CHAPTER 2

π-CALCULUS

2.1 Introduction to π-calculus

We saw that λ-calculus is a theory of *sequential computation.* Here, we are interested in the results of functions applied to data. In π-calculus, we are interested in concurrent and parallel computation, communication between computing agents and continuous exchanges of input and output. There are many theories for *concurrent computation* including π-calculus, and are described as *process calculus* or algebra, where *process* means an identifiable computing agent that can interact with the environment. So, π-calculus is a process calculus. Moreover, unlike other process calculi, it has *mobility*- we can send a communication link (channel) as data that can be sent across another link.

A *process* is a computing agent that can interact with other processes by sending and receiving messages. Messages can be sent on *channels* (or *names*). There can be several senders and receivers on a single channel, but each message is sent by one process and received by one process. Communication is *synchronous*- both sender and receiver block until the message is exchanged. There is no concept of *location.* If we define a system by two processes in parallel, we don't care about whether they are on the same CPU or at different places in a distributed system. Nonetheless, these concepts can be used to extend π-calculus.

Before defining the syntax, we will first consider π-calculus using some examples. These will involve numbers and arithmetic operations, which are not natively present in π-calculus, but still can be expressed by some π-calculus terms. This holds since π-calculus is a Turing-complete model of computation.

Consider the following term in π-calculus:

$$a(x).a(y).\overline{a}\langle x+y\rangle.0$$

In this term:

- the expression $a(x)$ means that we receive a message on some channel a, and refer to it using x- x is like a function parameter, and is a bound variable.

- the dot means sequencing, and the sequences are left-to-right, i.e. first receive x on a, then receive y on b.

- $\overline{a}\langle x+y\rangle$ means that we are sending a message on channel a, and this is the result of the computation $x+y$.

- 0 is the process that does nothing, and represents termination.

We can think of this term as some server- it receives 2 numbers from some client and sends back the sum of these two numbers.

We can define a process that *communicates* on a in a dual way, i.e. a client for a server. So, consider the following term:

$$\overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z)$$

In this case, we send the numbers 2 and 3 on the channel a and await its output. Then, we process the message in some way using the call $P(z)$.

We can now put the two process in parallel so that they can communicate with each on the channel a. This is done by *reduction*:

$$\begin{gathered}
a(x).a(y).\overline{a}\langle x+y\rangle.0 \mid \overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z) \\
\downarrow \\
a(y).\overline{a}\langle 2+y\rangle.0 \mid \overline{a}\langle 3\rangle.a(z).P(z) \\
\downarrow \\
\overline{a}\langle 2+3\rangle.0 \mid a(z).P(z) \\
\downarrow \\
\overline{a}\langle 5\rangle.0 \mid a(z).P(z) \\
\downarrow \\
0 \mid P(5)
\end{gathered}$$

We will now look at some more operations in π-calculus. The choice operation $+$ gives us a choice between two different ways of communication. For instance, consider the following term:

$$a(x).a(y).\overline{a}\langle x+y\rangle.0 + b(x).b\langle x^2\rangle.0$$

We can think of this as the server providing multiple functionalities, and we can choose the one we want based on the channel name (a or b). The choice is non-deterministic and part of reduction. This means that the expression

$$a(x).a(y).\overline{a}\langle x+y\rangle.0 + b(x).b\langle x^2\rangle.0 \mid \overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z)$$

reduces in one step to

$$a(y).\overline{a}\langle 2+y\rangle.0 \mid \overline{a}\langle 3\rangle.a(z).P(z)$$

We illustrate the choice operation for the following process:

$$\begin{aligned}
a(x).a(y).\overline{a}\langle x+y\rangle.0 + b(x).\overline{b}\langle x^2\rangle.0 \mid \overline{b}\langle 3\rangle.b(z).P(z) \\
\downarrow \quad \\
\overline{b}\langle 3^2\rangle.0 \mid b(z).P(z) \\
\downarrow \quad \\
\overline{b}\langle 9\rangle.0 \mid b(z).P(z) \\
\downarrow \quad \\
0 \mid P(9)
\end{aligned}$$

