Lecture Notes on The π-Calculus and Concurrent ML

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 26
December 7, 2004

In this lecture we first generalize the calculus of concurrent processes so that values can be transmitted during communication. But our language has no primitive values, so this just reduces to transmitting names along channels that are themselves represented as names. This means that a system of processes can dynamically change its communication structure because connections to processes can be passed as first class values. This is why the resulting language, the π-calculus, is called a calculus of *mobile* and *concurrent* communicating processes. In the second part of the lecture we show how concurrency primitives along the lines of the π-calculus can be embedded in ML, leading to Concurrent ML (CML).

We generalize actions and differentiate them more explicitly into input actions and output actions, since one side of a synchronized communication act has to send and the other to receive a name. We also replace primitive process identifiers and defining equations by process replication $!P$ explained below.

$$\begin{array}{rrcll}
\text{Action prefixes} & \pi & ::= & a(y) & \text{receive } y \text{ along } a \\
 & & \mid & \overline{a}\langle b\rangle & \text{send } b \text{ along } a \\
 & & \mid & \tau & \text{unobservable action} \\
\\
\text{Process exps} & P & ::= & N \mid (P_1 \mid P_2) \mid \mathsf{new}\, x.P \mid !P & \\
\text{Sums} & N & ::= & 0 \mid N_1 + N_2 \mid \pi.P &
\end{array}$$

In examples $\pi.0$ is often abbreviated by π. Note that in a summand $a(y).P$, y is a *bound variable* with scope P that stands for the value received along a. On the other hand, $\overline{a}\langle b\rangle.P$ does not bind any variables. Even though the syntax does not formally distinguish, we use x for binding occurrences of names (subject to renaming), and a and b for non-binding occurrences.

The structural congruence remains the same as before, except that in addition we have $!P \equiv P \mid !P$, that is, a process $!P$ can spawn arbitrarily many copies of itself. For references, we repeat the laws here.

1. Renaming of bound variables (α-conversion)
2. Reordering of terms in a summation
3. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) = (P \mid Q) \mid R$
4. $\mathsf{new}\, x.(P \mid Q) \equiv P \mid \mathsf{new}\, x.Q$ if $x \notin \mathrm{fn}(P)$
   $\mathsf{new}\, x.0 \equiv 0$, $\mathsf{new}\, x.\mathsf{new}\, y.P \equiv \mathsf{new}\, y.\mathsf{new}\, x.P$
5. $!P \equiv P \mid !P$

Before presenting the transition semantics, we consider the following example.

$$P = ((\overline{x}\langle y\rangle.0 + z(w).\overline{w}\langle y\rangle.0) \mid x(u).\overline{u}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0)$$

The middle process can synchronize and communicate with either the first or the last one. Reaction with the first leads to

$$P_1 = (0 \mid \overline{y}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0) \equiv (\overline{y}\langle v\rangle.0 \mid \overline{x}\langle z\rangle.0)$$

which cannot transition further. Reaction with the seconds leads to

$$P_1' = ((\overline{x}\langle y\rangle.0 + z(w).\overline{w}\langle y\rangle.0) \mid \overline{z}\langle v\rangle.0 \mid 0)$$

which can step further to

$$P_2' = (\overline{v}\langle y\rangle.0 \mid 0 \mid 0)$$

Next we show the reaction rules in a form which does not make an externally observable action explicit, and exploits structural congruence.

$$\frac{}{\tau.P + N \longrightarrow P}\ \text{Tau}$$

$$\frac{}{(a(x).P + M) \mid (\overline{a}\langle b\rangle.Q + N) \longrightarrow (\{b/x\}P) \mid Q}\ \text{React}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q}\ \text{Par} \qquad \frac{P \longrightarrow P'}{\mathsf{new}\, x.P \longrightarrow \mathsf{new}\, x.P'}\ \text{Res}$$

$$\frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}\ \text{Struct}$$

As a simple example we will model a storage cell that can hold a value and service get and put requests to read and write the cell contents. We first show it using definitions for process identifiers and then rewrite it using process replication.

$$\begin{array}{rcl} C(x, \mathrm{get}, \mathrm{put}) & \stackrel{\mathrm{def}}{=} & \overline{get}\langle x\rangle.C\langle x, \mathrm{get}, \mathrm{put}\rangle \\ & & + \, put(y).C\langle y, \mathrm{get}, \mathrm{put}\rangle \end{array}$$

We express this in the π-calculus by turning C itself into a name, left-hand side into an input action and occurrences on the right-hand side into an output action.

$$!\, c(x, \mathrm{get}, \mathrm{put}).(\overline{get}\langle x\rangle.\overline{c}\langle x, \mathrm{get}, \mathrm{put}\rangle.0 + put(y).\overline{c}\langle y, \mathrm{get}, \mathrm{put}\rangle.0)$$

