commits on the server's acceptance of the request, while the `clientCallEvt2` version commits on the server's reply to the request. Note the duality of `guard` and `wrap` with respect to the commit point. Using guards to generate requests in this way raises a couple of other problems. First of all, if the server cannot guarantee that requests will be accepted promptly, then evaluating the guard may cause delays. The solution to this is to spawn a new thread to issue the request asynchronously:

```
fun clientCallEvt3 x = guard (fn () => (
      spawn(fn () => send(reqCh, x));
      receive replyCh)
```

Another alternative is for the server to be a clearing-house for requests; spawning a new thread to handle each new request.

The other problem is more serious: what if this RPC event is used in a selective communication and some other event is chosen? How does the server avoid blocking forever on sending a reply? For idempotent services, this can be handled by having the client create a dedicated channel for the reply and having the server spawn a new thread to send the reply. The client side of this protocol is

```
fun clientCallEvt4 x = guard (fn () => let
      val replyCh = channel()
      in
        spawn(fn () => send(reqCh, (replyCh, x)));
        receive replyCh
      end)
```

When the server sends the reply it evaluates

```
spawn (fn () => send(replyCh, reply))
```

If the client has already chosen a different event, then this thread blocks and will be garbage collected. For services that are not idempotent, this scheme is not sufficient; the server needs a way to *abort* the transaction. The `wrapAbort` combinator provides this mechanism.

Abort actions. The `wrapAbort` combinator associates an *abort action* with an event value. The semantics are that if the event is *not* chosen in a `sync` operation, then a new thread is spawned to evaluate the abort action. The type of this combinator is:

```
val wrapAbort : ('a event * (unit -> unit)) -> unit
```

where the second argument is the abort action. This combinator is the complement of `wrap` in the sense that if you view every base event in a choice as having both a wrapper and an abort action, then, when `sync` is applied, the wrapper of the chosen event is called and threads are spawned for each of the abort actions of the other base events.

Using `wrapAbort`, we can now implement the RPC protocol for non-idempotent services. The client code for the RPC using abort must allocate two channels; one for the reply and one for the abort message:

```
fun clientCallEvt5 x = guard (fn () => let
      val replyCh = channel()
      val abortCh = channel()
      fun abortFn () = send (abortCh, ())
      in
        spawn(fn () => send (reqCh, (replyCh, abortCh, x)));
        wrapAbort (receive replyCh, abortFn)
      end)
```

When the server is ready to reply (i.e., commit the transaction), it synchronizes on the following event value:

```
choose[
    wrap (receive abortCh, fn () => abort the transaction),
    wrap (transmit (replyCh, reply), fn () => commit the transaction)
  ]
```

This mechanism can be used to implement the input-stream abstraction discussed at the beginning of this section, and in fact, the concurrent stream I/O library provided by **CML** is implemented in this way.

3.4 Stream I/O

CML provides a concurrent version of the **SML** stream I/O primitives. Input operations in this version are event valued, which allows them to be used in selective communication. For example, a program might want to give a user 60 seconds to supply a password. This can be programmed as:

```
fun getpasswd () = sync (choose [
        wrap (timeout{sec=60, usec=0}, fn () => NONE),
        wrap (input_line std_in, SOME)
      ])
```

This will return `NONE`, if the user fails to respond within 60 seconds, otherwise it wraps `SOME` around the user's response. Streams are implemented as threads which handle buffering. The input operations are actually request/reply/abort protocols, similar to the one discussed above.

4 Applications

CML is more than an exercise in language design; it is intended to be a useful tool for building large systems. I have implemented **CML** on top of **SML/NJ**. This implementation has been used by a number of people, including myself, for various different applications. This practical experience demonstrates the validity and usefulness of the design as well as the efficiency of the implementation. In this section, I describe two applications of **CML**, and how they use the features of **CML**.

4.1 Interactive Systems

Providing a better foundation for programming interactive systems, such as programming environments, was the original motivation for this line of research [RG86]. Because of their naturally concurrent structure, interactive systems are one of the most important application areas for **CML**. Concurrency arises in several ways in interactive systems:

User interaction. Handling user input is the most complex aspect of an interactive program. Most interactive systems use an *event-loop* and *call-back* functions. The event-loop receives input events (e.g., mouse clicks) and passes them to the appropriate *event-handler*. This structure is a poor-man's concurrency: the event-handlers are coroutines and the event-loop is the scheduler.

Multiple services. For example, consider a document preparation system that provides both editing and formatting. These two services are independent and can be naturally organized as two separate threads. Multiple views are implemented by replicating the threads.

Interleaving computation. A user of a document preparation system may want to edit one part of a document while another part is being formatted. Since formatting may take a significant amount of time, providing a responsive interface requires interleaving formatting and editing. If the editor and formatter are separate threads, then interleaving comes for free.

Output driven applications. Most windowing toolkits, for example **Xlib** [Nye90], provide an *input oriented* model, in which the application code is occasionally called in response to some external event. But many applications are *output* oriented. Consider, for example, a computationally intensive simulation with a graphical display of the current state of the simulation. This application must monitor window events, such as refresh and resize notifications, so that it can redraw itself when necessary. In a sequential implementation, the handling of these events must be postponed until the simulation is ready to update the displayed information. By separating the display code and simulation code into separate threads, the handling of asynchronous redrawing is easy.

The root cause of these forms of concurrency is computer-human interaction: humans are asynchronous and slow.

CML has been used to build a multi-threaded interface to the **X** protocol [SG86], called **eXene** [GR91]. This system provides a substantially different, and we think better, model of user interaction than the traditional **Xlib** model [Nye90]. Windows in **eXene** have an *environment*, consisting of three streams of input from the window's parent (mouse, keyboard and control), and one output stream for requesting services from the window's parent. For each child of the window, there will be corresponding output streams and an input stream. The input streams are represented by event values and the output streams by event valued functions. A window is responsible for routing messages to its children, but this can almost always be done using a generic router function provided by **eXene**. Typically, each window has a separate thread for each input stream as well as a thread, or two, for managing state and coordinating the other threads. By breaking the code up this way, each individual thread is quite simple. This model is similar to those of [Pik89] and [Haa90]).

This structure allows us to use *delegation* techniques to define new behavior from existing implementations. Delegation is an object-oriented technique that originated in concurrent actor systems [Lie86]. As an example, consider the case of adding a menu to an existing text window. We can do this in a general way by defining a wrapper that takes a window's environment and returns a new, wrapped, environment. The wrapped environment has a thread monitoring the mouse stream of the old environment. Normally, this thread just passes messages along to the wrapped window, but when a mouse "button down" message comes along, the thread pops up the menu and consumes mouse messages until an item is chosen. Emden Gansner, of AT&T Bell Laboratories, has developed a "widget" toolkit on top of **eXene**, which uses these delegation techniques heavily.

The implementation of **eXene**, which is currently about 8,500 lines of **CML** code, uses threads heavily. At the lowest level, threads are used to buffer communication with the **X** server. There are threads to manage shared state, such as graphics contexts, fonts and keycode translation tables. Because the internal threads are fairly specialized and tightly integrated, there is not much use of events as an abstraction mechanism.

The use of events as an abstraction mechanism is common at the application programmer's level. In addition to the event-valued interface of the window environments, there are higher-level objects that have abstract synchronous interfaces. One example is a *virtual terminal* window (vtty). This provides a synchronous stream interface to its clients, which is compatible with the signature of **CML**'s concurrent I/O library. If the client-code is implemented as a *functor* [Mac84] (parameterized module), then it can be used with either the concurrent I/O library or the vtty abstraction.

The vtty abstraction is a good example of where user-defined abstract synchronous operations are necessary for program modularity. At any time, the vtty thread must be ready to receive input from the user and output from the application; thus it needs selective communication. The underlying window toolkit (**eXene**) provides an abstract interface to the input stream, but, since it is event-valued, it can be used in the selective communication.

Another example of the use of new communication abstractions is a *buffered multicast* channel (a simple version is described in [Rep90b]). This abstraction has proven quite useful in supporting multiple views of an object. When the viewed object is updated, the thread managing its state sends a notification on the multicast channel. The multicast channel basically serves the role of a call-back function, while freeing the viewed object from the details on managing multiple views. All of the details of creating/destroying views and distributing messages are taken care of by the multicast abstraction.

