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 *process algebra*. Here, *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.

In π-calculus, 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.

Informal Definition of 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 and then receive y.
- $\overline{a}\langle x+y\rangle$ 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 internally 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* process $+$ gives us a choice between two different ways of communication. For instance, consider the following term:

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

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

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

reduces in one step to

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

We illustrate the choice operation for the following process:

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

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

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

We can always replace A by its definition. We now consider an example of reduction with recursion:

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

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:

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

We can either have the first two processes interact, so that we reduce to:

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

or we can have the first and the last process interact, in which case we reduce to:

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

Now, for the process to not get stuck, we need to ensure that the channel a receives the two messages 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 $(\nu\, x)(\dots)$. As we can see, 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 of the channel b 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:

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

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.

Formal definition of π-calculus

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:

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

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

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

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

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

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

It might be the case that the processes communicating might not be next to each other; we would like the reduction to occur in that stage as well. Due to the pedantic nature of syntax definitions, the rule above is not enough to accommodate that case.

To allow processes to communicate even when they are not adjacent, we define *structural congruence* (denoted $\equiv$) 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. It is 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.

The rules for congruence are given below:

- 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 $!P \equiv !Q$.

To make the congruence an equivalence relation, we also require:

- $P \equiv P$;
- 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
$P + (Q + R) \equiv (P + Q) + R$	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 (renaming of bound variables). Due to the recursive nature, structural congruence is the *smallest* congruence relation that includes α-equivalence. Informally, the definition states that:

- because of commutativity, we can ignore the order of processes in parallel and choice constructs;
- because of associativity, we do not need to write brackets in parallel and choice constructs;
- we can reorder ν binders;
- because of garbage collection, 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.

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

- substitution (e.g. $P[x := y]$) 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*, like in λ-calculus.

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 dots $(\dots)$ represent an arbitrary process. The RCom axiom allows us to substitute via communication, while the RTau axiom takes the τ choice in composition. We extend these rules to allow reduction in any context:

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

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

2.2 Modelling and Computation

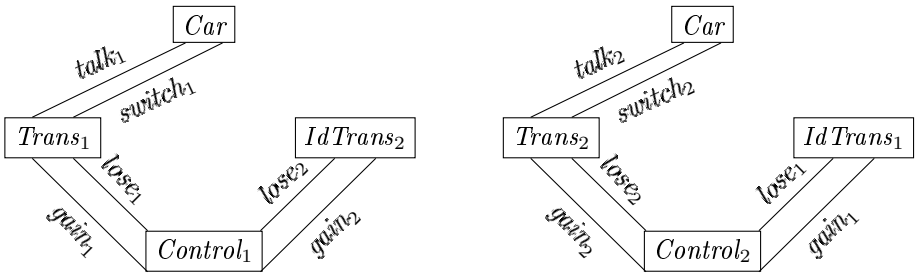
There are two directions we can go with π-calculus:

- 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 we did for λ-calculus.

Modelling in π-calculus

The mobile phones example is a classic π-calculus example that illustrates *dynamic* communication topology. This is because 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. This is summarised in the figure below.



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:

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

Note that the process *talk*() receives no messages- its mere purpose is synchronisation.

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:

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

When we put everything together, the $lose_i$, $talk_i$ and $switch_i$ channels will be in scope for $i \in \{1, 2\}$.

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:

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

Finally, we define $System_1$, with the starting state $Trans_1$. This is given by:

$$System_1 = (\nu\ talk_1, switch_1, gain_1, lose_1, talk_2, switch_2, gain_2, lose_2)$$
$$(Car(talk_1, switch_1) \mid Trans_1 \mid IdTrans_2 \mid Control_1)$$

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

$$Trans_i = Trans(talk_i, switch_i, gain_i, lose_i)$$
$$IdTrans_i = IdTrans(gain_i, lose_i)$$

We can define $System_2$ in an analogous manner.