We abbreviate this process expression by $!C$. In order to be in the calculus we must be able to receive and send multiple names at once. It is straightforward to add this capability. As an example, consider how to create cell with initial contents 3, write 4 to it, read the cell and then print the contents some output device. Printing a is represented by an output action $\overline{print}\langle a\rangle.0$. We also consider 3 and 4 just as names here.

$$!C \mid \mathsf{new}\, g.\mathsf{new}\, p.\overline{c}\langle 3, g, p\rangle.\overline{p}\langle 4\rangle.g(x).\overline{print}\langle x\rangle.0$$

Note that c and print are the only free names in this expression. Note also that we are creating new names g and p to stand for the channel to get or put a names into the storage cell C. We leave it to the reader as an instructive exercise to simulate the behavior of this expression. It should be clear, however, that we need to use structural equivalence initially to obtain a copy of C with which we can react after moving the quantifiers of g and p outside.

As a more involved example, consider the following specification of the sieve of Eratosthenes. We start with a stream to produce integers, assuming we have a primitive successor operation on integer names.[1] The idea is to have a channel which sends successive numbers.

$$!count(n, \mathrm{out}).\overline{out}\langle n\rangle.\overline{count}\langle n+1, \mathrm{out}\rangle$$

Second we show a process to filter all multiples of a given prime number from its input stream while producing the output stream. We assume an oracle $(x\,mod\,p = 0)$ and its negation.

$$\begin{aligned} !filter(p, \mathrm{in}, \mathrm{out}).in(x).((x\,mod\,p = 0)().\overline{filter}\langle p, \mathrm{in}, \mathrm{out}\rangle.0 \\ + (x\,mod\,p \neq 0)().\overline{out}(x).\overline{filter}\langle p, \mathrm{in}, \mathrm{out}\rangle.0) \end{aligned}$$

Finally, we come to the process that generates a sequence of prime numbers, starting from the first item of the input channel which should be prime (by invariant).

$$\begin{aligned} &!primes(\mathrm{in}, \mathrm{out}).in(p).\overline{out}\langle p\rangle. \\ &\quad \mathsf{new}\,\mathrm{mid}.(\overline{filter}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \mid \overline{primes}\langle \mathrm{mid}, \mathrm{out}\rangle.0) \end{aligned}$$

primes establishes a new filtering process for each prime and threads the input stream in into the filter. The first element of the filtered result stream is guaranteed to be prime, so we can invoke the primes process recursively.

At the top level, we start the process with the stream of numbers counting up from 2, the smallest prime. This will generate communication requests $\overline{out}\langle p\rangle$ for each successive prime.

$$\mathsf{new}\,\mathrm{nats}.\overline{count}\langle 2, \mathrm{nats}\rangle \mid \overline{primes}\langle \mathrm{nats}, \mathrm{out}\rangle$$

In this implementation, communication is fully synchronous, that is, both sender and receiver can only move on once the message has been exchanged. Here, this means that the prime numbers are guaranteed to be read in their natural order. If we don't care about the order, we can rewrite the process so that it generates the primes *asynchronously*. For this we use the general transformation of

$$\overline{a}\langle b\rangle.P \quad \Longrightarrow \quad \tau.(\overline{a}\langle b\rangle.0 \mid P)$$

[1]This can also be coded in the π-calculus, but we prefer to avoid this complication here.

which means the computation of P can proceed regardless whether the message b has been received along channel a. In our case, this would be a simple change in the primes generator.

$$\begin{array}{l} !\mathit{primes}(\mathrm{in}, \mathrm{out}).\mathit{in}(p). \\ \quad \overline{\mathit{out}}\langle p\rangle.0 \mid \mathsf{new}\,\mathrm{mid}.(\overline{\mathit{filter}}\langle p, \mathrm{in}, \mathrm{mid}\rangle.0 \mid \overline{\mathit{primes}}\langle \mathrm{mid}, \mathrm{out}\rangle.0) \end{array}$$

The advantage of an asynchronous calculus is its proximity to a realistic model of computation. On the other hand, synchronous communciation allows for significantly shorter code, because no protocol is needed to make sure messages have been received, and in received in order. Since asynchronous communication is very easily coded here, we stick to Milner's original π-calculus which was synchronous.

In the remainder of this lecture we discuss how Concurrent ML (CML) implements concurrency primitives that heavily borrow from the π-calculus. In CML, 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:[2]

> 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
```

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.

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

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 one 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 loop ch =
          let
            val p = CML.recv ch
            val _ = CML.send (primes, p)
            val mid = filter (p, ch)
          in
            loop (mid)
          end
    in
      CML.spawn (fn () => loop (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*. Events 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.

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

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`.

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

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

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

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.

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

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

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 $v \;\; \mathsf{value}$ 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 non-termination 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.