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

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

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:

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

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

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:

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

Control will either be $Control_1$ or $Control_2$. *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:

$$\begin{aligned} Control_1 &= \overline{lose_1}\langle talk_2, switch_2\rangle.\overline{gain_2}\langle talk_2, switch_2\rangle.Control_2 \\ Control_2 &= \overline{lose_2}\langle talk_1, switch_1\rangle.\overline{gain_1}\langle talk_1, switch_1\rangle.Control_1 \end{aligned}$$

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:

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

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

$$\begin{aligned} System_1 = (\nu\ & talk_1, switch_1, gain_1, lose_1, talk_2, switch_2, gain_2, lose_2) \\ & (Car(talk_1, switch_1) \mid Trans_1 \mid IdTrans_2 \mid Control_1) \end{aligned}$$

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

$$\begin{aligned} Trans_i &= Trans(talk_i, switch_i, gain_i, lose_i) \\ IdTrans_i &= IdTrans(gain_i, lose_i) \end{aligned}$$

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

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

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 $NewCar$:

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

We can now replace $Car(talk_1, switch_1)$ with:

$$(\nu\ start)(\overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar)$$

That way, we can reduce the following process:

$$\overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar \mid talk_1().0$$

$$\overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar$$

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

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

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

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

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

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

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

$$True(a) \mid Cond(P, Q)(a)$$

Similarly,

$$False(a) \mid Cond(P, Q)(a) \rightarrow^* Q$$

Next, we can define the process $Not(a, b)$, 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:

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

We illustrate this with the following reduction:

$$Not(a, b) \mid True(b)$$

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:

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

We illustrate this with the following reduction:

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

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

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

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

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

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

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

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:

$$IsZero(a, n) = Cases(True(a), False(a))(a, n).$$

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

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

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

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

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

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

It has the following reduction rules:

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

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

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

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

We can write the factorial function in a similar manner:

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

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

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.

2.3 Equivalence of Process

Part of the motivation for the topic of process calculus is to be able to reason about *communicating concurrent systems*: to specify how they behave and verify that they behave correctly.

The first step is to define the syntax and operational semantics (the reduction relation), which we have done. We have also seen how to define some interesting computational scenarios involving concurrency and communication.

We will now introduce a theory of equivalence of processes, which means equivalence of ongoing interactive behaviour (in contrast, to equivalence of a final result, as in λ-calculus).

Previously, we define the processes $\mathit{True}(x)$ and $\mathit{False}(x)$, and $\mathit{Not}(x, y)$. We would like to have a theory in which

$$(\nu\ y)(\mathit{Not}(x, y) \mid \mathit{True}(y)) = \mathit{False}(x).$$

The fact that $\mathit{False}(x)$ represents the boolean value *false* is to do with the way it interacts on the channel x, so the definition of equivalent must take that into account. Moreover, applying `Not` twice should be the identity function on booleans, i.e.

$$x(t, f).\overline{y}\langle t, f\rangle = (\nu\ z)(\mathit{Not}(x, z) \mid \mathit{Not}(z, y))$$

To develop the theory of equivalence, we will simplify π-calculus by assuming that all messages are empty. To reduce syntax, we will omit the empty messages and just write x or $\overline{x}$. We will refer to x and $\overline{x}$ as actions. This means that we are working in an earlier process calculus called CCS-calculus of Communication Systems. We will start with examples and then give proper definitions.

Reductions of the form $P \to Q$ tell us how a process evolves by *internal* communication. We need to be able to describe *potential* interaction between a process and its environment. We do this by introducing *labelled* transitions, e.g.

$$a.P \xrightarrow{a} P \qquad \overline{a}.Q \xrightarrow{\overline{a}} Q$$

We can describe actual interactions using τ transitions, e.g.

$$a.P \mid \overline{a}.Q \xrightarrow{\tau} P \mid Q$$

We will now define labelled transitions. Let α range over $\{a, \overline{a}, \tau\}$. The the inference rules for labelled transitions are given as follows:

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P' \mid Q} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} P \mid Q'}$$

$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \in \{x, \overline{x}\}}{(\nu\ x)P \xrightarrow{\alpha} (\nu\ x)P'}$$

$$\frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P} \qquad \frac{P \xrightarrow{\overline{a}} P' \quad P \xrightarrow{a} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

We can now talk about *traces* of a process, which are its possible sequences of actions. The empty trace ϵ is always a possibility. We look at some examples:

- $a.b.0$ has traces ϵ, a, ab;
- $a.0 + b.0$ has traces ϵ, a, b;
- $a.\overline{b}.0 + b.0$ has traces ϵ, a, b, $a\overline{b}$; and
- $a.b.0 + b.\overline{c}.0 + \overline{c}.0$ has traces ϵ, a, b, $\overline{c}$, ab, $b\overline{c}$.

