Supplementary Notes on Concurrent ML

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 25
November 21, 2002

In the last lecture we discussed the π-calculus, a minimal language with synchronous communication between concurrent processes. Mobility is modeled by allowing channels to transmit other channels, enabling dynamic reconfiguration of communication patterns.

Concurrent ML (CML) is an extension of Standard ML with concurrency primitives that heavily borrown from the π-calculus. In particular, channels can carry values (including other channels), communication is synchronous, and execution is concurrent. However, there are also differences. Standard ML is a full-scale programming language, so some idioms that have to be coded painfully in the π-calculus are directly available. Moreover, CML offers another mechanism called *negative acknowledgments*. In this lecture we will not discuss negative acknowledgments and concentrate on the fragment of CML that corresponds most directly to the π-calculus. The examples are drawn from the standard reference:[1]

> John H. Reppy, *Concurrent Programming in ML*, Cambridge University Press, 1999.

We begin with the representation of *names*. In CML they are represented by the type τ `chan` that carries values of type τ. We show the relevant portion of the signature for the structure `CML`.

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

[1] See also `http://people.cs.uchicago.edu/~jhr/cml/`.

The `send` and `recv` operations are *synchronous* which means that a call `send (a, v)` will block until there is a matching `recv (a)` in another thread of computation and the two rendezvous. We will see later that `send` and `recv` are actually definable in terms of some lower-level constructs.

What we called a process in the π-calculus is represented as a *thread* of computation in CML. They are called threads to emphasize their relatively lightweight nature. Also, they are executing with shared memory (the Standard ML heap), even though the model of communication is *message passing*. This imposes a discipline upon the programmer not to resort to possibly dangerous and inefficient use of mutable references in shared memory and use message passing instead.

The relevant part of the CML signature is reproduced below. In this lecture we will not use `thread_id` which is only necessary for other styles of concurrent programming.

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

Even without non-deterministic choice, that is, the sums from the π-calculus, we can now write some interesting concurrent programs. The example we use here is the sieve of Eratosthenes presented in the π-calculus in the last lecture. The pattern of programming this examples and other related programs in CML is the following: a function will accept a parameter, spawn a process, and return on or more channels for communication with the process it spawned.

The first example is a counter process that produces a sequence of integers counting upwards from some number n. The implementation takes n as an argument, creates an output channel, defines a function which will be the looping thread, and then spawns the thread before returning the channel.

```
(* val counter : int -> int CML.chan *)
fun counter (n) =
    let
      val outCh = CML.channel ()
      fun loop (n) = (CML.send (outCh, n); loop (n+1))
    in
      CML.spawn (fn () => loop n);
      outCh
    end
```

The internal state of the process is not stored in a reference, but as the argument of the `loop` function which runs in the counter thread.

Next we define a function `filter` which takes a prime number `p` as an argument, together with an input channel `inCh`, spawns a new filtering process and returns an output channel which returns the result of removing all multiples of `p` from the input channel.

```
(* val filter : int * int CML.chan -> int CML.chan *)
fun filter (p, inCh) =
    let
      val outCh = CML.channel ()
      fun loop () =
          let val i = CML.recv inCh
          in
            if i mod p <> 0
              then CML.send (outCh, i)
            else ();
            loop ()
          end
    in
      CML.spawn (fn () => loop ());
      outCh
    end
```

Finally, the `sieve` function which returns a channel along which an external thread can receive successive prime numbers. It follows the same structure as the functions above.

```
(* val sieve : unit -> int CML.chan *)
fun sieve () =
    let
      val primes = CML.channel ()
      fun head ch =
          let
            val p = CML.recv ch
          in
            CML.send (primes, p);
            head (filter (p, ch))
          end
    in
      CML.spawn (fn () => head (counter 2));
      primes
    end
```

When `sieve` is creates a new channel and then spawns a process that will produces prime numbers along this channel. It also spawns a process to enumerate positive integers, starting with 2 and counting upwards. At this point it blocks, however, until someone tries to read the first prime number from its output channel. Once that rendezvous has taken place, it spawns a new thread to filter multiples of the last prime produced with `filter (p, ch)` and uses that as its input thread.

To produce a list of the first n prime numbers, we successively communicate with the main thread spawned by the call to `sieve`.

```
(* val primes : int -> int list *)
fun primes (n) =
    let
      val ch = sieve ()
      fun loop (0, l) = List.rev l
        | loop (n, l) = loop (n-1, CML.recv(ch)::l)
    in
      loop (n, nil)
    end
```

For non-deterministic choice during synchronization, we need a new notion in CML which is called an *event*. Event are values that we can synchronize on, which will block the current thread. Event combinators will allow us to represent non-deterministic choice. The simplest forms of events are *receive* and *send* events. When synchronized, they will block until the rendezvous along a channel has happened.

```
type 'a event
val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
val never : 'a event
val alwaysEvt : 'a -> 'a event
val wrap : 'a event * ('a -> 'b) -> 'b event
val choose : 'a event list -> 'a event
val sync : 'a event -> 'a
```

Synchronization is achieved with the function `sync`. For example, the earlier `send` function can be defined as

```
val send = fn (a,x) => sync (sendEvt (a,x))
```

that is, `val send = sync o sendEvt`.