4.2 Distributed systems programming

Many distributed programming languages have concurrent languages at their core (e.g., **SR** [AOCE88]), and distributed programming toolkits often include thread packages (e.g., **Isis** [BCJ+90]). This is because threads provide a needed flexibility for dealing with the asynchronous nature of distributed systems. The flexibility provided by **CML** is a good base for distributed programming. Its support for low-level I/O is sufficient to build a structured synchronous interface to network communication (as was done in **eXene**). Higher-level linguistic support for distributed programming, such as the promise mechanism described in [LS88], can be built using events to define the new abstractions.[4]

Another example is Chet Murthy's reimplementation of the **Nuprl** environment [Con86] using **CML**. His implementation is structured as a collection of "proof servers" running on different workstations. When an expensive operation on a proof tree is required, it can be decomposed and run in parallel on several different workstations. This system uses **CML** to manage the interactions between the different workstations.

Another project involving **CML** is the development of a distributed programming toolkit for **ML** that is being developed at Cornell University [Kru91, CK92, Kru92]. This work builds on the mechanisms prototyped in Murthy's distributed **Nuprl** and on the protocols developed for **Isis** [BCJ+90]. A new abstraction, called a *port group* has been developed to model distributed communication. The communication operations provided by port groups are represented by event-value constructors (for details see [Kru91]).

4.3 Other applications of CML

CML has been used by various people for a number of other purposes. Andrew Appel has used it to teach concurrent programming to undergraduates at Princeton University (Appel, personal communication, January 1991). Gary Lindstrom and Lal George have used it to experiment with functional control of imperative programs for parallel programming [GL91]. And Clément Pellerin has implemented a compiler from a concurrent constraint language to **CML** [Pel92].

5 Implementation

CML is written entirely in **SML**, using a couple of non-standard extensions provided by **SML/NJ**: *first-class continuations* [DHM91] and *asynchronous signals* [Rep90a]. We added one minor primitive operation to the compiler (a ten line change in a 30,000 line compiler), which was necessary to guarantee that tail-recursion is preserved by sync. Threads are implemented using *first-class continuations* (a technique owed to

[4] See [GR91] or [Rep92] for a description of the implementation of promises in **CML**.

Wand [Wan80]), and the **SML/NJ** asynchronous signal facility is used to implement preemptive scheduling.

Unlike other continuation-passing style compilers, such as **Rabbit** [Ste78] and **Orbit** [KKR+86], the code generated by the **SML/NJ** compiler does not use a run-time stack [App92]. This means that `callcc` and `throw` are constant-time operations. While this is possible using a stack [HDB90]; heap-based implementations are better suited for implementing light-weight threads (Haahr's experience bears this out [Haa90]).

Event values have a natural implementation in terms of first-class continuations. Without the `choose` operator, an event value could be represented as

```
type 'a event = 'a cont -> 'a
```

with `sync` being directly implemented by `callcc`. This representation captures the intuition that an event is just a synchronous operation with its synchronization point continuation as a free variable. The `choose` operator requires polling, since we need to see which (if any) base events are immediately available for synchronization. Thus, the implementation of an event value is a list of base events, with each base event represented by a polling function, a function to call for immediate synchronization and a function for blocking. The implementation of **CML** is described in detail, including detailed performance measurements, in the author's dissertation [Rep92].

6 Dynamic Semantics

In this section, I present the dynamic semantics of λ_{cv}, a concurrent extension of Plotkin's λ_v calculus [Plo75]. Although λ_{cv} lacks many of the features of **CML**, it contains the core of the concurrency primitives, including first-class synchronous operations, and the various event-value combinators. A discussion of how λ_{cv} can be extended to include many of the missing features of **CML** can be found in [Rep92].

The dynamic semantics of λ_{cv} is defined by two evaluation relations: a sequential evaluation relation "$\longmapsto$," and a concurrent evaluation relation "$\Longrightarrow$," where concurrent evaluation is an extension of sequential evaluation to finite sets of processes.

6.1 The Syntax of λ_{cv}

We start with disjoint sets of *variables*, *function constants*, *base constants* and *channel names*, which comprise the ground terms of λ_{cv}:

$x \in \text{Var}$	variables
$b \in \text{Const} = \text{BConst} \cup \text{FConst}$	constants
$\quad \text{BConst} = \{\texttt{()}, \texttt{true}, \texttt{false}, 0, 1, \ldots\}$	base constants
$\quad \text{FConst} = \{\texttt{+}, \texttt{-}, \texttt{fst}, \texttt{snd}, \ldots\}$	function constants
$\kappa \in \text{Ch}$	channel names

The set FConst includes the following event-valued combinators and constructors:

choose, **guard**, **never**, **receive**, **transmit**, **wrap**, **wrapAbort**

There are three syntactic classes of terms in λ_{cv}:

$$\begin{array}{rll} e \in \textsc{Exp} & & \text{expressions} \\ v \in \textsc{Val} \subset \textsc{Exp} & & \text{values} \\ ev \in \textsc{Event} \subset \textsc{Val} & & \text{event values} \end{array}$$

where values are the irreducible terms in the dynamic semantics. The terms of λ_{cv} are defined by the grammar in Figure 5. Pairs have been included to make the handling

$$\begin{array}{rll}
e ::= & v & \text{value} \\
| & e_1\ e_2 & \text{application} \\
| & (e_1 . e_2) & \text{pair} \\
| & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \text{let} \\
| & \mathbf{chan}\ x\ \mathbf{in}\ e & \text{channel creation} \\
| & \mathbf{spawn}\ e & \text{process creation} \\
| & \mathbf{sync}\ e & \text{synchronization} \\
\\
v ::= & b & \text{constant} \\
| & x & \text{variable} \\
| & (v_1 . v_2) & \text{pair value} \\
| & \lambda x(e) & \lambda\text{-abstraction} \\
| & \kappa & \text{channel name} \\
| & ev & \text{event value} \\
| & (\mathbf{G}\, e) & \text{guarded event function} \\
\\
ev ::= & \Lambda & \text{never} \\
| & \kappa!v & \text{channel output} \\
| & \kappa? & \text{channel input} \\
| & (ev \Rightarrow v) & \text{wrapper} \\
| & (ev_1 \oplus ev_2) & \text{choice} \\
| & (ev \mid v) & \text{abort wrapper}
\end{array}$$

Fig. 5. Grammar for λ_{cv}

of two-argument functions easier. Note that the syntactic class of the term $(v_1 . v_2)$ is either Exp or Val; this ambiguity is resolved in favor of Val. The value Λ is a base event value that is never matched (equivalent to `choose[]` in **CML**). There are three binding forms in this term language: let binding, λ-abstraction and channel creation. Unlike **CML**, new channels are introduced by the special binding form for channel creation. This is done to simplify the presentation of the next chapter, and the `channel` function of **CML** can be easily defined in terms of λ_{cv}. The set $\textsc{Val}^{\circ}$ is the set of *closed* value terms (i.e., those without free variables); note, however, that closed values may contain free channel names. The free channel names of an expression e are denoted by $\mathrm{FCN}(e)$. Note that, since there are no channel name binding forms, $\mathrm{FCN}(e)$ is exactly the set of channel names that appear in e. There is no term for sequencing, but we write "$(e_1\,;\,e_2)$" for "$\texttt{snd}\ (e_1\,.\,e_2)$," which, since the language is call-by-value, has the desired semantics.

Channel names and event values are not part of the concrete syntax of the language; rather, they appear as the intermediate results of evaluation. A *program* is a closed term, which does not contain any guarded event functions (i.e., $(\mathbf{G}\,e)$ terms), or any subterms in the syntactic classes $\textsc{Event}$ or $\textsc{Ch}$. In other words, programs do not contain intermediate values.

6.2 Sequential Evaluation

There are a number of different ways to specify the dynamic semantics of programming languages. I use the style of *operational semantics* developed by Felleisen and Friedman [FF86], because it provides a good framework for proving type soundness results [WF91]. In this approach, the objects of the dynamic semantics are the syntactic terms in $\textsc{Exp}$.

