Programming Languages
The Implementation of Concurrent ML
Robert Harper
Spring, 1999

## 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 in SML/NJ

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`.

Slogan: continuations are the "goto's" of functional programming.



## Review: The $C$ Machine

Recall: the state of the $C$ machine is a pair $(k, e)$, where

- $k$ is a **control stack**, or **continuation**, representing the "rest" of the program after $e$.
- $e$ is the **current expression** being evaluated; it's value, if any, is passed to $k$ to complete evaluation.



## Review: The $C$ Machine

Continuations (control stacks) are compositions of frames:

$$\begin{array}{llcl} \textit{continuations} & k & ::= & \bullet \mid f \triangleright k \\ \textit{frames} & f & ::= & \Box(e) \mid v(\Box) \mid \ldots \end{array}$$

Think of a frame as a pending step of evaluation; think of a continuation as a sequence of pending steps.



## Review: The $C$ Machine

Example instructions:

$$\begin{array}{rcl} (k, e_1(e_2)) & \mapsto & (\Box(e_2) \triangleright k, e_1) \\ (\Box(e_2) \triangleright k, v_1) & \mapsto & (v_1(\Box) \triangleright k, e_2) \\ (v_1(\Box) \triangleright k, v_2) & \mapsto & (k, [v_1, v_2/f, x]e) \end{array}$$

where $v_1 = \texttt{fun}\, f\,(x)\; \texttt{is}\; e\; \texttt{end}$.

## First-Class Continuations

In a state $(k, e)$, we call $k$ the **current continuation**. It is a representaton of the work that remains to be done, once $e$ is completed. It takes as argument the value of $e$.

To implement threads we add two additional mechanisms to the language:

- The ability to **seize** $k$ as a data value that may be stored in reference cells, passed as results, *etc.*.
- The ability to **install** a continuation $k$ as the current continuation, effecting a "jump" to the state represented by $k$ with a given resumption value.



## First-Class Continuations

Signature of continuations in SML/NJ:

```
signature CONT =
  sig
    type 'a cont
    val callcc : ('a cont -> 'a) -> 'a
    val throw : 'a cont -> 'a -> 'b
  end
```



## First-Class Continuations

Informal description:

- The type `'a cont` is the type of continuations expecting a value of type `'a`.
- The expression `callcc`$(v)$ passes the current continuation as the argument to $v$, and continues execution. **The current continuation is duplicated.**
- The expression `throw`$(v_1$,$v_2)$ installs the continuation represented by $v_1$ and passes it the value $v_2$. **The current continuation is abandoned.**



## First-Class Continuations

Formally, we extend MinML with values of the form `cont`$(k)$, where $k$ is a continuation.

The $C$ machine is extended with the following instructions:

$$\begin{array}{rcl}
(k, \texttt{callcc}(e)) & \mapsto & (\texttt{callcc}(\square) \triangleright k, e) \\
(\texttt{callcc}(\square) \triangleright k, v) & \mapsto & (k, v(k)) \\
\\
(k, \texttt{throw}(e_1\texttt{,}e_2)) & \mapsto & (\texttt{throw}(\square\texttt{,}e_2) \triangleright k, e_1) \\
(\texttt{throw}(\square\texttt{,}e_2) \triangleright k, v_1) & \mapsto & (\texttt{throw}(v_1\texttt{,}\square) \triangleright k, e_2) \\
(\texttt{throw}(\texttt{cont}(k')\texttt{,}\square) \triangleright k, v) & \mapsto & (k', v)
\end{array}$$

## Type Safety

Define typing for continuations as follows:

- $\vdash \texttt{cont}(k) : \tau\,\texttt{cont}$ iff $\vdash k : \tau\,\texttt{stack}$.
- $\vdash \bullet : \texttt{int}\,\texttt{stack}$.
- $\vdash f \triangleright k : \tau\,\texttt{stack}$ if $f : (\tau', \tau)\,\texttt{frame}$ and $k : \tau'\,\texttt{stack}$.



