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:

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

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:

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

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 $(\nu\ x)\big(\dots\big)$. 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:

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

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:

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

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:

$P \mid Q \equiv Q \mid P$	parallel is commutative
$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$	parallel is associative
$P \mid 0 \equiv P$	garbage collection
$P + Q \equiv Q + P$	choice is commutative
$P + (Q \mid R) \equiv (P + Q) + R$	choice is associative
$P + 0 \equiv P$	garbage collection
$(\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$	reordering ν
$(\nu\ x)0 \equiv 0$	garbage collection
$!P \equiv P \mid !P$	replication
$P \mid (\nu\ x)Q \equiv (\nu\ x)(P \mid Q)$ if $x \notin FV(P)$	scope expansion

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:

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

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:

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

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

2.2 Modelling and Computation

There are two directions we can go with π-calculus:

- We can model systems by adding data and computation (e.g. integers and operations) when we need them. In this view, π-calculus is a concurrency layer on top of an assumed computational base.
- We can study π-calculus as a foundation for all computation, like λ-calculus.

Modelling in π-calculus

The mobile phones example is a classic π-calculus example that illustrates *dynamic* communication topology. Note that the key feature of π-calculus is mobility.

Here, we have a car, two transmitters and a controller. At any time, the car communicates with only one of the transmitters. The controller tells a transmitter to *lose* or *gain* connection to the car.

We will use parametrised recursive program definitions instead of replication and send/receive with multiple messages. We will later see how to translate recursive definitions into replication.

Trans is parametrised by the channels it shares with *Control*, which are *lose* and *gain*. On the other hand, *Car* is parametrised by *talk* and *switch*. It can either *talk* or obey an instruction to *lose* the connection.

IdTrans is the idle transmitter. It is parametrised by the channels it shares with *Control*, i.e. *lose* and *gain*. It can obey an instruction to gain a connection.

Putting this all together, we get:

$$\begin{aligned}
\mathit{Trans}(\mathit{talk}, \mathit{switch}, \mathit{gain}, \mathit{lose}) &= \mathit{talk}().\mathit{Trans}(\mathit{talk}, \mathit{switch}, \mathit{gain}, \mathit{lose}) \\
&\quad + \mathit{lose}(t, s).\overline{\mathit{switch}}\langle t, s\rangle.\mathit{IdTrans}(\mathit{gain}, \mathit{lose}) \\
\mathit{IdTrans}(\mathit{gain}, \mathit{lose}) &= \mathit{gain}(t, s).\mathit{Trans}(t, s, \mathit{gain}, \mathit{lose})
\end{aligned}$$

Control will either be $\mathit{Control}_1$ or $\mathit{Control}_2$. *Control* can tell one transmitter to lose a connection and the other to gain a connection. We ignore how it decides this. So, we have:

$$\begin{aligned}
\mathit{Control}_1 &= \overline{\mathit{lose}_1}\langle \mathit{talk}_2, \mathit{switch}_2\rangle.\overline{\mathit{gain}_2}\langle \mathit{talk}_2, \mathit{switch}_2\rangle.\mathit{Control}_2 \\
\mathit{Control}_2 &= \overline{\mathit{lose}_2}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.\overline{\mathit{gain}_1}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.\mathit{Control}_1
\end{aligned}$$

We treat lose_i, talk_i and switch_i channels as global variables for $i \in \{1, 2\}$. We will define a system in which they are declared in a scope that includes *Control*.

Now, the *Car* can either *talk* or *switch* to a new pair of channels. We would expect *switch* to have priority over *talk*, but this is not enforced. The definition of *Car* is therefore the following:

$$\mathit{Car}(\mathit{talk}, \mathit{switch}) = \overline{\mathit{talk}}().\mathit{Car}(\mathit{talk}, \mathit{switch}) + \mathit{switch}(t, s).\mathit{Car}(t, s)$$