The meaning of the function constants is given by the partial function

$$\delta : \textsc{FConst} \times \textsc{Val}^{\circ} \longrightarrow \textsc{Val}^{\circ}$$

Since a closed value $v \in \textsc{Val}^{\circ}$ can have free channel names in it, we require, that if $b \in \textsc{FConst}$ and $\delta(b, v)$ is defined, then

$$\mathrm{FCN}(\delta(b,v)) \subseteq \mathrm{FCN}(v)$$

In other words, δ is not allowed to introduce new channel names. For the standard built-in function constants, the meaning of δ is the expected one. For example:

$$\begin{aligned}
\delta(\texttt{+}, (\texttt{0}\,.\,\texttt{1})) &= \texttt{1}\\
\delta(\texttt{+}, (\texttt{1}\,.\,\texttt{1})) &= \texttt{2}\\
\delta(\texttt{fst}, (v_1\,.\,v_2)) &= v_1\\
\delta(\texttt{snd}, (v_1\,.\,v_2)) &= v_2
\end{aligned}$$

The meaning of δ is straightforward for most of the event-valued combinators and constructors:

$$\begin{aligned}
\delta(\texttt{never}, ()) &= \Lambda\\
\delta(\texttt{transmit}, (\kappa\,.\,v)) &= \kappa!v\\
\delta(\texttt{receive}, \kappa) &= \kappa?\\
\delta(\texttt{wrap}, (ev\,.\,v)) &= (ev \Rightarrow v)\\
\delta(\texttt{choose}, (ev_1\,.\,ev_2)) &= (ev_1 \oplus ev_2)\\
\delta(\texttt{wrapAbort}, (ev\,.\,v)) &= (ev \mid v)
\end{aligned}$$

The only complication arises in the case of guarded-event values:

$$\begin{aligned}
\delta(\texttt{guard}, v) &= (\mathbf{G}\ (v\ ()))\\
\delta(\texttt{wrap}, ((\mathbf{G}\,e)\,.\,v)) &= (\mathbf{G}\ (\texttt{wrap}\ (e\,.\,v)))\\
\delta(\texttt{choose}, ((\mathbf{G}\,e_1)\,.\,(\mathbf{G}\,e_2))) &= (\mathbf{G}\ (\texttt{choose}\ (e_1\,.\,e_2)))\\
\delta(\texttt{choose}, ((\mathbf{G}\,e_1)\,.\,ev_2)) &= (\mathbf{G}\ (\texttt{choose}\ (e_1\,.\,ev_2)))\\
\delta(\texttt{choose}, (ev_1\,.\,(\mathbf{G}\,e_2))) &= (\mathbf{G}\ (\texttt{choose}\ (ev_1\,.\,e_2)))\\
\delta(\texttt{wrapAbort}, ((\mathbf{G}\,e)\,.\,v)) &= (\mathbf{G}\ (\texttt{wrapAbort}\ (e\,.\,v)))
\end{aligned}$$

These rules reflect **guard**'s role as a delay operator; when another event constructor is applied to a guarded event value, then the guard operator (**G**) is pulled out to delay the event construction.

An *evaluation context* is a single-hole context where the hole marks the next redex (or is at the top if the term is irreducible) The evaluation of λ_{cv} is call-by-value and left-to-right, which leads to the following grammar for the evaluation contexts of λ_{cv}:

$$\begin{aligned} E ::= &\ [\,] \mid E\,e \mid v\,E \mid (E\,.\,e) \mid (v\,.\,E) \\ &\mid \mathtt{let}\ x = E\ \mathtt{in}\ e \mid \mathtt{spawn}\ E \mid \mathsf{sync}\ E \end{aligned}$$

The sequential evaluation relation is defined in terms of these contexts:

Definition 1 ($\longmapsto$). The *sequential evaluation relation*, written "$\longmapsto$," is the smallest relation satisfying the following four rules:

$$\begin{array}{lll} E[b\ v] & \longmapsto E[\delta(b, v)] & (\lambda_{cv}\text{-}\delta) \\ E[\lambda x(e)\ v] & \longmapsto E[e[x \mapsto v]] & (\lambda_{cv}\text{-}\beta) \\ E[\mathtt{let}\ x = v\ \mathtt{in}\ e] & \longmapsto E[e[x \mapsto v]] & (\lambda_{cv}\text{-}\mathbf{let}) \\ E[\mathsf{sync}\ (\mathbf{G}\,e)] & \longmapsto E[\mathsf{sync}\ e] & (\lambda_{cv}\text{-}\mathbf{guard}) \end{array}$$

Note that the rule (λ_{cv}-**guard**) forces the expression delayed by **guard**. As usual, $\longmapsto^*$ is the transitive closure of $\longmapsto$. The evaluation of the other new forms (e.g., `spawn`) is defined as part of the concurrent evaluation relation in Section 6.4.

6.3 Event Matching

The key concept in the semantics of concurrent evaluation is the notion of *event matching*, which captures the semantics of rendezvous and communication. Informally, if two processes synchronize on matching events, then they can exchange values and continue evaluation. Before we can make this more formal, we need an auxiliary definition

Definition 2. The *abort action* of an event value ev is an expression, which, when evaluated, spawns the abort wrappers of ev. The map

$$\mathrm{AbortAct} : \textsc{Event} \rightarrow \textsc{Exp}$$

maps an event value to its abort action, and is defined inductively as follows:

$$\begin{aligned} \mathrm{AbortAct}(\Lambda) &= () \\ \mathrm{AbortAct}(\kappa?) &= () \\ \mathrm{AbortAct}(\kappa!v) &= () \\ \mathrm{AbortAct}(ev \Rightarrow e) &= \mathrm{AbortAct}(ev) \\ \mathrm{AbortAct}(ev_1 \oplus ev_2) &= (\mathrm{AbortAct}(ev_1);\ \mathrm{AbortAct}(ev_2)) \\ \mathrm{AbortAct}(ev \mid v) &= (\mathrm{AbortAct}(ev);\ \mathtt{spawn}\ v) \end{aligned}$$

With this definition we can define the matching of event values formally:

Definition 3 (Event matching). The matching of event values is defined as a family of binary symmetric relations (indexed by CH). For $\kappa \in \mathrm{CH}$, define

$$ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)$$

(pronounced *"ev_1 matches ev_2 on channel κ with respective results e_1 and e_2*) as the smallest relation satisfying the six inference rules given in Figure 6. This relation is abbreviated to $ev_1 \overset{\kappa}{\frown} ev_2$ when the results are unimportant.

$$\frac{}{\kappa!v \overset{\kappa}{\frown} \kappa? \ \mathtt{with}\ ((), v)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)}{ev_2 \overset{\kappa}{\frown} ev_1 \ \mathtt{with}\ (e_2, e_1)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \Rightarrow v) \ \mathtt{with}\ (e_1, v\ e_2)}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \oplus ev_3) \ \mathtt{with}\ (e_1, (\mathrm{AbortAct}(ev_3);\ e_2))}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_3 \oplus ev_2) \ \mathtt{with}\ (e_1, (\mathrm{AbortAct}(ev_3);\ e_2))}$$

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \ \mathtt{with}\ (e_1, e_2)}{ev_1 \overset{\kappa}{\frown} (ev_2 \mid v) \ \mathtt{with}\ (e_1, e_2)}$$

Fig. 6. Rules for event matching

An example of event matching is:

$$(\kappa? \Rightarrow \lambda x((x\,.\,x))) \overset{\kappa}{\frown} (\kappa!\mathtt{17} \oplus (\kappa? \Rightarrow \lambda x())) \ \mathtt{with}\ (\lambda x((x\,.\,x))\ \mathtt{17}, ())$$

Informally, if two processes attempt to synchronize on matching event values, then we can replace the applications of sync with the respective results. This is made more precise in the next section where the concurrent evaluation relation is defined.

