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).

Channels

Channels carry values of an arbitrary type:

```ocaml
type 'a chan
```

Channels are created by the channel primitive:

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

Disused channels are recycled by the garbage collector.

Spawning Threads in CML

A new thread is created using spawn:

```ocaml
type runnable = unit -> unit
val spawn : runnable -> thread id
```

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

Synchronous Send and Receive

Synchronous (blocking) operations for communicating on channels:

```ocaml
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.

```ml
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

How is the constructor implemented?

```ml
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

```ml
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

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.

Concurrent MinML

Extend the abstract syntax of MinML as follows:

```
Types τ ::= τ event

Expr's e ::= sync(e) | channel() | spawn(e) |
          always(e) | never | recv(e) | send(e1,e2) |
          choose(e1,e2) | wrap(e1,e2) | guard(e)

Prog's t ::= l:e | p1 || p2
```

Concurrent MinML

A program is a parallel composition of labelled expressions, called threads.

Assumption: parallel composition is commutative and associative.

- Nesting structure of parallelism does not matter.
- Ordering of threads does not matter.
Dynamic Semantics

State: \( (C, p) \), where

- \( C \) is a set of channels;
- \( p \) is a well-formed program.

Initial state: \( (\emptyset, p) \).

Final state: \( (C, l_1:() \parallel \cdots \parallel l_n:O) \).

Transitions: \( (C, p) \xrightarrow{\text{CML}} (C', p') \).

Dynamic Semantics

A program context is a program with one or more empty “slots” for expressions:

\[
\text{Context } P \vdash l:e \mid l:e \parallel P_1 \parallel P_2
\]

We may fill a program context by replacing a labelled hole with a labelled expression: \( P[l_1 = e_1, \ldots, l_n = e_n] \).

The holes in a program context serve as focal points of evaluation.

Static Semantics

\[
\begin{align*}
\Gamma & \vdash e : \text{unit} & \Gamma & \vdash l:e \text{ ok} & \Gamma & \vdash p_1 \text{ ok} & \Gamma & \vdash p_2 \text{ ok} \\
\Gamma & \vdash e : \tau \text{ event} & \Gamma & \vdash \text{always}(e) : \tau \text{ event} & \Gamma & \vdash e : \tau \text{ event} & \Gamma & \vdash \text{never} : \tau \text{ event} \\
\Gamma & \vdash e : \tau \text{ chan} & \Gamma & \vdash \text{recv}(e) : \tau \text{ event} & \Gamma & \vdash \text{send}(e_1, e_2) : \text{unit event} \\
\Gamma & \vdash e_1 : \tau \text{ event} & \Gamma & \vdash e_2 : \tau \text{ event} & \Gamma & \vdash \text{choose}(e_1, e_2) : \tau \text{ event} \\
\Gamma & \vdash e_1 : \tau \rightarrow \tau' \text{ event} & \Gamma & \vdash \text{wrap}(e_1, e_2) : \tau' \text{ event} \\
\Gamma & \vdash e : \text{unit} \rightarrow \tau \text{ event} & \Gamma & \vdash \text{guard}(e) : \tau \text{ event}
\end{align*}
\]
Dynamic Semantics

Sequentialization contexts:

\[ S ::= [] | \text{let } x \text{ be } S \text{ inc end} \]

Sequential evaluation:

\[
(C, P)[\text{let } x \text{ be } S \text{ inc end}] \rightarrow_{\text{CML}} (C, P)[\text{let } x \text{ be } S \text{ inc end}]
\]

\[ (C, P)[\text{spwan}(c)] \rightarrow_{\text{CML}} (C, P)[\text{let } x \text{ be } S \text{ inc end}]
\]

(and similarly for the other CML primitives)

Type Safety

A machine state \((C, p)\) is \textbf{well-formed} iff

- there exists \(\Gamma\) such that
  - for every \(c \in C\), \(\Gamma(c) = \tau\text{ chan}\) for some \(\tau\);
  - \(\Gamma \vdash p\) ok.

Dynamic Semantics

\[
(C, P)[\text{let } x \text{ be } S \text{ inc end}] \rightarrow_{\text{CML}} (C, P)[\text{let } x \text{ be } S \text{ inc end}]
\]

\[ (C, P)[\text{spwan}(c)] \rightarrow_{\text{CML}} (C, P)[\text{let } x \text{ be } S \text{ inc end}]
\]

A complication: what about nested uses of CML primitives?

- Argument to a function performs a \textit{spawm}.
- Nested sync’s.

Impose these restrictions (without loss of generality):

- Require arguments to primitives to be \textbf{values} (except for \textit{spawm}).
- Explicitly sequentialize using \textit{let}’s.

Dynamic Semantics

The rules for wrap and guard must be changed slightly: Invariant:

- Replace \(v_2(\text{sync}(v_1))\) in the rule for wrappers by
  
  \[
  \text{let } x \text{ be } v(()) \text{ sync } x \text{ end}
  \]

- Replace \(\text{sync}(v())\) in the rule for guard by
  
  \[
  \text{let } x \text{ be } v(()) \text{ sync } x \text{ end}
  \]
**Type Safety**

Theorem 1 (Safety)

If \((C, p)\) is well-formed, then either it is a final state, or it is deadlocked, or there exists a well-formed state \((C', p')\) such that \((C, p) \rightarrow (C', p')\).

Note that typing does not preclude deadlock! (But there are type systems that do.)

---

**Concurrent MinML Summary**

We can define an abstract machine for CML that models

1. Thread creation.
2. Channel allocation.
3. Synchronization.

This model supports a type safety theorem that does not preclude deadlock.

---

**Dataflow Networks**

Recall: a stream is a suspended computation that produces values on demand.

Idea: model streams as concurrently executing threads.

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

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 perfect waveform).
- Model a gate as a stream transducer.

Example: RS latch built from two nor gates.

---

**Example: Flip-Flops**

Constantly high or low wires:

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

Nor-gate, with unit delay:

```ml
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:

```ml
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
```

Copy, with unit delay:

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

Build an RS latch, using backpatching to implement loopback:

```ml
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
```

Test harness:

```ml
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.

The Fibonacci network:

```ml
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:

```haskell
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:

```haskell
fun delay_init (inp, out) = forever () (fn x => (send (out, x); recv inp))
```

Example: Fibonacci Network

Synchronous split of one channel into two:

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

Example: Fibonacci Network

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:

```haskell
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!
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

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.

```plaintext
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)
```

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`.

**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`

**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, lvar's, etc..

- Startup: RunCML, etc..