Finally, we define the *System*, with the starting state Trans_1. This is given by:

$$\begin{aligned}
\mathit{System}_1 = (\nu\ &\mathit{talk}_1, \mathit{switch}_1, \mathit{gain}_1, \mathit{lose}_1, \mathit{talk}_2, \mathit{switch}_2, \mathit{gain}_2, \mathit{lose}_2) \\
&(\mathit{Car}(\mathit{talk}_1, \mathit{switch}_1) \mid \mathit{Trans}_1 \mid \mathit{IdTrans}_2 \mid \mathit{Control}_1)
\end{aligned}$$

where for $i \in \{1,2\}$,

$$\begin{aligned} \mathit{Trans}_i &= \mathit{Trans}(\mathit{talk}_i, \mathit{switch}_i, \mathit{gain}_i, \mathit{lose}_i) \\ \mathit{IdTrans}_i &= \mathit{IdTrans}(\mathit{gain}_i, \mathit{lose}_i) \end{aligned}$$

Using this definition, we can reduce System_1 reduces to System_2:
We defined *Car* as follows:

$$\mathit{Car}(\mathit{talk}, \mathit{switch}) = \overline{\mathit{talk}}().\mathit{Car}(\mathit{talk}, \mathit{switch}) + \mathit{switch}(t,s).\mathit{Car}(t,s)$$

This is recursive parametrised definition of *Car*. We can write it using replication using a replicated process that receives *talk* and *switch* on a channel called *start*. We first define *NewCar*:

$$\mathit{NewCar} = !(\mathit{start}(t,s).(\overline{\mathit{talk}}.\overline{\mathit{start}}\langle t,s\rangle.0 + \mathit{switch}(x,y).\overline{\mathit{start}}\langle x,y\rangle.0))$$

We can now replace $\mathit{Car}(\mathit{talk}_1, \mathit{switch}_1)$ with:

$$(\nu\ \mathit{start})(\overline{\mathit{start}}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.0 \mid \mathit{NewCar})$$

That way, we can reduce the following process:

$$\overline{\mathit{start}}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.0 \mid \mathit{NewCar} \mid \mathit{talk}_1().0$$

$$\overline{\mathit{start}}\langle \mathit{talk}_1, \mathit{switch}_1\rangle.0 \mid \mathit{NewCar}$$

Computation in π-calculus

In this section, we discuss how we can represent booleans and natural numbers in π-calculus.

In π-calculus, we can represent a boolean value by a process that is given two channels and communicates on one of them. This is similar to the λ-calculus representation, but we now also need to specify a channel for interaction with the boolean; this is called its *location*. We have

$$\mathit{True}(a) = a(t,f).\bar{t}\langle\rangle.0 \qquad \mathit{False}(a) = a(t,f).\bar{f}\langle\rangle.0$$

We can abbreviate definitions by omitting the 0 at the end and empty braces. For instance, we have

$$\mathit{True}(a) = a(t,f).\bar{t}$$

We can define a choice between processes P and Q on a boolean located at channel a. This is given by:

$$\mathit{Cond}(P,Q)(a) = (\nu\ t,f)\bar{a}\langle t,f\rangle.(t.P + f.Q)$$

We illustrate how this works with an example. In particular, consider the following reduction:

$$\mathit{True}(a) \mid \mathit{Cond}(P,Q)(a)$$

Similarly,

$$\mathit{False}(a) \mid \mathit{Cond}(P,Q)(a) \to^* Q$$

Next, we can define the process $\mathit{Not}(a, b)$, where b is where the boolean currently is located, and a is where it will be after the negation process. This process is given by:

$$\mathit{Not}(a, b) = (\nu\ t, f)(\bar{b}\langle t, f\rangle.(t.\mathit{False}(a) + f.\mathit{True}(a)))$$

We illustrate this with the following reduction:

$$\mathit{Not}(a, b) \mid \mathit{True}(b)$$

We now define the process $\texttt{And}(a, b, c)$, which takes in two booleans at b and c and produces a boolean at a. This is given by:

$$\mathit{And}(a, b, c) = (\nu t, f)(\bar{b}\langle t, f\rangle.(f.\mathit{False}(a) + t.\bar{c}\langle t, f\rangle.(f.\mathit{False}(a) + t.\mathit{True}(b))))$$