Note that event matching is nondeterministic; for example, both

$$\kappa? \overset{\kappa}{\frown} (\kappa!\mathtt{17} \oplus \kappa!\mathtt{29}) \ \mathtt{with}\ (\mathtt{17}, ())$$

and

$$\kappa? \overset{\kappa}{\frown} (\kappa!17 \oplus \kappa!29) \text{ with } (29, ())$$

It is also worth noting that even if one of the wrappers of an event value is non-terminating, the necessary abort actions for that event will be executed (assuming fair evaluation). This property is important because a common **CML** idiom is to have tail-recursive calls in wrappers (e.g., the buffered channel abstraction in Figure 3).

6.4 Concurrent Evaluation

Concurrent evaluation is defined as a transition system between finite sets of process states. This is similar to the style of the "Chemical Abstract Machine" [BB90], except that there are no "cooling" and "heating" transitions (the process sets of this semantics can be thought of as perpetually "hot" solutions). The concurrent evaluation relation extends "$\longmapsto$" to finite sets of terms (i.e., processes) and adds additional rules for process creation, channel creation, and communication. We assume a set of *process identifiers*, and define the set of *processes* and *process sets* as:

$$\begin{array}{rll} \pi \in \textsc{ProcId} & & \text{process IDs} \\ p = \langle \pi;\ e \rangle \in \textsc{Proc} = (\textsc{ProcId} \times \textsc{Exp}) & & \text{processes} \\ \mathcal{P} \in \mathrm{Fin}(\textsc{Proc}) & & \text{process sets} \end{array}$$

We often write a process as $\langle \pi;\ E[e] \rangle$, where the evaluation context serves the role of the program counter, marking the current state of evaluation.

Definition 4. A process set $\mathcal{P}$ is *well-formed* if for all $\langle \pi;\ e \rangle \in \mathcal{P}$ the following hold:

- $\mathrm{FV}(e) = \emptyset$ (e is *closed*), and
- there is no $e' \neq e$, such that $\langle \pi;\ e' \rangle \in \mathcal{P}$.

It is occasionally useful to view well-formed process sets as finite maps from $\textsc{ProcId}$ to $\textsc{Exp}$. If $\mathcal{P}$ is a finite set of process states and $\mathcal{K}$ is a finite set of channel names, then $\mathcal{K}, \mathcal{P}$ is a *configuration.*

Definition 5. A configuration $\mathcal{K}, \mathcal{P}$ is *well-formed*, if $\mathrm{FCN}(\mathcal{P}) \subseteq \mathcal{K}$ and $\mathcal{P}$ is well-formed.

The concurrent evaluation relation "$\Longrightarrow$" extends "$\longmapsto$" to configurations, with additional rules for the concurrency operations. It is defined by four inference rules that define single step evaluations. Each concurrent evaluation step affects one or two processes, called the *selected* processes. I first describe each of these rules independently, and then state the formal definition. In stating these rules, we use the notation $S{+}x$ for $S \cup \{x\}$.

The first rule extends the sequential evaluation relation ($\longmapsto$) to configurations:

$$\frac{e \longmapsto e'}{\mathcal{K}, \mathcal{P}{+}\langle \pi;\ e \rangle \Longrightarrow \mathcal{K}, \mathcal{P}{+}\langle \pi;\ e' \rangle} \qquad (\lambda_{cv}\text{-}\mapsto)$$

The selected process is π.

The creation of channels requires picking a new channel name and substituting for the variable bound to it:

$$\frac{\kappa \notin \mathcal{K}}{\mathcal{K}, \mathcal{P}+\langle \pi;\ E[\mathbf{chan}\ x\ \mathbf{in}\ e]\rangle \Longrightarrow \mathcal{K}+\kappa, \mathcal{P}+\langle \pi;\ E[e[x \mapsto \kappa]]\rangle} \qquad (\lambda_{cv}\text{-}\mathbf{chan})$$

Again, π is the selected process.

Process creation requires picking a new process identifier:

$$\frac{\pi' \notin \mathrm{dom}(\mathcal{P})+\pi}{\mathcal{K}, \mathcal{P}+\langle \pi;\ E[\mathtt{spawn}\ v]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}+\langle \pi;\ E[\texttt{()}]\rangle+\langle \pi';\ v\ \texttt{()}\rangle} \qquad (\lambda_{cv}\text{-}\mathbf{spawn})$$

This rule has two selected processes: π and π'.

The most interesting rule describes communication and synchronization. If two processes are attempting synchronization on matching events, then they may rendezvous — i.e., exchange a message and continue evaluation:

$$\frac{ev_1 \overset{\kappa}{\frown} ev_2 \text{ with } (e_1, e_2)}{\begin{array}{c}\mathcal{K}, \mathcal{P}+\langle \pi_1;\ E_1[\mathtt{sync}\ ev_1]\rangle+\langle \pi_2;\ E_2[\mathtt{sync}\ ev_2]\rangle \\ \Longrightarrow \mathcal{K}, \mathcal{P}+\langle \pi_1;\ E_1[e_1]\rangle+\langle \pi_2;\ E_2[e_2]\rangle\end{array}} \qquad (\lambda_{cv}\text{-}\mathbf{sync})$$

The selected processes for this rule are π_1 and π_2. We say that κ is *used* in this transition.

More formally, concurrent evaluation is defined as follows:

Definition 6 ($\Longrightarrow$). The *concurrent evaluation relation*, written "$\Longrightarrow$," is the smallest relation satisfying the rules: (λ_{cv}-$\longmapsto$), (λ_{cv}-**chan**), (λ_{cv}-**spawn**), and (λ_{cv}-**sync**).

Under these rules, processes live forever; i.e., if a process evaluates to a value, it will never again be selected, but it remains in the process set. We could add the following rule, which is similar to the *evaporation* rule of [BB90]:

$$\mathcal{K}, \mathcal{P}+\langle \pi;\ [v]\rangle \Longrightarrow \mathcal{K}, \mathcal{P}$$

This rule is not included because certain results are easier to state and prove if the process set is monotonicly increasing.

The following evaluation illustrates the concurrent evaluation relation:

$$\begin{array}{l}
\{\}, \{\langle \pi_0;\ [\mathtt{chan}\ k\ \mathtt{in}\ \mathtt{(spawn}\ \lambda x\mathtt{(sync\ (transmit\ (}k\mathtt{.5)));\ sync\ (receive}\ k\mathtt{))}]\rangle\} \\
\Longrightarrow\ \{\kappa_0\}, \{\langle \pi_0;\ \mathtt{(}[\mathtt{spawn}\ \lambda x\mathtt{(sync\ (transmit\ (}\kappa_0\mathtt{.5)))}];\ \mathtt{sync\ (receive}\ \kappa_0\mathtt{))}\rangle\} \\
\Longrightarrow\ \{\kappa_0\}, \{\langle \pi_0;\ \mathtt{(();\ sync\ (}[\mathtt{receive}\ \kappa_0]\mathtt{)))}\rangle, \\
\qquad\qquad \langle \pi_1;\ [\lambda x\mathtt{(sync\ (transmit\ (}\kappa_0\mathtt{.5)))\ ()}]\rangle\} \\
\Longrightarrow\ \{\kappa_0\}, \{\langle \pi_0;\ \mathtt{(();\ sync\ (}[\mathtt{receive}\ \kappa_0]\mathtt{)))}\rangle, \langle \pi_1;\ \mathtt{sync\ (}[\mathtt{transmit\ (}\kappa_0\mathtt{.5)}]\mathtt{)}\rangle\} \\
\Longrightarrow^*\ \{\kappa_0\}, \{\langle \pi_0;\ \mathtt{(();\ }[\mathtt{sync\ (}\kappa_0\mathtt{?)}]\mathtt{)}\rangle, \langle \pi_1;\ [\mathtt{sync\ (}\kappa_0\mathtt{!5)}]\rangle\} \\
\Longrightarrow\ \{\kappa_0\}, \{\langle \pi_0;\ [\mathtt{(();\ 5)}]\rangle, \langle \pi_1;\ [\mathtt{()}]\rangle\} \\
\Longrightarrow\ \{\kappa_0\}, \{\langle \pi_0;\ [\mathtt{5}]\rangle, \langle \pi_1;\ [\mathtt{()}]\rangle\}
\end{array}$$

Note that this is only one of several possible evaluation sequences, although, in this example, all evaluation sequences produce the same result.