We can show the labelled transitions of a process in a diagram, with or without the process terms. We can then read off the traces. For example, the process $a.\overline{b}.0 + b.0$ corresponds to the following figure:

A simple equivalence between processes is *trace equivalence*. We say that $P =_{tr} Q$ if they have the same traces. For example, $(\nu\ b)(a.\overline{b}.0 \mid b.c.0)$ and $a.\tau.c.0$ are trace equivalent since they have the same traces:

There is an issue with trace equivalence- it is too weak. To see this, consider the two processes $P = a.b.0 + a.c.0$ and $Q = a.(b.0 + c.0)$. They have the following trace diagrams: Both the processes have traces ϵ, a, ab and ac. Hence, they are trace equivalent. However, P can go into a state in which only one of b and c is available, whereas Q offers both b and c from the same state.

Next, define $R = \overline{a}.\overline{b}.0$. We have that $Q \mid R$ can communicate on a then b, and both processes reach 0. On the other hand, $P \mid R$ can communicate with a, with P becoming either $b.0$ or $c.0$. In the second case, no further communication is possible. This is a deadlock. We would like a definition of equivalence in which P and Q are *not* equivalent.

To do this, we define *bisimulation*. This concept is fundamental in process calculus and concurrency theory. There are many forms of it. We look at *strong bisimulation*. The idea is that for processes P and Q to be bisimilar, it must be that:

- for any action $\alpha \in \{a, \overline{a}, \tau\}$, if $P \xrightarrow{\alpha} P'$, then there is a Q' such that $Q \xrightarrow{\alpha} Q'$ and P' and Q' are bisimilar;
- this holds the other way as well, i.e. when $Q \xrightarrow{\alpha} Q'$, then there exists a P' bisimilar to Q' such that $P \xrightarrow{\alpha} P'$ as well.

We now consider the processes $P = (\nu\ b)(a.\overline{b}.0 \mid b.c.0)$ and $Q = a.\tau.c.0$. We previously saw that the two processes were trace equivalent. We will now see that they are also bisimilar, since they evolve in the following manner:

Now, we can work from the end to prove that P and Q are bisimilar. We use the symbol $\sim$ for strong bisimulation. In this case, $P_3 \sim Q_3$ because neither of them has any transitions; they are both structurally congruent to 0. So, there is nothing to check here. Now, consider P_2 and Q_2. We have $P_2 \xrightarrow{c} P_3$, which can be matched by $Q_2 \xrightarrow{c} Q_3$, and we know that $P_3 \sim Q_3$. In the other

direction, the only transition of Q_2 is $Q_2 \xrightarrow{c} Q_3$, which can equally be matched by $P_2 \xrightarrow{c} P_3$, with $P_3 \sim Q_3$.

Similarly, we can show that $P_1 \sim Q_1$- the only transition of P_1 is $P_1 \xrightarrow{\tau} P_2$ from communication on b. This can be matched by $Q_1 \xrightarrow{\tau} Q_2$, and we know that $P_2 \sim Q_2$. In the other direction, the only transition of Q_1 is $Q_1 \xrightarrow{\tau} Q_2$, which can be matched by $P_1 \xrightarrow{\tau} P_2$, and we know that $P_2 \sim Q_2$.

Finally, we consider the transitions of P and Q in the same way. We can see that each process has an a-transition that can match the a-transition of the other process, resulting in P_1 and Q_1, with $P_1 \sim Q_1$. We conclude that $P \sim Q$.

Now, consider $P = a.b.0 + a.c.0$ and $Q = a.(b.0 + c.0)$. We saw that P and Q were trace equivalent. However, they are not bisimilar- consider the trace diagram for P and Q. We see that P has two transitions- $P \xrightarrow{a} P_1$ and $P \xrightarrow{a} P_2$. We can match $P \xrightarrow{a} P_1$ by $Q \xrightarrow{a} Q_1$, so we would need $P_1 \sim Q_1$. Also, $P \xrightarrow{a} P_2$ is matched by $Q \xrightarrow{a} Q_1$. But, Q_1 has a transition $Q_1 \xrightarrow{b} 0$, which cannot be matched by P_2. So, we cannot have $P \sim Q$.