Using this definition, we can reduce $System_1$ reduces to $System_2$:

$$Car(talk_1, switch_1) \mid Trans_1 \mid IdTrans_2 \mid Control_1$$
$$\equiv$$
$$Car(talk_1, switch_1) \mid \cdots + lose(t, s).\cdots \mid IdTrans_2 \mid \overline{lose_1}\langle talk_2, switch_2\rangle \ldots$$
$$\downarrow$$
$$\cdots + switch(t, s).\ldots \mid \overline{switch}\langle talk_2, switch_2\rangle.\cdots \mid IdTrans_2 \mid \overline{gain_2} \ldots$$
$$\downarrow$$
$$Car(talk_2, switch_2) \mid IdTrans_1 \mid gain_2(t, s)\cdots \mid \overline{gain_2}\langle talk_2, switch_2\rangle \ldots$$
$$\downarrow$$
$$Car(talk_2, switch_2) \mid IdTrans_1 \mid Trans_2 \mid Control_2$$

We defined *Car* as follows:

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

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*:

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

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:

$$\begin{aligned}
&\overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar \mid talk_1().0 \\
&\downarrow \\
&\overline{start}\langle talk_1, switch_1\rangle.0 \mid start(t,s)\ldots \mid NewCar \mid talk_1().0 \\
&\downarrow \\
&0 \mid \overline{talk}\langle\rangle \cdots + \ldots \mid NewCar \mid talk_1().0 \\
&\downarrow \\
&0 \mid \overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar \mid 0 \\
&\equiv \\
&\overline{start}\langle talk_1, switch_1\rangle.0 \mid NewCar
\end{aligned}$$

During the process, we were able to *talk* to the car.

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:

$$\begin{aligned}
&True(a) \mid Cond(P,Q)(a) \\
&\equiv \\
&(\nu\ t,f)(a(t,f).\bar{t} \mid \overline{a}\langle t,f\rangle.(t.P + f.Q)) \\
&\downarrow \\
&(\nu\ t,f)(\bar{t} \mid t.P + f.Q) \\
&\downarrow \\
&(\nu\ t,f)(0 \mid P) \\
&\equiv \\
&P
\end{aligned}$$

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:

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

We illustrate this with the following reduction:

$$\begin{aligned} & Not(a, b) \mid True(b) \\ & \equiv \\ & (\nu\ t, f)(\overline{b}\langle t, f\rangle.(t.False(a) + f.True(a)) \mid b(t, f).\overline{t}) \\ & \downarrow \\ & (\nu\ t, f)(t.False(a) + f.True(a) \mid \overline{t}) \\ & \downarrow \\ & (\nu\ t, f)(False(a) \mid 0) \\ & \equiv \\ & False(a) \end{aligned}$$

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

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

We can reduce to find that

$$\begin{aligned} And(a, b, c) \mid True(b) \mid True(c) &\rightarrow^* True(a) \\ And(a, b, c) \mid True(b) \mid False(c) &\rightarrow^* False(a) \end{aligned}$$

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:

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

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

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

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

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

We illustrate how the process works with two examples, first running it in parallel with zero:

$$\begin{aligned}
&IsZero(a, n_0) \mid Z(n_0) \\
&\equiv \\
&Cases(True(a), False(a))(n_0, m) \mid n_0(z, s).\overline{z} \\
&\equiv \\
&(\nu\ z, s)(\overline{n_0}\langle z, s\rangle.(z.True(a) + s(m).False(a)) \mid n_0(z, s).\overline{z}) \\
&\downarrow \\
&(\nu\ z, s)(z.True(a) + s(m).False(a) \mid \overline{z}) \\
&\downarrow \\
&(\nu\ z, s)(True(a) \mid 0) \\
&\equiv \\
&True(a)
\end{aligned}$$

Now, we run *isZero* in parallel with *One*:

$$\begin{aligned}
&IsZero(a, n_1) \mid One(n_1) \\
&\equiv \\
&(\nu\ z, s, n_1)(\overline{n_1}\langle z, s\rangle.(z.True(a) + s(m).False(a)) \mid n_1(z, s).\overline{s}\langle n_0\rangle \mid n_0(z, s).\overline{z}) \\
&\downarrow \\
&(\nu\ z, s, n_1)(z.True(a) + s(m).False(a) \mid \overline{s}\langle n_0\rangle \mid n_0(z, s).\overline{z}) \\
&\downarrow \\
&(\nu\ z, s, n_1)(False(a) \mid 0 \mid n_0(z, s).\overline{z}) \\
&\equiv \\
&False(a) \mid n_0(z, s).\overline{z}
\end{aligned}$$

We have gotten the *False* result, and the two remaining processes cannot interact.