6.5 Traces

Because of the non-deterministic semantics of $\Longrightarrow$, a λ_{cv} program can have many (often infinitely many) different evaluations. Furthermore, there are many interesting programs that do not terminate. Thus some new terminology and notation for describing evaluation sequences is required. This is used to state type soundness results for λ_{cv} in the next section.

First we note the following properties of $\Longrightarrow$:

Lemma 7. *If $\mathcal{K}, \mathcal{P}$ is well-formed and $\mathcal{K}, \mathcal{P} \Longrightarrow \mathcal{K}', \mathcal{P}'$ then the following hold:*

1. *$\mathcal{K}', \mathcal{P}'$ is well-formed*
2. *$\mathcal{K} \subseteq \mathcal{K}'$*
3. *$\mathrm{dom}(\mathcal{P}) \subseteq \mathrm{dom}(\mathcal{P}')$*

Proof. By examination of the rules for $\Longrightarrow$.

Corollary 8. *The properties of Lemma 7 hold for $\Longrightarrow^*$.*

Proof. By induction on the length of the evaluation sequence.

Note that property (1) implies that evaluation preserves closed terms.

Definition 9. A *trace* T is a (possibly infinite) sequence of well-formed configurations

$$T = \langle\!\langle \mathcal{K}_0, \mathcal{P}_0;\ \mathcal{K}_1, \mathcal{P}_1;\ \ldots \rangle\!\rangle$$

such that $\mathcal{K}_i, \mathcal{P}_i \Longrightarrow \mathcal{K}_{i+1}, \mathcal{P}_{i+1}$ (for $i < n$, if T is finite with length n). The *head* of T is $\mathcal{K}_0, \mathcal{P}_0$.

Note that if a configuration $\mathcal{K}_0, \mathcal{P}_0$ is well-formed, then any sequence of evaluation steps starting with $\mathcal{K}_0, \mathcal{P}_0$ is a trace (by Corollary 8).

The possible states of a process with respect to a configuration are given by the following definition.

Definition 10. Let $\mathcal{P}$ be a well-formed process set and let $p \in \mathcal{P}$, with $p = \langle \pi; e \rangle$. The *state* of π in $\mathcal{P}$ is either *zombie*, *blocked*, or *ready*, depending on the form of e:

- if $e = [v]$, then p is a *zombie*,
- if $e = E[\mathtt{sync}\ ev]$ and there does not exist a $\langle \pi'; E'[\mathtt{sync}\ ev'] \rangle \in (\mathcal{P} \setminus \{p\})$, such that $ev \overset{\mathcal{K}}{\frown} ev'$, then π is *blocked* in $\mathcal{P}$.

– otherwise, π is *ready* in $\mathcal{P}$.

We define the set of ready processes in $\mathcal{P}$ by

$$\mathrm{Rdy}(\mathcal{P}) = \{\pi \mid \pi \text{ is ready in } \mathcal{P}\}$$

A configuration $\mathcal{K}, \mathcal{P}$ is *terminal* if $\mathrm{Rdy}(\mathcal{P}) = \emptyset$. A terminal configuration with blocked processes is said to be *deadlocked.*

Definition 11. A trace is a *computation* if it is maximal; i.e., if it is infinite or if it is finite and ends in a terminal configuration. If e is a program, then we define the computations of e to be

$$\mathrm{Comp}(e) = \{T | T \text{ is a computation with head } \langle \pi_0;\, e\rangle\}$$

Note, I follow the convention of using π_0 as the process identifier of the initial process in a computation of a program.

Definition 12. The *set of processes of a trace* T is defined as

$$\mathrm{Procs}(T) = \{\pi \mid \exists \mathcal{K}_i, \mathcal{P}_i \in T \text{ with } \pi \in \mathrm{dom}(\mathcal{P}_i)\}$$

Since a given program can evaluate in different ways, the sequential notions of convergence and divergence are inadequate. Instead, we define convergence and divergence relative to a particular computation of a program.

Definition 13. A process $\pi \in \mathrm{Procs}(T)$ *converges* to a value v in T, written $\pi \Downarrow_T v$, if $\mathcal{K}, \mathcal{P} + \langle \pi;\, v\rangle \in T$. We say that π *diverges* in T, written $\pi \Uparrow_T$, if for every $\mathcal{K}, \mathcal{P} \in T$, with $\pi \in \mathrm{dom}(\mathcal{P})$, π is ready or blocked in $\mathcal{P}$.

Divergence includes deadlocked processes and terminating processes that are not evaluated often enough to reach termination, as well as those with infinite loops. It does not include processes with run-time type errors, which are called *stuck* (see Section 7.3).

7 Static Semantics

It is well known that references (i.e., updatable memory cells) can be coded using channels and processes [GMP89, Rep91b, BMT92]. This fact makes it apparent that polymorphic channels incur the same typing problems as polymorphic references (see [Tof90] for a good description of these problems). It is possible, however, to give a sound typing to λ_{cv} programs using the imperative type scheme of **SML** [MTH90, Tof90]. In this section, I present a type system for λ_{cv} programs, and state soundness results about the system. The sound typing of λ_{cv} is discussed in greater detail (including a proof of the soundness of the type system) in [Rep92].

The presentation uses standard notation (e.g., see [Tof90] or [WF91]). Let $\iota \in \textsc{TyCon} = \{\texttt{int}, \texttt{bool}, \ldots\}$ designate the *type constants.* Type variables are partitioned into two sets:

$$\begin{array}{rll} u \in \textsc{ImpTyVar} & & \text{imperative type variables} \\ t \in \textsc{AppTyVar} & & \text{applicative type variables} \\ \alpha, \beta \in \textsc{TyVar} = \textsc{ImpTyVar} \cup \textsc{AppTyVar} & & \text{type variables} \end{array}$$

The set of types, $\tau \in \textsc{Ty}$, is defined by

$$\begin{array}{rcll} \tau & ::= & \iota & \text{type constants} \\ & | & \alpha & \text{type variables} \\ & | & (\tau_1 \rightarrow \tau_2) & \text{function types} \\ & | & (\tau_1 \times \tau_2) & \text{pair types} \\ & | & \tau\ \texttt{chan} & \text{channel types} \\ & | & \tau\ \texttt{event} & \text{event types} \end{array}$$

and the set of type schemes, $\sigma \in \textsc{TyScheme}$, are defined by

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha . \sigma \end{array}$$

We write $\forall \alpha_1 \cdots \alpha_n . \tau$ for the type scheme $\sigma = \forall \alpha_1 \cdots \forall \alpha_n . \tau$, and write $\mathrm{FTV}(\sigma)$ for the free type variables of σ. We define the set of *imperative types* by

$$\theta \in \textsc{ImpTy} = \{\tau \mid \mathrm{FTV}(\tau) \subset \textsc{ImpTyVar}\}$$

Note that all of the free type variables in an imperative type are imperative.

Type environments assign type schemes to variables in terms. Since we are interested in assigning types to intermediate stages of evaluation, channel names also need to be assigned types. Therefore, a *typing environment* is a pair of finite maps: a *variable typing* and a *channel typing*:

$$\begin{array}{rcl} \mathrm{VT} \in \textsc{VarTy} & = & \textsc{Var} \xrightarrow{\text{fin}} \textsc{TyScheme} \\ \mathrm{CT} \in \textsc{ChanTy} & = & \textsc{Ch} \xrightarrow{\text{fin}} \textsc{ImpTy} \\ \mathrm{TE} = (\mathrm{VT}, \mathrm{CT}) \in \textsc{TyEnv} & = & (\textsc{VarTy} \times \textsc{ChanTy}) \end{array}$$

We use $\mathrm{FTV}(\mathrm{VT})$ and $\mathrm{FTV}(\mathrm{CT})$ to denote the sets of free type variables of variable and channel typings, and

$$\mathrm{FTV}(\mathrm{TE}) = \mathrm{FTV}(\mathrm{VT}) \cup \mathrm{FTV}(\mathrm{CT})$$

where $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$. Note that there are no bound type variables in a channel typing, and that $\mathrm{FTV}(\mathrm{CT}) \subset \textsc{ImpTyVar}$. The following shorthand is useful for type environment modification:

$$\begin{array}{rcl} \mathrm{TE} \pm \{x \mapsto \sigma\} & \equiv_{def} & (\mathrm{VT} \pm \{x \mapsto \sigma\}, \mathrm{CT}) \\ \mathrm{TE} \pm \{\kappa \mapsto \theta\} & \equiv_{def} & (\mathrm{VT}, \mathrm{CT} \pm \{\kappa \mapsto \theta\}) \end{array}$$

where $x \in \textsc{Var}$, $\kappa \in \textsc{Ch}$, and $\mathrm{TE} = (\mathrm{VT}, \mathrm{CT})$.