We do not use `alwaysEvt` here, but its meaning should be clear: it corresponds to a τ action returning a value without any communication.

`choose [`$v_1, \ldots, v_n$`]` for event values $v_1, \ldots, v_n$ corresponds to a sum $N_1 + \cdots + N_n$. In particular, `choose []` will block and can never proceed, while `choose [`v`]` should be equivalent to v.

`wrap (`v`, `f`)` provides a function v to be called on the result of synchronizing v. This is needed because different actions may be taken in the different branches of a `choose`. It is typical that each primitive receive or send event in a non-deterministic choice is wrapped with a function that indicates the action to be taken upon the synchronization with the event.

As an example we use the implementation of a storage cell via a concurrent process. This is an implementation of the following signature.

```
signature CELL =
sig
  type 'a cell
  val cell : 'a -> 'a cell
  val get : 'a cell -> 'a
  val put : 'a cell * 'a -> unit
end;
```

In this example, creating a channel returns two channels for communication with the spawned thread: one to read the contents of the cell, and one to write the contents of the cell. It is up to the client program to make sure the calls to `get` and `put` are organized in a way that does not create incorrect interference in case different threads want to use the cell.

```
structure Cell' :> CELL =
struct
datatype 'a cell =
  CELL of 'a CML.chan * 'a CML.chan
fun cell x =
    let
      val getCh = CML.channel ()
      val putCh = CML.channel ()
      fun loop x = CML.synch (
        CML.choose [CML.wrap (CML.sendEvt (getCh, x),
                              fn () => loop x),
                    CML.wrap (CML.recvEvt putCh,
                              fn x' => loop x')])
    in
      CML.spawn (fn () => loop x);
      CELL (getCh, putCh)
    end
fun get (CELL(getCh, _)) = CML.recv getCh
fun put (CELL(_, putCh), x) = CML.send (putCh, x)
end;
```

This concludes our treatment of the high-level features of CML. Next we will sketch a formal semantics that accounts for concurrency and synchronization. The most useful basis is the C-machine, which makes a continuation stack explicit. This allows us to easily talk about blocked processes or synchronization. The semantics is a simplified version of the one presented in Reppy's book, because we do not have to handle negative acknowledgments. Also, the notation is more consistent with our earlier development.

First, we need to introduce channels. We denote them by a, following the π-calculus. Channels are typed $a : \tau\,\mathsf{chan}$ for types τ. During the evaluation, new channels will be created and have to be carried along as a *channel environment*. This reminiscent of thunks, or memory in other evaluation models we have discussed. These channels are global, that is, shared across the whole process state. Finally we have the state s of individual thread, which are as in the C-machine.

$$\begin{array}{rrcl} \text{Channel env} & \mathcal{N} & ::= & \cdot \mid \mathcal{N}, a\,\mathsf{chan} \\ \text{Machine state} & P & ::= & \cdot \mid P, s \\ \text{Thread state} & s & ::= & K > e \mid K < v \end{array}$$

In order to write rules more compactly, we allow the silent re-ordering of threads in a machine state. This does imply any scheduling strategy.

We have two judgments for the operational semantics

$s \mapsto s'$	Thread steps from s to s'
$(\mathcal{N} \vdash P) \mapsto (\mathcal{N}' \vdash P')$	Machine steps from P to P'

In the latter case we know that $\mathcal{N}'$ is either $\mathcal{N}$ or contains one additional channel that may have been created. The first judgment, $s \mapsto s'$ is exactly as it was before in the C-machine. We have one general rule

$$\frac{s \mapsto s'}{(\mathcal{N} \vdash P, s) \mapsto (\mathcal{N} \vdash P, s')}$$

We now define the new constructs, one by one.

Channels. Channels are created with the `channel` function. They are value.

$$\frac{}{a \ \mathsf{value}} \qquad \frac{(a \ \mathsf{chan} \notin \mathcal{N})}{(\mathcal{N} \vdash P, K > \texttt{channel}\,()) \mapsto (\mathcal{N}, a \ \mathsf{chan} \vdash P, K < a)}$$

We do not define the semantics of the `send` and `recv` functions because they are definable.

Threads. New threads are created with the `spawn` function. We ignore here the `thread_id` type and return a unit element instead.

$$\frac{}{(\mathcal{N} \vdash P, K > \texttt{spawn}\, v) \mapsto (\mathcal{N} \vdash P, \bullet > v\,(), K < ())}$$

$$\frac{}{(\mathcal{N} \vdash P, K > \texttt{exit}\,()) \mapsto (\mathcal{N} \vdash P)}$$

Recall that even though we write the relevant thread among P last, it could in fact occur anywhere by our convention that the order of the threads is irrelevant.

