CS 524

Lecture 18

José Meseguer

The π-Calculus

By design, Robin Milner's π-calculus tried to be for concurrency what the λ-Calculus is for functional computation: a language in which one could encode other concurrency models. He did indeed show how that: (i) λ-calculus itself could be encoded; (ii) his previous Calculus of Communicating Systems (CCS), a concurrency model of synchronously ~~communicating automata~~ interacting automata was naturally generalized by the π-Calculus, and (iii) argued that the Actor model could in some way be simulated by the π-Calculus, which some encodings of this nature developed by some of his collaborators.

As argued in earlier lectures, concurrency is such a "polymorphic" [from "poly" = many, and "morphe" = "form"] phenomenon, that claims of universality are harder to sustain than for deterministic systems and, what is even less satisfactory, even if an ecoding can be found for a given model, much of the intrinsic nature and features of the original model can easily be lost in translation. This problem is well-known even in the considerably more favorable case of deterministic computation: any effectively computable deterministic system can indeed be encoded, i.e., compiled,

into a Turing machine; and, therefore, assuming the idealized illusion of unlimited secondary storage, executed in a microprocessor, which is just the Turing machine's physical realization. But nobody has any illusions that the original features of the system have been preserved by the encoding/compilation process: many are lost in translation. For starters, any concurrency is lost, since a Turing machine is an intrinsically sequential system: it can only do one thing at a time.

In spite of the above limitations of any claims to universality, either for sequential or for concurrent computation, the very brave fact of having tried to seek a model of concurrency general enough so as to be able to simulate other such models is both commendable and, in the expert hands of the late Robin Milner, did result in an elegant and beautiful model which has been one of his main contributions to concurrency, and has inspired work by many other researchers.

1. Main Ideas and Basic Syntax

The main idea in the π-calculus is that of dynamically changing patterns of synchronous

communication between concurrent processes. Milner calls the dynamically changing nature of the synchronous communication links between processes, — where such links are called channels — mobility.

Milner's notion of mobility is quite abstract. In his own words, the π-calculus notion of mobility means that, as we shall shortly see by example,

> "links move, in the virtual space of linked processes;"
>
> (Milner, the π-Calculus, Cambridge UP, pg. 78)

However, he illustrates by an example of physically mobile vehicles communicating wirelessly with different communication towers coordinated by a central control station that physical mobility can also be naturally modeled in the π-calculus [See Section 3.2 of Milner's book].

Let us focus on the main features of the π-calculus supporting synchronous process communication. I will later on explain how two more features, namely, non-deterministic choice and replication, can easily be added. Here is the basic syntax.

The π-calculus, like the λ-calculus in which it is inspired ~~to~~ is parametric on a set of <u>names</u> $\mathcal{N}$, which we assume countably infinite. To distinguish between <u>concrete names</u> like $x, y, z, \ldots \in \mathcal{N}$, and <u>meta-variables for names</u>, I use <u>capital letters</u> for such metavariables X, Y, Z.

Since Milner is interested in uncovering the <u>essential features</u> of concurrent communication and not so much in specific applications, in his calculus names are used, <u>simultaneously</u> for <u>two</u> <u>completely different purposes</u>:

1. to <u>name</u> <u>communication channels</u>, and
2. as <u>data elements</u> <u>sent</u> along channels.

Of course, this basic π-calculus can and has been extended to the more practical case where elements of data types are the data sent through channels.

<u>Processes</u> are the concurrent entities that communicate asynchronously with each other through specific channels. I will use P, Q, R, and so on as specific <u>meta-variables</u> ranging over

such processes. Here is now the basic syntax of our π-calculus fragment:

$$\text{Process} ::= \mathbf{0} \mid P \mid Q \mid \underline{\text{new}}[X]\{P\} \mid \underline{\text{out}}\ X\langle Y\rangle.P \mid \underline{\text{in}}\ X[Y].P$$

Milner himself favors an even more Spartan syntax, namely,

$$\text{Process} ::= \mathbf{0} \mid P \mid Q \mid \underline{\text{new}}\ X\ P \mid \overline{X}\langle Y\rangle.P \mid X(Y).P$$

where, as a legacy of CCS, he assumes a name complementation operation $\overline{x}$, so that $\overline{\overline{x}} = x$. He also allows an <u>unobservable action</u> τ, so that a process can have the form $\tau.P$. These are technical legacies from CCS that I will safely ignore; but I will indicate in the crucial semantic rule for synchronous communication the slight notational difference between Milner's notation and my own.

