Programming Languages
Concurrent ML
Robert Harper
Spring, 1999

Concurrent Programming

Concurrency in a PL is the ability to execute several programs "simultaneously" or "in parallel" or . . . "concurrently".

The **essence** of concurrency is **indeterminacy** in the operational semantics.

- Provide many choices of what to do next, *i.e.*multiple **threads of control**.
- Leave unspecified how the choice is made, *i.e.*how threads are **scheduled**.

Concurrent Programming

The principal **benefits** of concurrent programming are

- Flexibility in building systems from components.
- Concise expression of programming idioms such as demand-driven computation.
- Convenient handling of synchronization with external events (such as I/O).
- Scalability to distributed programming environments (with multiple processors).

Concurrent Programming

The principal **drawback** of concurrent programming is that it is tricky to manage indeterminacy.

- Must program "against" an arbitrarily demonic scheduler to ensure robustness.
- Interactions between threads can be remarkably subtle and difficult to debug by trial-and-error. Strong reasoning methods are essential to success.

Serializability

The principal technique for handling indeterminacy is to ensure that **any** concurrent execution of a concurrent program is equivalent to **some** serial execution.

But what is a concurrent execution? What is a serial execution?

Key concept: **interleaving** semantics.

- Think of a concurrent execution as an arbitrary interleaving of **basic steps** of computation.
- If the basic steps are **atomic**, then the execution is serializable.

Serializability

Example, in pseudo-code:

```
val x = ref 0;
fun inc r = (r := !r + 1);
(* concurrent execution of two expressions *)
val _ = (inc x) || (inc x)
(* what is the value of x here? *)
```


Serializability

In any serial execution, the value of `x` will be `2`.

- Both threads increment `x`.
- Choose either order in which to do the increments.
- Must always execute both increments.

Synchronization

The concurrent execution may or may not be serializable!

- Typically the increment is performed in three steps:

  1. Fetch `!r`.
  2. Add `1` to it.
  3. Store result back into `r`.

- If these basic steps can be interleaved, the execution not serializable!

Serializability

A non-serializable interleaving:

- Thread 1 fetches `!r`.
- Thread 2 fetches `!r`.
- Thread 2 adds 1 and stores result in `r`.
- Thread 1 adds 1 and stores result in `r`.

The result is that `x` has value `1`, which does not correspond to **any** serial execution!

Synchronization

Fundamentally, **something** must be (assumed to be) atomic.

- In low-level languages such as C or assembler, word-sized loads and stores are often atomic.
- Processors usually provide more complex atomic instructions such as `fetch-and-add`, `compare-and-swap`, or `load-locked` / `store-conditional`.

Synchronization

Using these primitives, one can build higher-level atomic operations.

- Semaphores, locks.
- Condition variables.
- Channels, monitors, mailboxes, ivar's, mvar's, ...

Synchronization

This approach is OK for **concrete** languages, which are very close to the hardware (and therefore not very portable).

For **abstract** languages it is preferable to simply start with a high-level atomicity mechanism.

- In Java the fundamental mechanism is a **synchronized** object, which is the classical notion of a **monitor**.
- In CML the fundamental mechanism is **message passing** over a **typed channel**.

Deadlock and Livelock

Serializability is not the only issue in concurrent programming!

- **Deadlock**: no progress can be made because two threads are each waiting for the other to complete an atomic operation.

- **Livelock**: a thread is not chosen to execute by the scheduler, even though it can make progress.

One source of livelock is an **unfair** scheduler that simply refuses to execute some thread that can make progress. We normally assume that the schedule is **fair**, meaning that it cannot bypass a ready thread forever.

Deadlock and Livelock

Deadlock, and other forms of livelock, must be avoided by the programmer.

- The possibility of deadlock cannot be detected statically!

- Usually solved by adherence to a synchronization protocol that avoids circular dependencies.

Synchronization in Java

The Java approach to controlling concurrent interaction is to **synchronize** access to the **state** of an object.

- Make instance variables private.

- Access their state via **synchronized** methods.

Synchronized methods are executed **atomically** to avoid non-serializable executions.

Synchronization in Java

```
public class Counter {
  private int value;
  public Cell (int init) {
    value = init;
  }
  public synchronized int get {
    return value;
  }
  public synchronized void inc {
    value = value + 1;                // executed atomically
  }
}
```

Synchronization in CML

Shared-memory approaches are a poor fit for value-oriented languages such as ML in which assignment is rare.