Because of the need to preserve imperative types, we require that substitutions map imperative type variables to imperative types. As before, we allow substitutions to be applied to types and type environments.

Definition 14. A type τ' is an *instance* of a type scheme $\sigma = \forall\alpha_1 \cdots \alpha_n.\tau$, written $\sigma \succ \tau'$, if there exists a finite substitution, S, with $\mathrm{dom}(S) = \{\alpha_1, \ldots, \alpha_n\}$ and $S\tau = \tau'$. If $\sigma \succ \tau'$, then we say that σ is a *generalization* of τ'. We say that $\sigma \succ \sigma'$ if whenever $\sigma' \succ \tau$, then $\sigma \succ \tau$.

Definition 15. The *closure* of a type τ with respect to a type environment TE is defined as: $\textsc{Clos}_{\mathrm{TE}}(\tau) = \forall\alpha_1 \cdots \alpha_n.\tau$, where

$$\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})$$

And the *applicative closure* of τ is defined as: $\textsc{AppClos}_{\mathrm{TE}}(\tau) = \forall\alpha_1 \cdots \alpha_n.\tau$, where

$$\{\alpha_1, \ldots, \alpha_n\} = (\mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})) \cap \textsc{AppTyVar}$$

7.1 Expression Typing Rules

To associate types with the constants, we assume the existence of a function

$$\mathrm{TypeOf} : \textsc{Const} \rightarrow \textsc{TyScheme}$$

For the concurrency related constants, it assigns the following type schemes:

$$\begin{array}{ll}
\texttt{never} & : \forall\alpha.(\texttt{unit} \rightarrow \alpha\ \texttt{event}) \\
\texttt{receive} & : \forall\alpha.(\alpha\ \texttt{chan} \rightarrow \alpha\ \texttt{event}) \\
\texttt{transmit} & : \forall\alpha.((\alpha\ \texttt{chan} \times \alpha) \rightarrow \texttt{unit}\ \texttt{event}) \\
\texttt{wrap} & : \forall\alpha\beta.((\alpha\ \texttt{event} \times (\alpha \rightarrow \beta)) \rightarrow \beta\ \texttt{event}) \\
\texttt{choose} & : \forall\alpha.((\alpha\ \texttt{event} \times \alpha\ \texttt{event}) \rightarrow \alpha\ \texttt{event}) \\
\texttt{guard} & : \forall\alpha.((\texttt{unit} \rightarrow \alpha\ \texttt{event}) \rightarrow \alpha\ \texttt{event}) \\
\texttt{wrapAbort} & : \forall\alpha.((\alpha\ \texttt{event} \times (\texttt{unit} \rightarrow \texttt{unit})) \rightarrow \alpha\ \texttt{event})
\end{array}$$

We also assume that there are no event-valued constants. More formally, we require that there does not exist any b such that $\mathrm{TypeOf}(b) = \tau\ \texttt{event}$, for some type τ.

The typing rules for λ_{cv} are divided into two groups. The core rules are given in Figure 7. There are two rules for `let`: the rule (**τ-app-let**) applies in the non-expansive case (in the syntax of λ_{cv}, this is when the bound expression is in $\textsc{Val}$); the rule (**τ-imp-let**) applies when the expression is expansive (not a value). There are also rules for typing channel names, and pair expressions. The rule (**τ-chan**) restricts the type of the introduced channel to be imperative. In addition to these core typing rules, there are rules for the other syntactic forms (see Figure 8). Given the appropriate environment, these rules can be derived from rule (**τ-app**) (rule (**τ-const**) in the case of Λ). It is useful, however, to include them explicitly. As before, it is worth noting that the syntactic form of a term uniquely determines which typing rule applies.

In order that the typing of constants be sensible, we impose a typability restriction on the definitions of δ and TypeOf. If $\mathrm{TypeOf}(b) \succ (\tau' \rightarrow \tau)$ and $\mathrm{TE} \vdash v : \tau'$, then $\delta(b, v)$ is defined and $\mathrm{TE} \vdash \delta(b, v) : \tau$. It is worth noting that the δ rules we defined for the concurrency constants respect this restriction.

$$\frac{\mathrm{TypeOf}(b) \succ \tau}{\mathrm{TE} \vdash b : \tau} \qquad (\tau\textbf{-const})$$

$$\frac{x \in \mathrm{dom}(\mathrm{VT}) \quad \mathrm{VT}(x) \succ \tau}{(\mathrm{VT}, \mathrm{CT}) \vdash x : \tau} \qquad (\tau\textbf{-var})$$

$$\frac{\mathrm{CT}(\kappa) = \theta}{(\mathrm{VT}, \mathrm{CT}) \vdash \kappa : \theta} \qquad (\tau\textbf{-chvar})$$

$$\frac{\mathrm{TE} \vdash e_1 : (\tau' \to \tau) \quad \mathrm{TE} \vdash e_2 : \tau'}{\mathrm{TE} \vdash e_1\ e_2 : \tau} \qquad (\tau\textbf{-app})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \tau\} \vdash e : \tau'}{\mathrm{TE} \vdash \lambda x\,(e) : (\tau \to \tau')} \qquad (\tau\textbf{-abs})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau_1 \quad \mathrm{TE} \vdash e_2 : \tau_2}{\mathrm{TE} \vdash (e_1 . e_2) : (\tau_1 \times \tau_2)} \qquad (\tau\textbf{-pair})$$

$$\frac{\mathrm{TE} \vdash v : \tau' \quad \mathrm{TE} \pm \{x \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau')\} \vdash e : \tau}{\mathrm{TE} \vdash \texttt{let}\ x = v\ \textbf{in}\ e : \tau} \qquad (\tau\textbf{-app-let})$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \textsc{AppClos}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \texttt{let}\ x = e_1\ \textbf{in}\ e_2 : \tau} \qquad (\tau\textbf{-imp-let})$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \theta\ \textbf{chan}\} \vdash e : \tau}{\mathrm{TE} \vdash \textbf{chan}\ x\ \textbf{in}\ e : \tau} \qquad (\tau\textbf{-chan})$$

Fig. 7. Core type inference rules for λ_{cv}

7.2 Process Typings

A *process typing* is a finite map from process identifiers to types:

$$\mathrm{PT} \in \textsc{ProcTy} = \textsc{ProcId} \xrightarrow{\text{fin}} \textsc{Ty}$$

Typing judgements are extended to process configurations by the following definition.

