510: Programming Languages

Concurrent Programming in ML

David Walker

ML Concurrency Mechanisms

Explicit parallelism:

- **Spawn** a new thread of control.
- Wait for thread completion.

Communication via **channels**:

- Dynamic channel creation.
- Asynchronous send and receive of arbitrary values.

ML Concurrency Mechanisms

Synchronization via **events**.

- Asynchronous operations create completion events.
- Event combinators build up complex events.
- Synch on an event to await its completion (and obtain result).

Spawning Threads in CML

A new thread is created using **spawn**:

```
type runnable = unit -> unit
val spawn : runnable -> thread_id
```

The new thread is started by applying the given **runnable** to ().

Channels

Channels carry values of an arbitrary type:

```
type 'a chan
```

Channels are created by the `channel` primitive:

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

Disused channels are 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

Model 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

Some observations:

- No mutable storage is required! The state is in the recursion, just like hardware.

- The request/reply protocol is hidden behind the `CELL` abstraction.

- The cell operations are atomic because `send` is synchronous.

Dataflow Networks

A **stream** is an infinite sequence of values.

Idea: a channel represents a stream.

- Read from a stream by receiving on a channel.

- Generate a new stream by allocating a new channel and spawning a thread that sends the appropriate values on the channel.

- Write functions from streams to streams by spawning threads that read input channels and write on output channels.

This is called a **dataflow**, or **Kahn-MacQueen**, network.

Example: Flip-Flops as Dataflow Networks

Modelling a digital circuit as a dataflow network:

- Model a **wire** as a stream of boolean values (a Boolean-valued channel).

- Model a **gate** as a **stream transducer** (a thread).

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
```

Example: Fibonacci Network

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

- Allocate all channels.

- Wire them up using transducers.

The structure turns out to be excessively brittle.

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 forever init f =
  let fun loop s = loop (f s) in
    ignore (spawn (fn () => loop init))
  end

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 **selective** communication.

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

Selective Read

Avoid committing to the order of arrival of messages.

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

Selective 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.

Selective Read

```
fun nondet_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)
```

Selective 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.

Selective 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
```

Selective 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
```

Selective 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.

Selective 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)
```

Selective Write

In practice it is generally best to use **both** selective 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**.

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.*.