We have found two processes that are trace equivalent but not strongly bisimilar. Strong bisimulation is a better definition of equivalence when we have non-determinism. This is because it takes into account how a process can interact with other processes. If two processes are strongly bisimilar, then they are also trace equivalent. If two deterministic processes are trace equivalent, then they are also strongly bisimilar.

Note that the term 'strong' means that we include τ transitions when matching. Sometimes, this is undesirable because τ transitions are supposed to be internal unobservable actions. There is another form of equivalence, *weak bisimulation*, which ignores τ transitions.

We will now aim to define strong bisimulation. A first attempt might be the following:

Definition 2.3.1. We say that P and Q are *strongly bisimilar*, denoted $P \sim Q$ if for every action α:

- if $P \xrightarrow{\alpha} P'$, then there is a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$; and
- if $Q \xrightarrow{\alpha} Q'$, then there is a P' such that $P \xrightarrow{\alpha} P'$ and $P' \sim Q'$.

This definition is recursive in nature, but does not have a base case. Moreover, there is no reason that P' and Q' are 'smaller' than P and Q.

For instance, consider the process $P = !a.0$. We want $P \sim P$. Because $a.0 \xrightarrow{a} 0$, we have

$$P \xrightarrow{a} 0 \mid !a.0 \equiv 0 \mid P.$$

Applying the attempted definition of strong bisimulation to P and P, we see that they can match each other's actions. Checking the subsequent processes tells us that $P \sim P$ depends on $0 \mid P \sim 0 \mid P$. This is circular and not a valid definition. Moreover, the reduced process is equivalent to itself, meaning that it does not get smaller.

Instead of defining bisimulation for π-calculus processes, we can work with more abstract *labelled transition systems* (LTS). An LTS is a directed graph in which edges are labelled with actions. We use the transition notation

$$m \xrightarrow{a} n$$

where m and n are nodes in the graph. We can think of an LTS as a transition diagram for processes, in which the nodes have π-calculus processes associated with them.

The first step is to consider a general relation $\mathcal{R}$ on processes (or nodes of an LTS). If P_1 and P_2 are processes related by $\mathcal{R}$, we write $(P_1, P_2) \in \mathcal{R}$ or $P_1 \mathcal{R} P_2$. Now, we define the property of being a strong bisimulation, which is a property that a relation might or might not have.

Definition 2.3.2. A relation $\mathcal{R}$ on processes (or on nodes of an LTS) is a *strong bisimulation* if whenever $P\mathcal{R}Q$, for every action α, it is the case that

- if $P \xrightarrow{\alpha} P'$, then there is Q' such that $Q \xrightarrow{\alpha} Q'$, and $P'\mathcal{R}Q'$; and
- if $Q \xrightarrow{\alpha} Q'$, then there is P' such that $P \xrightarrow{\alpha} P'$, and $P'\mathcal{R}Q'$.

Consider a labelled transition system as follows.

- The empty relation $\varnothing$ is a strong bisimulation- this is trivially the case.
- The relation $\{(P_2, Q_2)\}$ is not a strong bisimulation- we have $P_2 \xrightarrow{b} P_2$, but in Q_2, the only b-transition is to Q_3, but $(P_2, Q_3) \notin \mathcal{R}$.
- The relation $\{(P_2, P_2)\}$ is a strong bisimulation since it can only reduce to itself.
- The relation $\{(P_2, Q_2), (P_2, Q_3)\}$ is a strong bisimulation- we have $P_2 \xrightarrow{b} P_2$ matching $Q_2 \xrightarrow{b} Q_3$ and $Q_3 \xrightarrow{b} Q_2$; and $P_2 \xrightarrow{a} P_2$ matching $Q_2 \xrightarrow{a} Q_2$ and $Q_3 \xrightarrow{a} Q_3$.
- The relation $\{(P_2, Q_2), (P_2, Q_3), (Q_2, P_2), (Q_3, P_2)\}$ is a strong bisimulation- we have added the symmetrical cases.
- The relation $\{(P_1, Q_1), (P_2, Q_2), (P_2, Q_3)\}$ is a strong bisimulation since $P_1 \xrightarrow{c} P_2$ is matched by $Q_1 \xrightarrow{c} Q_2$.
- The relation $\{(P_1, Q_1), (P_2, Q_2), (P_2, Q_3), (Q_1, P_1), (Q_2, P_2), (Q_3, P_2)\}$ is a strong bisimulation.