We can add recursive definitions to the syntax. For instance, consider the following term:

$$A = b(x).\overline{b}\langle x^2\rangle.A$$

Then, the following illustrates how we can reduce recursively:

$$
\begin{array}{c}
b(x).\overline{b}\langle x^2\rangle.A \mid \overline{b}\langle 2\rangle.b(z).\overline{b}\langle 3\rangle.b(w).P(z,w) \\
\downarrow \\
\overline{b}\langle 2^2\rangle.A \mid b(z).\overline{b}\langle 3\rangle.b(w).P(z,w) \\
\downarrow \\
A \mid \overline{b}\langle 3\rangle.b(w).P(4,w) \\
= \\
b(x).\overline{b}\langle x^2\rangle.A \mid \overline{b}\langle 3\rangle.b(w).P(4,w) \\
\downarrow \\
\overline{b}\langle 3^2\rangle.A \mid b(w).P(4,w) \\
\downarrow \\
A \mid P(4,9)
\end{array}
$$

Instead of adding recursion, we can introduce *replication* to get a simpler theory. For a process P, the term $!P$ represents a potentially unlimited number of copies of P in parallel. We can pull another copy out whenever we need to. For instance, the process

$$!(b(x).\overline{b}\langle x^2\rangle.0) \mid \overline{b}\langle 2\rangle.b(z).\overline{b}\langle 3\rangle.b(w).P(z,w)$$

is equal to

$$b(x).\overline{b}\langle x^2\rangle.0 \mid !(b(x).\overline{b}\langle x^2\rangle.0) \mid \overline{b}\langle 2\rangle.b(z).\overline{b}\langle 3\rangle.b(w).P(z,w)$$

which reduces (eventually) to

$$0 \mid !(b(x).\overline{b}\langle x^2\rangle.0) \mid \overline{b}\langle 3\rangle.b(w).P(4,w)$$

At this point, we can pull out another copy, to get the equivalent process

$$0 \mid b(x).\overline{b}\langle x^2\rangle.0 \mid !(b(x).\overline{b}\langle x^2\rangle.0) \mid \overline{b}\langle 3\rangle.b(w).P(4,w)$$

and continue reduction.

The π-calculus is based on non-determinism, which can lead to some issues. For instance, there can be several senders and receivers on the same channel in parallel. Consider the following term:

$$a(x).a(y).\overline{a}\langle x+y\rangle.0 \mid \overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z) \mid \overline{a}\langle 4\rangle.\overline{a}\langle 5\rangle.a(w).Q(w)$$

Then, this process can reduce to either

$$a(y).\overline{a}\langle 2+y\rangle.0 \mid \overline{a}\langle 3\rangle.a(z).P(z) \mid \overline{a}\langle 4\rangle.\overline{a}\langle 5\rangle.a(w).Q(w)$$

or

$$a(y).\overline{a}\langle 3+y\rangle.0 \mid \overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z) \mid \overline{a}\langle 3\rangle.a(w).Q(w)$$

Now, for the process to not get stuck, we need to ensure that the channel a receives the second message from the same channel.

To avoid the issue above of getting stuck, we can make use of the restriction operator ν. The restriction operator defines a local scope for a channel. It is a binder, and we can use α-equivalence to rename a local channel, e.g. the channel

$$(\nu\ a)\big(a(x).a(y).\overline{a}\langle x+y\rangle.0 \mid \overline{a}\langle 2\rangle.\overline{a}\langle 3\rangle.a(z).P(z)\big)$$

is α-equivalent to

$$(\nu\ b)\big(b(x).b(y).\overline{b}\langle x+y\rangle.0 \mid \overline{b}\langle 2\rangle.\overline{b}\langle 3\rangle.b(z).P(z)\big)$$

Note that the channel also leads to a bound variable, i.e. x is bound in $(\nu\ x)\big(\ldots\big)$. Bound variables can be renamed using α-equivalence.