Definition 16. A well-formed configuration $\mathcal{K}, \mathcal{P}$ has type PT under a channel typing CT, written

$$\mathrm{CT} \vdash \mathcal{K}, \mathcal{P} : \mathrm{PT}$$

if the following hold:

- $\mathcal{K} \subseteq \mathrm{dom}(\mathrm{CT})$,
- $\mathrm{dom}(\mathcal{P}) \subseteq \mathrm{dom}(\mathrm{PT})$, and
- for every $\langle \pi; e \rangle \in \mathcal{P}$, $(\{\}, \mathrm{CT}) \vdash e : \mathrm{PT}(\pi)$.

$$\frac{\mathrm{TE} \vdash e : (\mathtt{unit} \to \tau)}{\mathrm{TE} \vdash \mathtt{spawn}\ e : \mathtt{unit}} \qquad (\tau\text{-}\mathbf{spawn})$$

$$\frac{\mathrm{TE} \vdash e : \tau\ \mathtt{event}}{\mathrm{TE} \vdash \mathtt{sync}\ e : \tau} \qquad (\tau\text{-}\mathbf{sync})$$

$$\frac{\mathrm{TE} \vdash e : \tau\ \mathtt{event}}{\mathrm{TE} \vdash (\mathbf{G}\, e) : \tau\ \mathtt{event}} \qquad (\tau\text{-}\mathbf{guard})$$

$$\frac{\forall\alpha.\alpha\ \mathtt{event} \succ \tau}{\mathrm{TE} \vdash \Lambda : \tau} \qquad (\tau\text{-}\mathbf{never})$$

$$\frac{\mathrm{TE} \vdash \kappa : \tau\ \mathtt{chan} \quad \mathrm{TE} \vdash v : \tau}{\mathrm{TE} \vdash \kappa!v : \mathtt{unit\ event}} \qquad (\tau\text{-}\mathbf{output})$$

$$\frac{\mathrm{TE} \vdash \kappa : \tau\ \mathtt{chan}}{\mathrm{TE} \vdash \kappa? : \tau\ \mathtt{event}} \qquad (\tau\text{-}\mathbf{input})$$

$$\frac{\mathrm{TE} \vdash ev : \tau'\ \mathtt{event} \quad \mathrm{TE} \vdash e : (\tau' \to \tau)}{\mathrm{TE} \vdash (ev \Rightarrow e) : \tau\ \mathtt{event}} \qquad (\tau\text{-}\mathbf{wrap})$$

$$\frac{\mathrm{TE} \vdash ev_1 : \tau\ \mathtt{event} \quad \mathrm{TE} \vdash ev_2 : \tau\ \mathtt{event}}{\mathrm{TE} \vdash (ev_1 \oplus ev_2) : \tau\ \mathtt{event}} \qquad (\tau\text{-}\mathbf{choice})$$

$$\frac{\mathrm{TE} \vdash ev : \tau\ \mathtt{event} \quad \mathrm{TE} \vdash v : (\mathtt{unit} \to \mathtt{unit})}{\mathrm{TE} \vdash (ev \mid v) : \tau\ \mathtt{event}} \qquad (\tau\text{-}\mathbf{abort})$$

Fig. 8. Other type inference rules for λ_{cv}

For **CML**, where `spawn` requires a ($\mathtt{unit} \to \mathtt{unit}$) argument, the process typing is $\mathrm{PT}(\pi) = \mathtt{unit}$ for all $\pi \in \mathrm{dom}(\mathcal{P})$.

7.3 Type Soundness

This section presents a statement of the soundness of the above type system with respect to the dynamic semantics of Section 6. To prove these results, I use the "syntactic" approach of Wright and Felleisen [WF91] (see [Rep92] for the proofs). The basic idea is to show that evaluation preserves types (also called *subject reduction*); then characterize run-time type errors (called "*stuck states*") and show that stuck states are untypable. This allows us to conclude that well-typed programs cannot go wrong.

The first step in this process is to show that the sequential evaluation relation preserves the types of expressions:

Theorem 17 Sequential type preservation. *For any type environment* TE*, expression* e_1 *and type* τ*, such that* $\mathrm{TE} \vdash e_1 : \tau$*, if* $e_1 \longmapsto e_2$ *then* $\mathrm{TE} \vdash e_2 : \tau$*.*

This result is then extended to concurrent evaluation and process typings.

Theorem 18 (Concurrent type preservation). *If a configuration $\mathcal{K}, \mathcal{P}$ is well-formed with*

$$\mathcal{K}, \mathcal{P} \Longrightarrow \mathcal{K}', \mathcal{P}'$$

and, for some channel typing CT,

$$\mathrm{CT} \vdash \mathcal{K}, \mathcal{P} : \mathrm{PT}$$

Then there is a channel typing CT' *and a process typing* PT'*, such that the following hold:*

- $\mathrm{CT} \subseteq \mathrm{CT}'$,
- $\mathrm{PT} \subseteq \mathrm{PT}'$, *and*
- $\mathrm{CT}' \vdash \mathcal{K}', \mathcal{P}' : \mathrm{PT}'$.
- $\mathrm{CT}' \vdash \mathcal{K}, \mathcal{P} : \mathrm{PT}'$.

With these results, it is fairly easy to show soundness results:

Theorem 19 (Syntactic soundness). *Let e be a program, with $\vdash e : \tau$. Then, for any $T \in \mathrm{Comp}(e)$, $\pi \in \mathrm{Procs}(T)$, with $\mathcal{K}_i, \mathcal{P}_i$ the first occurrence of π in T, there exists a* CT *and* PT*, such that*

$$\mathrm{CT} \vdash \mathcal{K}_i, \mathcal{P}_i : \mathrm{PT}$$

and $\mathrm{PT}(\pi_0) = \tau$. *And either*

- $\pi \Uparrow_T$, *or*
- $\pi \Downarrow_T v$ *and there exists an extension* CT' *of* CT *with* $(\{\}, \mathrm{CT}') \vdash v : \mathrm{PT}(\pi)$.

Theorem 20 (Soundness). *If e is a program with $\vdash e : \tau$, then for any computation $T \in \mathrm{Comp}(e)$ and any process ID $\pi \in \mathrm{Procs}(T)$, the following hold:*

(Strong soundness) *If* $\mathrm{eval}_T(\pi) = v$, *and $\mathcal{K}_i, \mathcal{P}_i$ is the first occurrence of π in T, then for any* CT *and* PT, *such that* $\mathrm{CT} \vdash \mathcal{K}_i, \mathcal{P}_i : \mathrm{PT}$ *and* $\mathrm{PT}(\pi_0) = \tau$, *there is an extension* CT' *of* CT, *such that* $(\{\}, \mathrm{CT}') \vdash v : \mathrm{PT}(\pi)$.

(Weak soundness) $\mathrm{eval}_T(\pi) \neq \mathrm{WRONG}$

8 Related Work

There are many approaches to concurrent language design (see either [AS83] or [And91] for an overview); our approach is an offshoot of the **CSP**-school of concurrent language design. **CML** began as a reimplementation of the concurrency primitives of **PML** [Rep88] in **SML/NJ**, but has evolved into a significantly more powerful language. **PML** in turn was heavily influenced by **amber** [Car86]. There have been other attempts at adding concurrency to various versions of **ML**. Most of these have been based on message passing ([Hol83], [Mat89], and [Ram90] for example), but there is at least one shared memory approach [CM90]. As we have shown in this paper, message passing fits very nicely into **SML**. It allows an applicative style of programming to be used most of the time; the state modifying operations are hidden in the thread and channel abstractions. **CML** extends the message passing paradigm by making synchronous operations first-class, which provides a mechanism for building user-defined synchronization abstractions.

Using concurrency to implement interactive systems has been proposed and implemented by several people. In [RG86], we made the argument that concurrency is vital for the construction of interactive programming environments. Pike (in [Pik89]) and Haahr (in [Haa90]) describe experimental window systems built out of threads and channels, but neither of these were fast enough for real use.

The semantics of **Facile** have been proposed as a model for **PML** in [PGM90], but no translation from **PML** to **Facile** is given. Independent work by Berry, Milner and Turner at the University of Edinburgh has resulted in an operational semantics for a small concurrent language, which includes the **PML** version of events [BMT92]. The semantics presented in this paper has some strong similarities with that of [BMT92], but λ_{cv} is a richer language; in particular, the language of [BMT92] does not include the `guard` and `wrapAbort` event value constructors found in **CML**. An earlier version of the semantics of first-class synchronous operations was presented in [Rep91b], and a more complete treatment of the version presented here can be found in [Rep92] (including extensions to cover features such as polling and exceptions).

9 Conclusions

We have described a higher-order concurrent language, **CML**, its use in real-world applications, and its formal semantics. **CML** supports first-class synchronous operations, which provide a powerful mechanism for communication and synchronization abstraction. Our experience with **CML** "in-the-field" and our measurements of the performance of the implementation show that **CML** is a practical tool for building real systems. We feel that **CML** is unique in that it combines a flexible high-level notation with good performance. In addition, it is well-defined, with a formal semantics.