The <u>intended</u> <u>meaning</u> of the above syntax is:

1. $\mathbf{0}$ is the <u>null</u> process: it does not <u>do</u> anything
2. $P \mid Q$ is <u>parallel</u> <u>composition</u> of P and Q. It is of course <u>associative</u> and <u>commutative</u> and has $\mathbf{0}$ as its <u>identity</u> element.

3. new $[x]\{P\}$ is a name abstraction construct, like $\lambda x.U$, where the name x, which may appear in several of the processes making up P, has been bound for the purpose of restricting its use for communication ~~pocu~~ purposes outside the "bubble" $\{P\}$, which I have made explicit in my choice of syntax for this very purpose. That is, any process of the form in $x[y].P$, or out $x\langle z\rangle.Q$ outside the bubble $\{P\}$ cannot communicate with a process inside $\{P\}$ through channel x.

4. out $x\langle y\rangle.P$ is a process ready to send value y through channel x. After this sending has been performed it will become its continuation P. Note that out is not a (name) binding construct at all: no names are bound here.

5. in $x[y].P$ is a process awaiting to receive a value through channel x. This value is currently unknown, so the name y is used as a pattern, which will be instantiated in P itself [where y will typically appear] so that, after receiving value, say, z, the continuation will be $P\{y \mapsto z\}$.

2. Process Equivalence

Q: When are two π-calculus processes the same

A: If the can be proved equal by means of the following equations:

1. Alpha-conversion of bound names for the new and in binders [similar to the λ-calculus].
2. Associativity, commutativity of $\_|\_$ with identity 0.
3. $P \mid \underline{new}[X]\{Q\} = new[X]\{P \mid Q\}$ if $X \notin fn(P)$

   where $fn(P)$ denotes the set of free [i.e. non-bound] names of P in the same sense as $fv(U)$ for a λ-calculus term.
4. $new[X]\{0\} = 0$, $new[X]\{new[Y]\{P\}\} =$

   $= new[Y]\{new[X]\{P\}\}$.

3. Synchronous Communication

Like the λ-calculus, the π-calculus has a single semantic rule, analogous to the β-rule, namely,

$$\boxed{\begin{array}{l}\underline{\text{Synchronous Communication}}: \\ \quad \underline{out}\ X\langle Y\rangle.P \mid \underline{in}\ X[Z].Q \longrightarrow P \mid (Q\{Z \mapsto Y\})\end{array}}$$

In Milner's original notation this would be written using complementary names and his Spartan notation as:

$$\overline{X}\langle Y\rangle.P \mid X(Z).Q \rightarrow P \mid (Q\{Z \mapsto Y\})$$

4. The Full Language

Can be obtained by adding:

1. non-deterministic choice: $P+Q$, where $\_+\_$ is associative and commutative and has 0 as its identity element
2. replication: $!P$ can do P forever, thanks to the equality $!P = P \mid !P$, which allows definition of recursive processes.

In the full language, the synchronous communication rule takes the somewhat more general form:

$$\boxed{P_1 + \overline{X}\langle Y\rangle.P \mid X(Z).Q + Q_1 \rightarrow P \mid (Q\{Z \mapsto Y\})}$$

5. Graphical Notation

A picture is worth a thousand words. To illustrate Milner's intuition that "links move, in the virtual space of linked processes" can best be done by adopting a simple

<u>graphical notation</u>. I will adopt the following one of my own choosing, similar to Milner's but, hopefully, slightly more suggestive.

1. <u>out</u> $y\langle x\rangle.P$ will be depicted as:

$+y \longleftarrow \langle x\rangle.P$, to indicate the <u>male</u> nature of sending through channel y, which is <u>pictorially identical to</u>:

$Q.\langle x\rangle \longrightarrow +y$

To <u>break the slavery of syntax</u>.

2. <u>in</u> $y[z].Q$ will be depicted as:

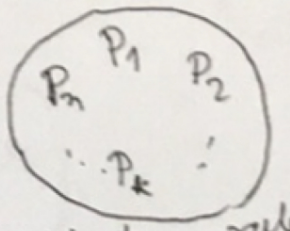
$-y \longleftarrow [z].Q$ to indicate the <u>female</u> nature of receiving through channel y, which is pictorially identical to:

