510: Programming Languages

# Implementation of Concurrent ML

David Walker

## Implementing CML in SML/NJ

The implementation of CML consists of three major components:

- User-level threads.
- Communication channels.
- Synchronization on events.

## Implementing Threads

User-level threads are implemented as **co-routines** of the (user-level) scheduler.

- Synchronization primitives manage scheduler queues and hand-off.
- Threads can explicitly yield control back to the scheduler.
- Scheduler maintains a ready queue of runnable threads.

## Implementing Threads

Thus, concurrency is implemented by **interleaving** thread execution, rather than by running them in **parallel**.

- We ignore thread pre-emption (*e.g.*, by a timer interrupt).

- We ignore multi-processor scheduling.

Supporting parallelism requires **interlock** mechanisms to control access to shared resources (such as the ready queue).

## Implementing Threads

Co-routines may be implemented using **first-class continuations**.

- A **continuation** is essentially a **reified** control stack (as in the $C$ machine).
- The current continuation is **seized** using `callcc`.
- A continuation is **activated** using `throw`.

## Implementing Threads

A basic thread signature:

```
signature THREADS =
  sig
    val yield : unit -> unit
    val exit : unit -> 'a
    val fork : (unit -> unit) -> unit
  end
```

CML threads are similar, but include thread id's and `wait`.

## Implementing Threads

Basic ideas:

- Maintain a global, mutable **ready queue** of threads.
- Dispatch by dequeueing a thread and throwing to it.
- Yield grabs the current continuation, enqueues it, and dispatches.
- Fork grabs the current continuation, enqueues it, and invokes the thread body.

# Implementing Threads

```
structure Threads :> THREADS =
  struct
    structure Q = Queue
    val rdyQ : unit cont Q.queue =
          Q.mkQueue()
    fun dispatch () =
          throw (Q.dequeue rdyQ) ()
    val exit = dispatch
    ...
  end
```

## Implementing Threads

```
structure Threads :> THREAD =
  struct
    ...
    fun yield () =
      callcc (fn k =>
        Q.enqueue (rdyQ, k) ; dispatch ())
    ...
  end
```

## Implementing Threads

```
structure Threads :> THREAD =
  struct
    ...
    fun fork f =
      callcc (fn parent =>
        Q.enqueue (rdyQ, parent) ;
        (f ()) handle _ => () ;
        exit ())
  end
```

Exercise: implement a version of `fork` that favors the parent thread.

## Asynchronous Channels

Simple message-passing interface:

```
signature CHAN =
  sig
    type 'a chan
    val channel : unit -> 'a chan
    val send : ('a chan * 'a) -> unit
    val recv : 'a chan -> 'a
  end
```

## Asynchronous Channels

Main ideas:

- A channel consists of a pair of queues, one for senders, one for receivers.

- The implementation of (asynchronous) send and receive matches senders with receivers.

## Asynchronous Channels

```
structure ASynchChan :> CHAN =
  struct
    structure Q = Queue
    datatype 'a chan = CH of {
      sendQ : 'a Q.queue
      recvQ : 'a cont Q.queue
    }
    fun channel () =
        CH { sendQ = Q.mkQueue(),
             recvQ = Q.mkqueue() }
    ...
  end
```

## Asynchronous Channels

```
structure AsynchChan :> CHAN =
  struct
    ...
    fun send (CH { sendQ, recvQ }, msg) =
        if (Q.isEmpty recvQ) then
           Q.enqueue (sendQ, msg)
        else
           callcc (fn k =>
             Q.enqueue (rdyQ, k);
             throw (Q.dequeue recvQ) msg)
    ...
  end
```

## Asynchronous Channels

```
structure AsynchChan :> CHAN =
  struct
    ...
    fun recv (CH { sendQ, recvQ }) =
        if (Q.isEmpty sendQ) then
           callcc (fn k =>
             Q.enqueue (recvQ, k); dispatch())
        else
           Q.dequeue sendQ
  end
```

## Synchronous Channels

Main change: associate a thread with each entry in the `sendQ` to be resumed by the receiver.

```
datatype 'a chan =
  CH { sendQ : ('a * unit cont) Q.queue,
       recvQ : 'a cont Q.queue }
```

## Synchronous Channels

```
fun send (CH{sendQ, recvQ}, msg) =
    callcc (fn k =>
      if Q.isEmpty recvQ then
         Q.enqueue (sendQ, (msg, k)) ;
         dispatch ()
      else
         (Q.enqueue (rdyQ, k);
          throw (Q.dequeue recvQ) msg))
```

## Synchronous Channels

```
fun recv (CH{sendQ, recvQ}) =
    if Q.isEmpty sendQ then
       callcc (fn k => Q.enqueue (recvQ, k);
       dispatch ())
    else
       let val (msg, sender) =
           Q.dequeue sendQ
       in
           Q.enqueue (rdyQ, sender);
           msg
       end
```

## Simple Selective Communication

Consider a simple selective read-and-wrap operation:

```
val select :
  ('a chan * ('a -> 'b)) list -> 'b
```

In full CML `select` could be defined by

```
val select =
    sync o choose o
      (map (fn (c, f) =>
              wrap (readEvt c, f)))
```

We will implement `select` directly to illustrate how non-deterministic choice may be implemented.