CML is a stable system, and is freely available for distribution. The latest version of both **CML** and its manual are available via anonymous ftp in the `/pub` directory on `ftp.cs.cornell.edu`; for more information send electronic mail to `cml-bugs@cs.cornell.edu`. In addition, both **CML** and **eXene** will be included as part of the **SML/NJ** system (most likely in the fall of 1992).

Acknowledgements

Most of this work was done while the author was a graduate student at Cornell University, and was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862, and by the NSF under NSF grant CCR-89-18233. Anne Rogers helped with proof reading this paper.

References

[AB86] Abramsky, S. and R. Bornat. Pascal-m: A language for loosely coupled distributed systems. In Y. Paker and J.-P. Verjus (eds.), *Distributed Computing Systems*, pp. 163–189. Academic Press, New York, N.Y., 1986.

[Agh86] Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

[AM87] Appel, A. W. and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, vol. 274 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1987, pp. 301–324.

[And91] Andrews, G. R. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[AOCE88] Andrews, G. R., R. A. Olsson, M. Coffin, and I. Elshoff. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, **10**(7), January 1988, pp. 51–86.

[App92] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, New York, N.Y., 1992.

[AS83] Andrews., G. R. and F. B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, **15**(1), March 1983, pp. 3–43.

[BB90] Berry, G. and G. Boudol. The chemical abstract machine. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 81–94.

[BCJ+90] Birman, K., R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS system manual, version 2.0.* Department of Computer Science, Cornell University, Ithaca, N.Y., March 1990.

[BMT92] Berry, D., R. Milner, and D. N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992, pp. 119–129.

[Bor86] Bornat, R. A protocol for generalized occam. *Software – Practice and Experience*, **16**(9), September 1986, pp. 783–799.

[Bur88] Burns, A. *Programming in occam 2.* Addison-Wesley, Reading, Mass., 1988.

[Car86] Cardelli, L. Amber. In *Combinators and Functional Programming Languages*, vol. 242 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1986, pp. 21–47.

[CK92] Cooper, R. and C. D. Krumvieda. Distributed programming with asynchronous ordered channels in Distributed ML. In *Proceedings of the 1992 ACM SIGPLAN Workshop on ML and its Applications*, June 1992, pp. 134–148.

[CM90] Cooper, E. C. and J. G. Morrisett. Adding threads to Standard ML. *Technical Report CMU-CS-90-186*, School of Computer Science, Carnegie Mellon University, December 1990.

[Con86] Constable, R. *et al.* *Implementing Mathematics with The Nuprl Development System.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[CP85] Cardelli, L. and R. Pike. Squeak: A language for communicating with mice. In *SIGGRAPH '85*, July 1985, pp. 199–204.

[DHM91] Duba, B., R. Harper, and D. MacQueen. Type-checking first-class continuations. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, January 1991, pp. 163–173.

[FF86] Felleisen, M. and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing (ed.), *Formal Description of Programming Concepts – III*, pp. 193–219. North-Holland, New York, N.Y., 1986.

[GL91] George, L. and G. Lindstrom. Using a functional language and graph reduction to program multiprocessor machines. *Technical Report UUCS-91-020*, Department of Computer Science, University of Utah, October 1991.

[GMP89] Giacalone, A., P. Mishra, and S. Prasad. Facile: A symemetric integration of concurrent and functional programming. In *TAPSOFT'89 (vol. 2)*, vol. 352 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1989, pp. 184–209.

[GR91] Gansner, E. R. and J. H. Reppy. eXene. In *Proceedings of the 1991 CMU Workshop on Standard ML*, Carnegie Mellon University, September 1991.

[GR92] Gansner, E. R. and J. H. Reppy. A foundation for user interface construction. In B. A. Myers (ed.), *Languages for Developing User Interfaces*, pp. 239–260. Jones & Bartlett, Boston, Mass., 1992.

[Haa90] Haahr, D. Montage: Breaking windows into small pieces. In *USENIX Summer Conference*, June 1990, pp. 289–297.

[Har86] Harper, R. Introduction to Standard ML. *Technical Report ECS-LFCS-86-14*, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, August 1986.

[HDB90] Hieb, R., R. K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, June 1990, pp. 66–77.

[Hol83] Holmström, S. PFL: A functional language for parallel programming. In *Declarative programming workshop*, April 1983, pp. 114–139.

[KKR+86] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, July 1986, pp. 219–233.

[Kru91] Krumvieda, C. D. DML: Packaging high-level distributed abstractions in SML. In *Proceedings of the 1991 CMU Workshop on Standard ML*, September 1991.

[Kru92] Krumvieda, C. D. Expressing fault-tolerant and consistency-preserving programs in Distributed ML. In *Proceedings of the 1992 ACM SIGPLAN Workshop on ML and its Applications*, June 1992, pp. 157–162.

[LCJS87] Liskov, B., D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, November 1987, pp. 111–122.

[Lie86] Lieberman, H. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA'86 Proceedings*, September 1986, pp. 214–223.

[LS88] Liskov, B. and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 260–267.

[Mac84] MacQueen, D. B. Modules for Standard ML. In *Conference record of the 1984 ACM Conference on Lisp and Functional Programming*, July 1984, pp. 198–207.

[Mat89] Matthews, D. C. J. Processes for Poly and ML. In *Papers on Poly/ML, Technical Report 161*. University of Cambridge, February 1989.

[MT91] Milner, R. and M. Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Mass, 1991.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[Nye90] Nye, A. *Xlib Programming Manual*, vol. 1. O'Reilly & Associates, Inc., 1990.

[Oph92] Ophel, J. An introduction to the high-level language Standard ML, 1992. In this volume.

[Pau91] Paulson, L. C. *ML for the Working Programmer*. Cambridge University Press, New York, N.Y., 1991.

[Pel92] Pellerin, C. The concurrent constraint programming language taskell. Master's dissertation, School of Computer Science, McGill University, Montréal, Québec, Canada, January 1992.

[PGM90] Prasad, S., A. Giacalone, and P. Mishra. Operational and algebraic semantics for Facile: A symemetric integration of concurrent and functional programming. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, vol. 443 of *Lecture Notes in Computer Science*. Springer-Verlag, July 1990, pp. 765–780.

[Pik89] Pike, R. A concurrent window system. *Computing Systems*, **2**(2), 1989, pp. 133–153.

[Plo75] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**, 1975, pp. 125–159.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

[Rep88] Reppy, J. H. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988, pp. 250–259.

[Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Department of Computer Science, Cornell University, December 1989.

[Rep90a] Reppy, J. H. Asynchronous signals in Standard ML. *Technical Report TR 90-1144*, Department of Computer Science, Cornell University, August 1990.

[Rep90b] Reppy, J. H. *Concurrent programming with events – The Concurrent ML manual.* Department of Computer Science, Cornell University, Ithaca, N.Y., November 1990. (Last revised October 1991).

[Rep91a] Reppy, J. H. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991, pp. 293–305.

[Rep91b] Reppy, J. H. An operational semantics of first-class synchronous operations. *Technical Report TR 91-1232*, Department of Computer Science, Cornell University, August 1991.

[Rep92] Reppy, J. H. *Higher-order concurrency*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1992. Available as Technical Report TR 92-1285.

[RG86] Reppy, J. H. and E. R. Gansner. A foundation for programming environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, December 1986, pp. 218–227.

[SG86] Scheifler, R. W. and J. Gettys. The X window system. *ACM Transactions on Graphics*, **5**(2), April 1986, pp. 79–109.

[Ste78] Steele Jr., G. L. Rabbit: A compiler for Scheme. Master's dissertation, MIT, May 1978.

[Tof90] Tofte, M. Type inference for polymorphic references. *Information and Computation*, **89**, 1990, pp. 1–34.

[UNI86] University of California, Berkeley. *UNIX Programmer's Reference Manual (4.3bsd)*, 1986.

[Wan80] Wand, M. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, August 1980, pp. 19–28.

[WF91] Wright, A. and M. Felleisen. A syntactic approach to type soundness. *Technical Report TR91-160*, Department of Computer Science, Rice University, April 1991.

This article was processed using the LaTeX macro package with LLNCS style