## Type Safety

Define typing for frames as follows:

- $\vdash \Box(e_2) : (\tau_2 \to \tau, \tau)\,\texttt{frame}$, if $\vdash e_2 : \tau_2$.
- $\vdash v_1(\Box) : (\tau_2, \tau)\,\texttt{frame}$, if $\vdash v_1 : \tau_2 \to \tau$.
- *etc.*



## Type Safety

**Lemma** If $\vdash k : \tau\,\texttt{stack}$, and $\vdash v : \tau$, then $\vdash [v/\Box]k : \texttt{int}$.

**Lemma** If $v : \tau\,\texttt{cont}$, then $v = \texttt{cont}(k)$ with $\vdash k : \tau\,\texttt{stack}$.



## Type Safety

Define $(k, e)$ to be **well-typed** iff

- $\vdash k : \tau\,\texttt{stack}$ and
- $\vdash e : \tau$.

**Theorem** If $(k, e) \mapsto (k', e')$ and $(k, e)$ is well-formed, then $(k', e')$ is well-formed.

**Theorem** If $(k, e)$ is well-formed, then either $k = \bullet$ and $e$ is a value, or $(k, e) \mapsto (k', e')$ for some $k'$ and $e'$.

## Short-Circuiting with Continuations

One use of continuations is to effect an early return from a function.

```
fun prod xs =
    let fun loop (nil, res) = res
          | loop (h::t, res) = loop (t, h*res)
    in  loop (xs, 1) end
```



## Short-Circuiting with Continuations

Would like to “short-circuit” the result if some element is zero.

```
fun prod xs =
    callcc (fn exit =>
      let fun loop (nil, res) = res
            | loop (0::_, _) = throw exit 0
            | loop (h::t, res) = loop (t, h*res)
       in loop (xs, 1) end)
```



## Wrapping a Continuation with a Function

Problem: pre-composition of a continuation with a function:

```
val wrap_cont : 'a cont -> ('b -> 'a) -> 'b cont
```

Given $k$ and $f$, want to create $k'$ such that

$$\texttt{throw}\ k'\ v \cong \texttt{throw}\ k\ (f(v))$$

How to create $k'$?



## Wrapping a Continuation with a Function

The only way to create a continuation is with `callcc`.

- Must seize the appropriate continuation, namely

  $$f(\square) \triangleright \texttt{throw(cont(}k\texttt{),}\square\texttt{)} \triangleright k_0,$$

  where $k_0$ is the current continuation. We will use `callcc` to create this continuation.

- Must yield this continuation as result of the function. We will use `callcc` a second time to accomplish this.

## Wrapping a Continuation with a Function

Hold onto your hats!

```
fun wrap_cont k f =
    callcc (fn ret =>
      throw k (f (callcc (fn k' => throw ret k'))))
```



## Exceptions from Continuations

We can also "short circuit" with exceptions:

```
exception Exit
fun prod xs =
    let fun loop (nil, res) = res
          | loop (0::_, _) = raise Exit
          | loop (h::t, res) = loop (t, h*res)
    in loop (xs, 1) end
    handle Exit => 0
```

What is the relationship between the two?



## Simulating Exceptions Using Continuations

General idea:

- Global, shared reference cell containing the current exception handler, a function of type `exn cont`.
- The `handle` expression saves the current handler and installs one that passes control to the exception handler for the duration of evaluation of the body.
- The `raise` expression retrieves the current handler and throws the exception value to it.



## Simulating Exceptions

Current handler:

```
val handler =
    ref (... initialized to print "Uncaught exception.\n" ...)
```

Implementation of `raise`:

```
fun raise exc = throw (!handler) exc
```

## Simulating Exceptions

Implementation of $e$ `handle` $v$, where $v$ has type `exn->'a` and $e$ has type `'a`:

```
let val old_handler = !handler
    val new_handler =
      callcc (fn ret =>
        let val x = callcc (fn k => throw ret k)
        in  handler := old_handler ; v x end)
    val _ = handler := new_handler
    val x = e
    val _ = handler := old_handler
in  x  end
```



## 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, enqueus 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
                 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.



## Summary

It is relatively straightforward to implement a user-level threads package with communication primitives using first-class continuations.

Some remaining issues:

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