Using restriction, we can share private channels to ensure complete interaction. This is done using scope extrusion, which is shown in the reduction below:

$$\begin{gathered}
r(a).a(x).a(y).\overline{a}\langle x+y\rangle.0 \mid (\nu\ b)\big(\overline{r}\langle b\rangle.\overline{b}\langle 2\rangle.\overline{b}\langle 3\rangle.b(z).P(z)\big) \\
= \\
(\nu\ b)\big(r(a).a(x).a(y).\overline{a}\langle x+y\rangle.0 \mid \overline{r}\langle b\rangle.\overline{b}\langle 2\rangle.\overline{b}\langle 3\rangle.b(z).P(z)\big) \\
\downarrow \\
(\nu\ b)\big(b(x).b(y).\overline{b}\langle x+y\rangle.0 \mid \overline{b}\langle 2\rangle.\overline{b}\langle 3\rangle.b(z).P(z)\big)
\end{gathered}$$

At the first step, we expand the scope to include both processes, and is called *scope expansion*. Then, we send in the channel that will be used in communication. The two steps are referred to as *scope extrusion*. The output $\overline{r}\langle b\rangle$ carries the scope of b with it, which allows us to create a private channel for the rest of the communication.

We will now illustrate how we can combine replication and restriction:

$$\begin{gathered}
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid (\nu\ b)\big(\overline{r}\langle b\rangle.\overline{b}\langle 2\rangle.b(z).P(z)\big) \\
= \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid r(a).a(x).\overline{a}\langle x^2\rangle.0 \mid \mid (\nu\ b)\big(\overline{r}\langle b\rangle.\overline{b}\langle 2\rangle.b(z).P(z)\big) \\
= \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid (\nu\ b)\big(r(a).a(x).\overline{a}\langle x^2\rangle.0 \mid \mid \overline{r}\langle b\rangle.\overline{b}\langle 2\rangle.b(z).P(z)\big) \\
\downarrow \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid (\nu\ b)\big(b(x).\overline{b}\langle x^2\rangle.0 \mid \overline{b}\langle 2\rangle.b(z).P(z)\big) \\
\downarrow \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid (\nu\ b)\big(\overline{b}\langle 2^2\rangle.0 \mid b(z).P(z)\big) \\
\downarrow \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid (\nu\ b)\big(0 \mid P(4)\big) \\
= \\
!(r(a).a(x).\overline{a}\langle x^2\rangle.0) \mid 0 \mid P(4)
\end{gathered}$$

The ability to send a channel as a message is called *mobility*. This was the key advance of π-calculus in comparison with previous process calculi such as CCS and CSP. π-calculus is called a theory of mobile processes, although actually it is the channels that are mobile. Moving a process around a network can be modelled- instead of process moving, a channel that gives access to it can move. There are extensions of π-calculus in which *processes* can be sent as messages. This is called *higher-order* communication.

We will now define π-calculus formally. Let $x, y, \ldots$ denote channel *names* or *variables*, and $P, Q, \ldots$ denote *processes*. Then, the syntax of processes is defined by the BNF below:

$$
\begin{array}{ll}
P, Q := 0 & \text{terminated process} \\
\quad\ \mid x(y).P & \text{input/receive} \\
\quad\ \mid \overline{x}(y).P & \text{output/send} \\
\quad\ \mid \tau.P & \text{silent action} \\
\quad\ \mid P + Q & \text{choice} \\
\quad\ \mid (\nu\ x)P & \text{scope/restriction} \\
\quad\ \mid !P & \text{replication} \\
\quad\ \mid P \mid Q & \text{parallel composition}
\end{array}
$$

Other process constructions, like conditions, case, etc. can be added to the syntax of processes, but they are not in the core.