$Q.[z] \longrightarrow -y$

3. $P \mid Q$ will be written with <u>empty syntax</u> $P\ Q$

4. <u>new</u>$[x]\{P_1 \mid \cdots \mid P_n\}$ will be depicted in bubble form as:

new $[x]$



5. The synchronous communication rule now takes the form:

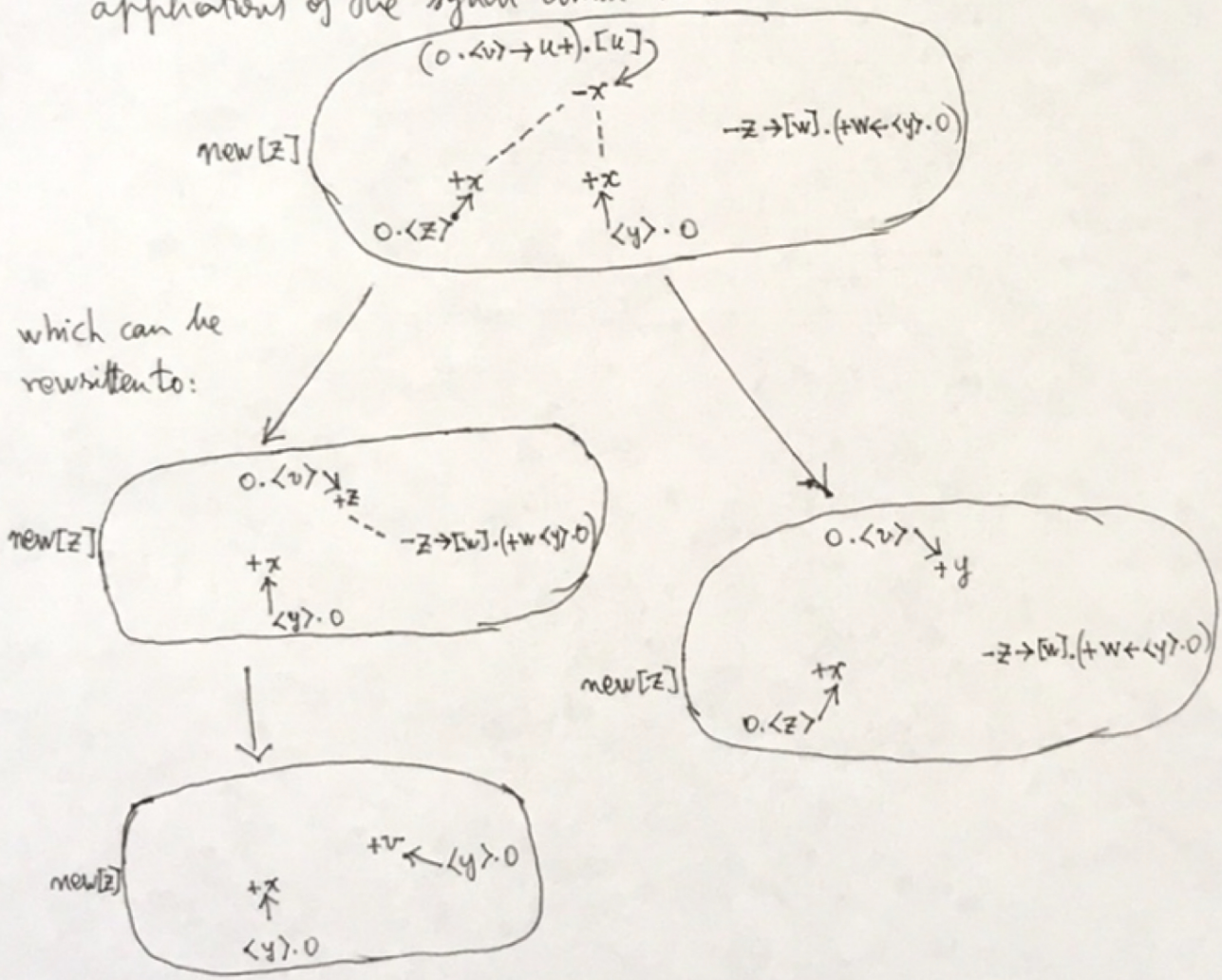
$$\boxed{\left(P.\langle Y\rangle \rightarrow +x \text{ ---- } -x \rightarrow [Z].Q\right) \rightarrow P\ \left(Q\{Z \mapsto Y\}\right)}$$

Let us see a simple example illustrating link mobility:

The process:

$$\underline{new}\,[z]\{\,\underline{in}\ x\,[u].\,\underline{out}\ u\,\langle v\rangle.0\ \mid$$
$$\underline{in}\ z\,[w].\,\underline{out}\ w\,\langle y\rangle.0\ \mid$$
$$out\ x\,\langle y\rangle.0\ \mid$$
$$out\ x\,\langle z\rangle.0\ \}$$

Can be depicted below, with dashed lines indicating possible applications of the synch-comm rule:



where the last two states cannot be further rewritten. This example, though very simple illustrates well Milner's idea of mobility. It also suggests a <u>chemical metaphor</u> [through <u>valences</u>] of <u>synchronous communication</u> as <u>chemical reaction</u>.