Finally, we come to events. We make one minor change to make them syntactically easier to handle. Instead of `choose` to take an arbitrary list of events, we have two constructs:

```
val choose : 'a event * 'a event -> 'a event
val never : 'a event
```

Events must be values in this implementation, because they must become arguments to the synchronization function `sync`.

$$\frac{v \text{ value}}{\texttt{sendEvt}(a, v) \text{ value}} \qquad \frac{}{\texttt{recvEvt}(a) \text{ value}} \qquad \frac{v \text{ value}}{\texttt{always}(v) \text{ value}}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{choose}(v_1, v_2) \text{ value}} \qquad \frac{}{\texttt{never} \text{ value}}$$

$$\frac{v_1 \text{ value} \quad v_2 \text{ value}}{\texttt{wrap}(v_1, v_2) \text{ value}}$$

From these value definitions one can straightforwardly derive the rules that evaluate subexpressions. Interestingly, there only two new rules for the operational semantics: for two-way synchronization (corresponding to a value being sent) and one-way synchronization (corresponding to a τ-action with a value). This requires two new judgments, $(v, v') \rightsquigarrow (e, e')$ and $v \rightsquigarrow e$. We leave the one-way synchronization as an exercise and show the details of two-way synchronization.

$$\frac{(v, v') \rightsquigarrow (e, e')}{(\mathcal{N} \vdash P, K > \texttt{sync}(v), K > \texttt{sync}(v')) \mapsto (\mathcal{N} \vdash P, K > e, K > e')} \text{ R}_2$$

$$\frac{v \rightsquigarrow e}{(\mathcal{N} \vdash P, K > \texttt{sync}(v)) \mapsto (\mathcal{N} \vdash P, K > e)} \text{ R}_1$$

The judgment $(v, v') \rightsquigarrow (e, e')$ means that v and v' can rendezvous, returning expression e to the first thread and e' to the second thread. We show the rules for it in turn, considering each event combinator. We presuppose that subexpressions marked v are indeed values, without checking this explicitly with the v value judgment.

Send and receive events. This is the base case. The sending thread continues with the unit element, while the receiving thread continues with the value carried along the channel a.

$$\frac{}{(\texttt{sendEvt}(a, v), \texttt{recvEvt}(a)) \rightsquigarrow ((), v)} \; sr$$

$$\frac{}{(\texttt{recvEvt}(a), \texttt{sendEvt}(a, v)) \rightsquigarrow (v, ())} \; rs$$

Choice events. There are no rules to synchronize on `never` events, and there are four rules for the binary `choose` event.

$$\frac{(v_1, v') \rightsquigarrow (e, e')}{(\texttt{choose}(v_1, v_2), v') \rightsquigarrow (e, e')}\ c_1^l \qquad \frac{(v_2, v') \rightsquigarrow (e, e')}{(\texttt{choose}(v_1, v_2), v') \rightsquigarrow (e, e')}\ c_2^l$$

$$\frac{(v, v_1') \rightsquigarrow (e, e')}{(v, \texttt{choose}(v_1', v_2')) \rightsquigarrow (e, e')}\ c_1^r \qquad \frac{(v, v_2') \rightsquigarrow (e, e')}{(v, \texttt{choose}(v_1', v_2')) \rightsquigarrow (e, e')}\ c_2^r$$

Wrap events. Finally we have `wrap` events that construct bigger expressions, to be evaluated if synchronization selects the corresponding event. This is way synchronization returns an expression, to be evaluated further, rather than a value.

$$\frac{(v_1, v') \rightsquigarrow (e_1, e')}{(\texttt{wrap}(v_1, v_2), v') \rightsquigarrow (v_2\, e_1, e')}\ w^l$$

$$\frac{(v, v_1') \rightsquigarrow (e, e_1')}{(v, \texttt{wrap}(v_1', v_2')) \rightsquigarrow (e, v_2'\, e_1')}\ w^r$$

With the typing rules derived from the CML signature and the operational semantics, it is straightforward to prove a type preservation result. The only complication is presented by names, since they are created dynamically. But we have already seen the solution to a very similar problem when dealing with mutable references (since locations l are also created dynamically), so no new concepts are required.

Progress is more difficult. The straightforward statement of the progress theorem would be false, since the type system does not track whether processes can in fact deadlock. Also, we would have to re-think what nontermination means, because some processes might run forever, while others terminate, while yet others block. We will not explore this further, but it would clearly be worthwhile to verify that any thread can either progress, exit, return a final value, or block on an event. This means that there are no "unexpected" violations of progress. Along similar lines, it would be very interesting to consider type systems in which concurrency and communication is tracked to the extent that a potential deadlock would be a type error! This is currently an active area of research.