Now, we want to define the *semantics* by *reduction relation* on processes. The main rule of communication is:

$$a(x).P \mid \overline{a}\langle y\rangle.Q \to P[x := y] \mid Q$$

We want to be able to apply this rule in the presence of other parallel processes, i.e. in bigger *contexts*, e.g.

$$a(x).P \mid \mathbf{R} \mid \overline{a}\langle y\rangle.Q \to P[x := y] \mid \mathbf{R} \mid Q$$

We have to dom something about the fact that communicating parts of the process might not be written next to each other. Syntax is all in a line, but we want to think of parallel processes in a space where any process can interact with any other.

To do so, we define *structural congruence* ($\equiv$) on processes; it compensates for inessential syntactic details, as well as defining some important aspects of the behaviour of processes. It is defined by several axioms, and is also a *congruence*, meaning that it is preserved by all the syntactic constructs, i.e. we can apply reduction in bigger contexts, and it is an equivalence relation. In particular, congruence means that:

- if $P \equiv Q$, then $P \mid R \equiv Q \mid R$;
- if $P \equiv Q$, then $P + R \equiv Q + R$;
- if $P \equiv Q$, then $x(y).P \equiv x(y).Q$;
- if $P \equiv Q$, then $\overline{x}\langle y\rangle.P \equiv \overline{x}\langle y\rangle.Q$;
- if $P \equiv Q$, then $(\nu\ x)P \equiv (\nu\ x)Q$; and
- if $P \equiv Q$, then $!P \equiv\ !Q$.

And, equivalence relation means that:

- $P \equiv P$;
- if $P \equiv Q$ then $Q \equiv P$; and
- if $P \equiv Q$ and $Q \equiv R$, then $P \equiv R$.

The full definition of structural congruence is given below:

$$\begin{aligned}
P \mid Q &\equiv Q \mid P && \text{parallel is commutative}\\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R && \text{parallel is associative}\\
P \mid 0 &\equiv P && \text{garbage collection}\\
P + Q &\equiv Q + P && \text{choice is commutative}\\
P + (Q \mid R) &\equiv (P + Q) + R && \text{choice is associative}\\
P + 0 &\equiv P && \text{garbage collection}\\
(\nu\ x)(\nu\ y)P &\equiv (\nu\ y)(\nu\ x)P && \text{reordering } \nu\\
(\nu\ x)0 &\equiv 0 && \text{garbage collection}\\
!P &\equiv P \mid !P && \text{replication}\\
P \mid (\nu\ x)Q &\equiv (\nu\ x)(P \mid Q) \text{ if } x \notin FV(P) && \text{scope expansion}
\end{aligned}$$

It also includes α-equivalence. Informally, the definition states that:

- we can ignore the order of processes in parallel and choice constructs;
- we do not need to write brackets in parallel and choice constructs;
- we can reorder ν binders;
- we can remove 0 and $(\nu\ x)0$ from parallel and choice constructs;
- we can pull out a copy of P from $!P$ if necessary; and
- we can expand the scope of $(\nu\ x)$ whenever necessary, and rename x if we need to, so as to avoid a variable capture.

We can now define the reduction relation. Before doing so, there are two things to consider:

- substitution is defined in a similar way to λ-calculus, but we only substitute variables; and
- bound variables can be renamed if necessary to avoid variable capture. This can be done using Barendregt convention.

Now, these are the reduction axioms:

$$\begin{aligned}
(\overline{a}\langle x\rangle.P + \dots) \mid (a\langle y\rangle.Q + \dots) &\to P \mid Q[y := x] && \text{RCom}\\
\tau.P + \dots &\to P && \text{RTau.}
\end{aligned}$$

The first one allows us to substitute via communication, while the second one takes the τ choice (which can always be chosen). We extend these using the following inference rules:

$$\frac{P \to Q}{(\nu\ x)P \to (\nu\ x)Q}\text{RNew} \qquad \frac{P \to Q}{P \mid R \to Q \mid R}\text{RPar}$$

$$\frac{P' \equiv P \quad P \to R \quad Q \equiv Q'}{P' \equiv Q'}\text{RStruct}$$