Instead we employ a more abstract mechanism for concurrent interaction called **message-passing**.

- Threads share dynamically-created **channels** carrying values of an **arbitrary** type.
- Threads synchronize by **send** and **receive** operations on channels.

Threads in CML

A new thread is created using `spawn`:

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

The new thread applies the given function to () to begin execution. The thread terminates when this function returns; its storage is automatically recycled by the garbage collector.

The `spawn` returns to the parent thread with a unique identifier for the child.

Channels

Channels carry values of an arbitrary type:

```
type 'a chan
```

Channels are created by the `channel` primitive:

```
val channel : unit -> 'a chan
```

Unused channels are automatically recycled by the garbage collector.

Synchronous Send and Receive

Synchronous (blocking) operations for communicating on channels:

```
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
```

The `send` operation blocks its thread until the message has been received; the `recv` blocks its thread until a matching send has occurred.

The sender and receiver **synchronize** on the channel. This is sometimes called a **rendezvous**.

Example: Mutable Cells as Threads

Idea: think of a mutable cell as a **server** that accepts requests to set and get its value.

```
signature CELL = sig
  type 'a cell
  val new : 'a -> 'a cell     (* constructor *)
  val get : 'a cell -> 'a
  val set : 'a cell * 'a -> unit
end
```


Example: Mutable Cells as Threads

```
structure Cell :> CELL = struct
  datatype 'a request = GET | PUT of 'a
  datatype 'a cell =
    CELL of { reqCh : 'a request chan, replyCh : 'a chan }

  fun new x = ...

  fun get (CELL {reqCh, replyCh}) =
      (send (reqCh, GET); recv (replyCh))
  fun set (CELL {reqCh, replyCh}, x) =
      (send (reqCh, PUT x))
end
```


Example: Mutable Cells as Threads

How is the constructor implemented?

```
fun new x =
    let
        val reqCh = channel ()
        val replyCh = channel ()
        fun server x =
            (case recv.reqCh
               of GET => (send (replyCh, x); server x)
                | PUT x' => server x')
    in
        spawn (fn () => server x);
        CELL {reqCh=reqCh, replyCh=replyCh}
    end
```


Example: Mutable Cells as Threads

Observations about this implementation of cells:

- No mutable storage is used! The state is in the recursion (just like hardware).
- The request/reply protocol is hidden behind the `CELL` abstraction. Client cannot accidentally `recv` from `replyCh` without first sending a `GET` request.
- Synchronous `send` operation ensures that the cell operations are atomic.

Example: Streams as Threads

Recall the view of streams as **suspended computations** that produce values on demand, becoming dormant in between requests.

We will take this one step further and regard streams as **threads** that execute concurrently with the client.

- The thread scheduler handles suspending and awakening threads.
- Synchronous `send` and `recv` control production of elements of the stream.

This style of programming is called a **dataflow network**.

Example: Streams as Threads

Example: the stream of natural numbers:

```
fun nats_from start =
    let
        val ch = channel ()
        fun loop i = (send (ch, i); loop (i+1))
    in
        spawn (fn () => loop start); ch
    end
```

Successive `recv`'s on the returned channel yield successive naturals, starting with `start`.

Example: Streams as Threads

Filter out multiples of a given natural number:

```
fun filter (p, ch) =
    let
        val out = channel ()
        fun loop () =
            let val i = recv ch
            in  if ((i mod p) <> 0) then send (out, i) else ();
                loop ()
            end
    in
        spawn loop; out
    end
```


Example: Streams as Threads

Sieve of Eratosthenes:

```
fun sieve () =
    let
        val primes = channel ()
        fun head ch =
            let val p = recv ch
            in  send (primes, p);
                head (filter (p, ch))
            end
    in
        spawn (fn () => head (nats_from 2));
        primes
    end
```

Example: Streams as Threads

Take the first `n` values from a channel:

```
fun take n ch =
    let
        fun loop 0 acc = acc
          | loop n acc = loop (n-1) (recv ch :: acc)
    in
        rev (loop n)
    end
```


Example: Streams as Threads

Test harness:

```
fun test_primes n () =
    let
        val primes = sieve ()
    in
        print (toString (take n primes))
    end
```


Aside: Using CML

To invoke CML, type `cml-cm` at the command prompt.