## Simple Selective Communication

Main ideas:

- Build on the asynchronous send/receive given above (for simplicity).

- Selection on several empty channels enqueues a continuation on **each** channel to be invoked by a matching sender.

- The activated continuation must **remove** the pending receive from all other channels in the set.

## Simple Selective Communication

Revised definition of channels:

```
datatype 'a chan = CH of {
  sendQ : 'a Q.queue
  recvQ :
    (unit ref * (unit ref * 'a) cont) Q.queue
}
```

The `unit ref` serves as a **unique identifier** for the suspended thread. This is used to remove non-selected threads when a thread is selected.

## Simple Selective Communication

```
structure ASynchSelChan :> CHAN =
  struct
    ...
    fun send (CH{sendQ, recvQ}, msg) =
        callcc (fn k =>
          if Q.isEmpty recvQ then
             Q.enqueue (rdyQ, k);
             Q.enqueue (sendQ, msg);
             dispatch ()
          else
             let val (id, recvK) =
               Q.dequeue recvQ
             in
               Q.enqueue (rdyQ, k);
               throw recvK (id, msg)
             end)
    ...
  end
```

## Simple Selective Communication

```
structure ASynchSelChan :> CHAN =
  struct
    ...
    fun recv (CH{sendQ, recvQ}) =
        if Q.isEmpty sendQ then
           let val (_, msg) =
             callcc (fn k =>
               Q.enqueue (recvQ, (ref (), k));
               dispatch ())
           in
             msg
           end
        else
           Q.dequeue sendQ
    ...
  end
```

## Simple Selective Communication

```
fun select choices =
    let
        ...
    in
        case List.find pollCh choices
          of SOME (CH{sendQ,...}, act) =>
             act (Q.dequeue sendQ)
           | NONE =>
             let val ids =
               map (fn _ => ref ()) choices
             in let val (id, msg) =
               callcc (wait ids)
             in
               remove (id, ids) msg
             end
    end
```

## Simple Selective Communication

```
fun pollCh ({sendQ, ...}, _) =
    not(Q.isEmpty sendQ)
fun wait ids k =
    let fun blk (id, (CH{recvQ, ...}, _)) =
        Q.enqueue (recvQ, (id, k))
    in  ListPair.app blk (ids, choices) ;
        dispatch()
    end
```

## Simple Selective Communication

Remove the selected channel from the receive queues of the other choices, returning the wrapper of the selected channel:

```
fun remove (id, ids) =
  let fun del
        (id'::r1, (CH{recvQ, ...}, act)::r2) =
      if id=id' then
           (del (r1, r2); act)
      else
           (Q.delete
              (recvQ, fn (id'', _) =>
                             id'=id'');
            del (r1, r2))
  in
    del (ids, choices)
  end
```

## First-Class Events

How are events implemented? Recall the first implementation of `recv`:

```
fun recv (CH { sendQ, recvQ }) =
    if Q.isEmpty sendQ then
       callcc (fn k =>
         Q.enqueue (recvQ, k); dispatch())
    else
       Q.dequeue sendQ
```

## First-Class Events

This can be re-written as follows:

```
fun recv (CH { sendQ, recvQ }) =
    let fun recvEvt k =
        if Q.isEmpty sendQ then
           (Q.enqueue (recvQ, k); dispatch())
        else
           Q.dequeue sendQ
    in  callcc recvEvt  end
```

The helper function is the event . . . and `sync` is `callcc`!

# First-Class Events

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

fun recvEvt
  (CH { sendQ, recvQ } : 'a chan)
    : 'a event =
    fn k =>
      if Q.isEmpty sendQ then
         (Q.enqueue (recvQ, k) ; dispatch())
      else
         Q.dequeue sendQ

val sync = callcc
```

## First-Class Events

Using this representation, `wrap` is implemented as follows:

```
fun wrap (evt : 'a event) (f : 'a -> 'b) : 'b event =
    fn k : 'b cont =>
       throw k (f (callcc evt))
```

That is, when we `sync` on `wrap` $(e, f)$, we `sync` on $e$, apply $f$, and yield that value as result.

## Remaining Issues

Some remaining issues:

- Interactions between exceptions and continuations.
- Event combinators and synchronization.
- Pre-emption, true parallelism.