We illustrate this with the following reduction:

$$\mathit{And}(a, b, c) \mid \mathit{True}(b) \mid \mathit{False}(c)$$

We will now use a similar idea to represent natural numbers. It is more direct than in λ-calculus since we do not need *pair*. We define it recursively, with

$$\begin{aligned} Z(n_0) &= n_0(z, s).\overline{z} \\ S(n_k, N(n_{k-1})) &= (\nu\ n_{k-1})(n_k(z, s).\overline{s}\langle n_{k-1}\rangle \mid N(n_{k-1})) \end{aligned}$$

This way, we represent the value *One* as follows:

$$\mathit{One}(n_1) = S(n_1, Z(n_0)) = (\nu\ n_1)(n_0(z, s)\overline{s}\langle n_1\rangle \mid n_1(z, s).\overline{z})$$

Like *Cond* for booleans, we can define a case-analysis process for natural numbers, as follows:

$$\mathit{Case}(P, Q)(a, n) = (\nu\ z, s)\overline{a}\langle z, s\rangle.(z.P + s(n).Q(n)).$$

The process *Cases* interacts with a number located at a. If it is zero, it will next perform P; if it is instead $S(n)$, then it will continue as $Q(n)$. Using this, we can define the *IsZero* process:

$$\mathit{IsZero}(a, n) = \mathit{Cases}(\mathit{True}(a), \mathit{False}(a))(a, n).$$

We will now define the *even* process in π-calculus. Formally, the *even* function can be defined as follows:

$$\begin{aligned} \mathit{even}(Z) &= \mathit{true} \\ \mathit{even}(S(n)) &= \mathit{not}(\mathit{even}(n)) \end{aligned}$$

In π-calculus, we can define *Even* using replication. This is given as follows:

$$\mathit{Even} = !(\mathit{even}(a, n).\mathit{Cases}(\mathit{True}(a), (\nu\ b)(\mathit{Not}(a, b) \mid \overline{\mathit{even}}\langle b, m\rangle)(n, m)))$$

The channel where we check is *even*. The value that we are checking is n, and the result is outputted into channel a.

The example of the even function looks complicated because it combined two ideas:

- the encoding of natural numbers as processes; and
- the use of replication to express a recursive function definition.

We can explain the second point more easily if we assume that we extend π-calculus with integers, booleans, expressions formed from standard operations, reduction of boolean and integer expressions, and a condition construct

$$\textit{if } e \textit{ then } P \textit{ else } Q.$$

It has the following reduction rules:

$$\textit{if } \texttt{true} \textit{ then } P \textit{ else } Q \to P \qquad \textit{if } \texttt{false} \textit{ then } P \textit{ else } Q \to Q$$

$$\frac{e \to e'}{\textit{if } e \textit{ then } P \textit{ else } Q \to \textit{if } e' \textit{ then } P \textit{ else } Q}$$

That way, we can write the *even* process as follows:

$$\textit{Even} = !(\textit{even}(n).(\textit{if IsZero(n) then } \texttt{true} \textit{ else not(Even(n-1))}))$$

We can write the factorial function in a similar manner:

$$\textit{Fact} = !(\textit{fact}(a,n).if isZero(n) then \overline{a}\langle 1\rangle else(\nu\ b)(\overline{\textit{fact}}\langle b, n-1\rangle \mid b(x).\overline{a}\langle n \cdot x\rangle))$$

Note that the functions defined above consume the number they interact with. This makes it impossible to define functions that use their arguments more than once. It is possible to define persistent versions of the booleans and natural numbers, by using replication, e.g. $\textit{True}(a) = !(a(t,f).\bar{t})$.

Using these examples, and a formal proof, we can see that π-calculus can express all computable functions. It is possible to prove this by defining a translation from λ-calculus into π-calculus, or by directly showing that all recursive functions on natural numbers can be defined in π-calculus.

There is an argument that π-calculus is more fundamental than λ-calculus because π-calculus can easily express functional behaviour in terms of communication, whereas modeling concurrency behaviour in λ-calculus would require elaborate encodings.