Use the compilation manager with `sources.cm` as follows:

```
Group is
        cml.cm
        your_file.sml
```

Then type

```
CML.make();
```


Aside: Using CML

To start a CML-based program, use

```
val RunCML.doit :
    (unit -> unit) * Time.time option -> OS.Process.status
```

For example,

```
RunCML.doit (test_primes 10, NONE);
```

Example: Flip-Flops as Dataflow Networks

Modelling a digital circuit as a dataflow network:

- Model a **wire** as a stream of boolean values (a perfect wave-form).
- Model a **gate** as a **stream transducer**.

Example: RS latch built from two nor gates.

Example: Flip-Flops

Constantly high or low wires:

```
fun constantly b =
    let
        val C = channel()
        fun loop () = (send (C, b); loop())
    in
        spawn loop ; C
    end
```


Example: Flip-Flops

Nor-gate, with unit delay:

```
fun nor_gate (A, B) =
    let
        val C = channel()
        fun nor (a, b) = not (a orelse b)
        fun loop (a, b) =
            (send (C, nor (a, b)); loop (recv A, recv B))
    in
        spawn (fn () => loop (false, false)); C
    end
```


Example: Flip-Flops

Split a wire:

```
fun split A =
    let
        val B1 = channel()
        val B2 = channel()
        fun loop a =
            (send (B1, a); send (B2, a); loop (recv A))
    in
        spawn (fn () => loop (recv A)); (B1, B2)
    end
```

Example: Flip-Flops

Copy, with unit delay:

```
fun copy (A, B) =
    let
        fun loop a = (send (B, a); loop (recv A))
    in
        spawn (fn () => loop false)
    end
```


Example: Flip-Flops

Build an RS latch, using backpatching to implement loopback:

```
fun RS_ff (S, R) =
    let
        val X' = channel ()
        val Y' = channel ()
        val X = nor_gate (S, Y')
        val Y = nor_gate (X', R)
        val (Y1, Y2) = split Y
    in
        copy (X, X'); copy (Y1, Y'); Y2
    end
```


Example: Flip-Flops

Test harness:

```
fun test_ff n () =
    let
        val R = constantly false
        val S = constantly true
        val Y = RS_ff (S, R)
    in
        print (toString (take n Y))
    end
```


Summary

The synchronous fragment of CML provides:

- Multiple threads of control.
- Dynamically-allocated communication channels.
- Synchronous send and receive operations on channels.

Next we will discuss asynchronous CML and first-class events.

Programming Languages
Concurrent ML
Robert Harper
Spring, 1999

Synchronous CML

Spawn a new thread:

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

Communication channels:

```
type 'a channel
val channel : unit -> 'a channel
```


Synchronous CML

Synchronous send and receive:

```
val send : 'a channel * 'a -> unit
val recv : 'a channel -> 'a
```

The `recv` operation blocks until data is available. The `send` operation blocks until the receiver has `recv`'d the data.

Example: Fibonacci Network

A dataflow network to calculate the Fibonacci numbers using synchronous CML.

- Allocate all channels.

- Wire them up using transducers.

The result will turn out to be excessively brittle, leading to the asynchronous fragment of CML.

Example: Fibonacci Network

The Fibonacci network:

```
fun fib_net () =
    let
        val out = channel() and c1 = channel()
        and c2 = channel() and c3 = channel()
        and c4 = channel() and c5 = channel()
    in
        delay 0 (c4, c5); split (c2, c3, c4);
        add (c3, c5, c1); split (c1, c2, out);
        send (c1, 1); out
    end
```


Example: Fibonacci Network

Synchronous combination of two channels:

```
fun synch_combine f (in1, in2, out) =
    forever
      ()
      (fn () => send (out, f (recv in1, recv in2)))

val add = synch_combine (op +)
```


Example: Fibonacci Network

Buffered copy of one channel to another:

```
fun delay init (inp, out) =
    forever
      init
      (fn x => (send (out, x); recv inp))
```


Example: Fibonacci Network

Synchronous split of one channel into two:

```
fun synch_split (inp, out1, out2) =
    forever
      ()
      (fn () =>
       let
           val x = recv inp
       in
           send (out1, x); send (out2, x)
       end)
```

Deadlock in the Fibonacci Network

It is **crucial** that `combine` read its channels in the same order that `split` writes them!