For a given LTS (or a set of processes), there can be several relations that satisfy the conditions to be a strong bisimulation. We say that *strong bisimilarity*, denoted by $\sim$, is the largest strong bisimulation. We say that processes P and Q are strong bisimilar, denoted $P \sim Q$, if (P, Q) is in the largest strong bisimulation. To prove this, it is sufficient to find any bisimulation containing (P, Q).

We have previously used inductive definitions, e.g. the definitions of reduction and typing judgments. An inductive definition is when we define a set or relation to be the smallest entity satisfying certain conditions. Ordinary recursive function definitions are also in this category. A *coninductive* definition is when we define a set or a relation to be the largest entity satisfying certain conditions, such as strong bisimilarity.

We will now look at processes in the syntax π-calculus, rather than abstract LTS. When defining labelled transitions for π-calculus, we did not take structural congruence into account. So, if we want to show something like $P \mid Q$ and $Q \mid P$ to be bisimilar, we have to prove it by constructing a strong bisimulation that contains $(P \mid Q, Q \mid P)$.

Proposition 2.3.3. *Let P and Q be processes. Then $P \mid Q \sim Q \mid P$.*

Proof. To prove this, it suffices to show that there is a relation containing $(P \mid Q, Q \mid P)$. So, define the relation

$$\mathcal{R} = \{(P \mid Q, Q \mid P) \mid P, Q \text{ processes}\}.$$

We show that $\mathcal{R}$ is a strong bisimulation. So, let $(P \mid Q, Q \mid P) \in \mathcal{R}$ There are 3 transitions for $P \mid Q$:

- First assume that $P \mid Q \xrightarrow{\alpha} P' \mid Q$ because $P \xrightarrow{\alpha} P'$. In that case, we find that $Q \mid P \xrightarrow{\alpha} Q \mid P'$ by definition of parallel composition. We know that $(P' \mid Q, Q \mid P') \in \mathcal{R}$ by definition of $\mathcal{R}$.
- The same result follows if $P \mid Q \xrightarrow{\alpha} P \mid Q'$ because $Q \xrightarrow{\alpha} Q'$.
- Finally, assume that P is $a.P_1 + P_2$ and Q is $\overline{a}.Q_1 + Q_2$, and $P \mid Q \xrightarrow{\tau} P_1 \mid Q_1$. Then, we also have $Q \mid P \xrightarrow{\tau} Q_1 \mid P_1$, and we have $(P_1 \mid Q_1, Q_1 \mid P_1) \in \mathcal{R}$.

So, it follows that $\mathcal{R}$ is a strong bisimulation. Since $(P \mid Q, Q \mid P) \in \mathcal{R}$, we conclude that $P \mid Q \sim Q \mid P$. □

We can do the same thing to show that structural congruence is included within strong bisimulation.

Now, let $P = (\nu\ b, c)((a.b.0 + c.d.0) \mid \overline{b}.0)$ and $Q = a.\tau.0$. We show that P and Q are strongly bisimilar. In P, b and c are private channels, so we cannot transition using these channels. This means that our only option is:

$$P \xrightarrow{a} P_1 = (\nu\ b, c)(b.0 \mid \overline{b}.0).$$

Then, we have

$$P_1 \xrightarrow{\tau} P_2 = (\nu\ b, c)(0 \mid 0).$$

So, the relation we want is:

$$\mathcal{R} = \{(P, Q), (P_1, \tau.0), (P_2, 0)\}.$$

We note that P_2 and 0 are strongly bisimilar since they has no transitions. This means that $\mathcal{R}$ is a strong bisimulation. Hence, P and Q are strongly bisimilar.

There are many extensions to strong bisimulation, including:

- allowing messages, including mobile names- this requires a more complex version of the labelled transition rules; and
- defining *weak bisimulation*, which ignores τ-transitions. This allows internal interaction to be ignored and focuses on observable input-output.