Consider the following variant of `split`:

```
fun swap_split (inp, out1, out2) =
    forever
      ()
      (fn () =>
       let
           val x = recv inp
       in
           send (out2, x); send (out1, x)
       end)
```


Deadlock in the Fibonacci Network

What happens to the Fibonacci network? **It deadlocks!**

- `combine` waits for input on its first input channel before reading the second.
- `split` writes to the second output channel, waiting for it to be read before writing the first.

No progress can be made!

Deadlock in the Fibonacci Network

The problem is that there is **too little** indeterminacy in the network.

The solution is to introduce additional **indeterminacy** into the network by using **non-blocking**, or **asynchronous**, communication.

We will consider **reader-side** and a **writer-side** solutions to the deadlock problem.

Asynchronous Read

Avoid committing to the order of arrival of messages.

- Set up paths for either arrival order.
- Non-deterministically select whichever is appropriate.

Asynchronous Read

This requires three new concepts:

- **Read events**, for the arrival of data on a channel.
- Non-deterministic selection of available events.
- Wrapping an event with a post-processor.

Asynchronous Read

```
fun asynch_combine f (in1, in2, out) =
    forever
      ()
      (fn () =>
       let
           val (a, b) =
               select
                 [wrap (recvEvt in1,
                    fn a => (a, recv in2)),
                  wrap (recvEvt in2,
                    fn b => (recv in1, b))]
       in
           send (out, f (a, b))
       end)
```


Asynchronous Read

The operation **recvEvt** yields an **event** representing the arrival of a message on a channel.

Synchronizing on a read event completes the read operation, yielding the value on the channel.

The operation **wrap** defines a **post-processor** to be executed when the event arrives. The wrapper is applied to the value of the event when synchronized.

Asynchronous Read

The operation **select** synchronizes on **one** of a finite list of events. Selection order is unspecified (indeterminate).

The **select** operation is defined as follows:

```
val select = sync o choose
```

Asynchronous Read

An **event combinator**:

```
val choose : 'a event list -> 'a event
```

Synchronization on an event:

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

Synchronous operations:

```
val recv = sync o readEvt
val send = sync o sendEvt
```


Asynchronous Write

Similarly, we may avoid choosing an order in which to write channels in the `split` operation using **write events**.

- Set up send to both channels, followed by a send to the other.
- Non-deterministically select whichever is available.

Asynchronous Write

```
fun asynch_split (inp, out1, out2) =
    forever
      ()
      (fn () =>
       let
           val x = recv inp
       in
           select
             [wrap (sendEvt (out1, x),
                fn () => send (out2, x)),
              wrap (sendEvt (out2, x),
                fn () => send (out1, x))]
       end)
```


Asynchronous Write

In practice it is generally best to use **both** asynchronous read and write to avoid deadlock situations.

Doing so imposes synchronization overhead that can be avoided (to some extent) with very careful programming.

But **any** channel rendezvous involves synchronization, so there is little to be gained by avoiding **select**.

First-Class Channels

It is often convenient, even essential, to pass channels as values between threads.

Example: swap channels.

```
signature SWAP_CHAN =
    sig
        type 'a swap_chan
        val new : unit -> 'a swap_chan
        val swap : 'a swap_chan * 'a -> 'a CML.event
    end
```

Matching swap operations exchange values.

Example: Swap Channels

A non-solution:

```
type 'a swap_chan = 'a chan
val new = channel
fun swap (ch, msg_out) =
    select
      [wrap (recvEvt ch,
        fn msg_in => (send (ch, msg_out); msg_in)),
       wrap (sendEvt (ch, msg_out),
        fn () => recv ch)]
```


Example: Swap Channels

Problem: what if several threads swap on the same channel more or less simultaneously?

The swapped values could get confused between threads because of the shared channel! There is no guarantee that threads swap "matching pairs".

The culprit is the use of a shared channel for both request and response.

Example: Swap Channels

Solution: allocate a private channel for each response.

```
datatype 'a swap_chan = SC of ('a * 'a chan) chan
fun new () = SC (chanel ())
fun swap (SC ch, msg_out) = ...
```

Example: Swap Channels

```
fun swap (SC ch, msg_out) =
    guard
      (fn () =>
        let
            val in_ch = channel ()
        in
            choose
              [wrap (recvEvt ch,
                 fn (msg_in, out_ch) => (send (out_ch, msg_out); msg_in)),
               wrap (sendEvt (ch, (msg_out, in_ch)),
                 fn () => recv in_ch)]
        end
```


Example: Swap Channels

The **guard** construct provides **pre-processing** before setting up an event.

In this case we allocate a private channel for the reply message before choosing between a receive and a send event.

The arrival of the receive event is handled by sending the outgoing message on the received reply channel, returning the received data.

The arrival of the send event is handled by performing a synchronous receive on the reply channel.

Example: Merge Sort

Overall structure:

- If the input is "short", sort it and return.
- Otherwise,
  - Divide the list into two roughly equal parts.
  - Recursively sort the parts **concurrently**.
  - Merge the sorted parts to obtain the sorted list.

Represent the input and output as channels.

Example: Mergesort

```
fun merge_sort () =
    let
        val ch = channel ()
        fun sort () = ...
    in
        spawn sort; ch
    end
```

Example: Mergesort

```
fun sort () =
    (case recv ch
       of NONE => ()
        | SOME v1 =>
           (case recv ch
              of NONE => send (ch, SOME v1)
               | SOME v2 => ...))
```


Example: Mergesort

```
let
    val ch1 = merge_sort ()
    val ch2 = merge_sort ()
in
    send (ch1, SOME v1); send (ch2, SOME v2);
    split (ch, ch1, ch2); merge (ch1, ch2, ch)
end);
send (ch, NONE))
```


Example: Mergesort

Split a channel into two by alternation:

```
fun split (inp, out1, out2) =
    let
        fun loop (NONE, _, _) =
            (send (out1, NONE); send (out2, NONE))
          | loop (SOME v, out1, out2) =
            (send (out1, SOME v);
             loop (recv inp, out2, out1))
    in
        loop (recv inp, out1, out2)
    end
```


Example: Mergesort

```
fun merge (in1, in2, out : int option chan) =
let
  fun merge' (from1, from2) =
      case (from1, from2)
        of (NONE, NONE) => send (out, NONE)
         | (_, NONE) =>
            (send (out, from1); merge' (recv in1, from2))
         | (NONE, _) =>
            (send (out, from2); merge' (from1, recv in2))
         | (SOME a, SOME b) =>
            if (a < b) then
               (send (out, from1); merge' (recv in1, from2))
            else
               (send (out, from2); merge' (from1, recv in2))
in
  merge' (recv in1, recv in2)
end
```

Example: Lock Server

Provide a service for allocating **locks** to control access to a shared resource.

- Must **acquire** a lock before using the resource.
- Must **release** the lock when done with it.
- Can create **new** locks at will.

Twist: lock acquisition can **time out**, abandoning request for the lock.

Example: Lock Server

```
signature LOCK_SERVER =
    sig
        type lock
        type server =
            { new : unit -> lock,
              acquire : lock -> unit CML.event,
              release : lock -> unit }
        val new_server : unit -> server
    end
```


Example: Lock Server

Typical client code:

```
   val {new,acquire,release} = LockServer.new_server()
⋮
   select
     [wrap (timeOutEvt time_limit, fn () => NONE),
      wrap (acquire lock, SOME)]
⋮
   release lock
```


Example: Lock Server

Design issues:

- Clients must obey the locking protocol for access to the resource.
- Client synchronizes on lock acquisition event to avoid blocking unnecessarily.
- Client may abandon an acquisition request, *e.g.* due to time-out.

CML Summary

Threads:

```
type thread_id
val spawn : (unit -> unit) -> thread_id
val exit : unit -> 'a
val yield : unit -> unit
val threadWait : thread_id -> unit event
```


CML Summary

Events:

```
type 'a event
val choose : 'a event list -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val withNack : (unit event -> 'a event) -> 'a event
val always : 'a -> 'a event
val never : 'a event
val sync : 'a event -> 'a
```


CML Summary

Channels:

```
type 'a chan
val channel : unit -> 'a chan
val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
```


CML Summary

Plus more:

- System events: asynchronous I/O, timers.
- Libraries: condition variables, mvar's, ivar's, *etc.*.
- Startup: **RunCML**, *etc.*.