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. We run the process on some number N at n_k using the process

$$(\nu\ n_k)(\overline{even}\langle a, n_k\rangle \mid N(n_k) \mid Even)$$

For instance, we can run it on the zero process as follows:

$$\begin{aligned}
&\overline{even}\langle a, n_0\rangle \mid Z(n_0) \mid Even \\
&\equiv \\
&\overline{even}\langle a, n_0\rangle \mid n_0(z,s).\overline{z} \mid even(a, n_0)\ldots \mid Even \\
&\downarrow \\
&(\nu\ z,s)(0 \mid n_0\langle z,s\rangle.\overline{z} \mid \overline{n_0}\langle z,s\rangle\ldots) \mid Even \\
&\downarrow \\
&(\nu\ z,s)(z \mid z.True(a) + \ldots) \mid Even \\
&\downarrow \\
&(\nu\ z,s)(0 \mid True(a)) \mid Even \\
&\equiv \\
&True(a) \mid Even
\end{aligned}$$

Next, we run it on the *One* process:

$$\begin{aligned}
&\overline{even}\langle a, n_1\rangle \mid One(n_1) \mid Even \\
&\equiv \\
&(\nu\ n_1)(\overline{even}\langle a, n_1\rangle \mid n_1(z,s).\overline{s}\langle n_0\rangle \mid n_0(z,s).\overline{z} \mid even(a, n_1)\ldots) \mid Even \\
&\downarrow \\
&(\nu\ z,s,n_1)(0 \mid n_1(z,s).\overline{s}\langle n_0\rangle \mid n_0(z,s).\overline{z} \mid n_1\langle z,s\rangle\ldots) \mid Even \\
&\downarrow \\
&(\nu\ b,z,s,n_1)(\overline{s}\langle n_0\rangle \mid n_0(z,s).\overline{z} \mid \cdots + s(m).(Not(a,b) \mid \overline{even}(b,m))) \mid Even \\
&\downarrow \\
&(\nu\ z,s,n_1)(0 \mid n_0(z,s).\overline{z} \mid Not(a,b) \mid \overline{even}(b,n_0)) \mid Even \\
&\equiv \\
&(\nu\ b)(Not(a,b) \mid \overline{even}\langle b,n_0\rangle \mid Z(n_0)) \mid Even \\
&\downarrow_* \\
&(\nu\ b)(Not(a,b) \mid True(b)) \mid Even \\
&\downarrow_* \\
&False(a) \mid Even
\end{aligned}$$

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

$$if\ e\ then\ P\ else\ Q.$$

It has the following reduction rules:

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

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

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

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

We can write the factorial function in a similar manner:

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

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

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 was to define the syntax and operational semantics, i.e. the reduction relation. 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. This is in contrast with λ-calculus, where we had an equivalence of the final result.

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.

Trace Equivalence

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 $\alpha \in \{a, \overline{a}, \tau\}$. Then, the inference rules for labelled transitions are given as follows:

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

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

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

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

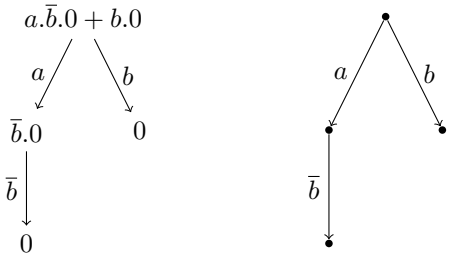
$$\frac{P \xrightarrow{\alpha} P' \quad \alpha \notin \{y, \overline{y}\}}{(\nu\ y)P \xrightarrow{\alpha} (\nu\ y)P'}$$

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

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:

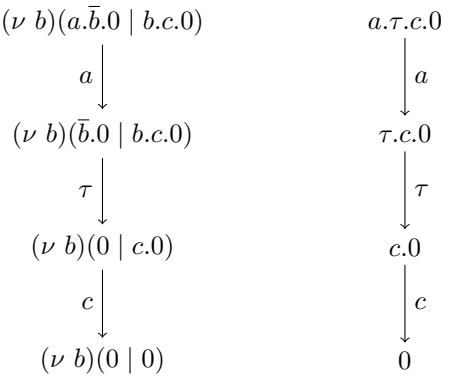
- $a.b.0$ has traces ϵ, a, ab;
- $a.0 + b.0$ has traces ϵ, a, b;
- $a.\overline{b}.0 + b.0$ has traces ϵ, a, b, $a\overline{b}$; and
- $a.b.0 + b.\overline{c}.0 + \overline{c}.0$ has traces ϵ, a, b, $\overline{c}$, ab, $b\overline{c}$.

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:



On the left, we highlight all the processes, while on the right, we only show the traces.

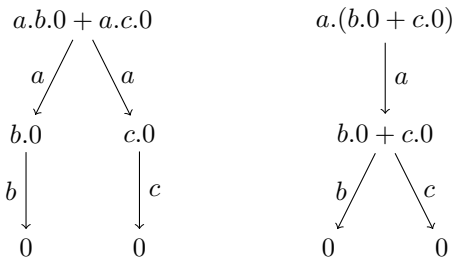
A simple equivalence between processes is *trace equivalence*. We say that $P =_{tr} Q$ if they have the same traces. For example, $(\nu\ b)(a.\overline{b}.0 \mid b.c.0)$ and $a.\tau.c.0$ are trace equivalent since they have the same traces:



Since the channel b is private, note that $(\nu\ b)(a.\overline{b}.0 \mid b.c.0)$ cannot have a b-trace.

Informal introduction to bisimulation

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, after an a-reduction, P can go into a state in which only one of b and c is available, whereas Q offers both b and c.

We illustrate the issue further. Define $R = \overline{a}.\overline{b}.0$. If we have the process $Q \mid R$, then we can reduce it as follows:

$$\begin{array}{c} a.(b.0 + c.0) \mid \overline{a}.\overline{b}.0 \\ \downarrow \\ b.0 + c.0 \mid \overline{b}.0 \\ \downarrow \\ 0 \mid 0 \end{array}$$

This is the only reduction possible from $Q \mid R$. So, every reduction leads to termination in this case. On the other hand, if we have the process $P \mid R$, it could reduce in two ways:

$$\begin{array}{c} a.b.0 + a.c.0 \mid \overline{a}.\overline{b}.0 \\ \downarrow \\ c.0 \mid \overline{b}.0 \end{array} \qquad \begin{array}{c} a.b.0 + a.c.0 \mid \overline{a}.\overline{b}.0 \\ \downarrow \\ b.0 \mid \overline{b}.0 \\ \downarrow \\ 0 \mid 0 \end{array}$$

Since $\overline{a}.\overline{b}.0$ can communicate with $a.c.0$ at the top level, there is nothing stopping the left reduction from happening. However, this reduction leads to a deadlock. On the other hand, no reduction of $Q \mid R$ leads to a deadlock. We would like the equivalence of process to be deadlock-invariant, i.e. if two processes are equivalent, then either they both can reduce to a deadlock or neither can.

To do this, we define *bisimulation*. This concept is fundamental in process calculus and concurrency theory.

There are many forms of bisimulation. 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 have the same trace diagram:

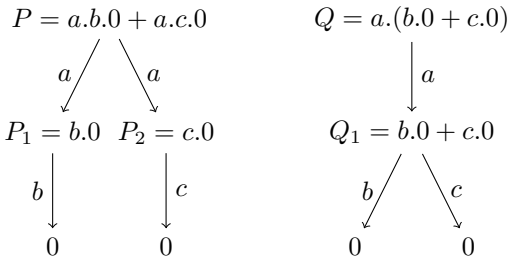
$$
\begin{array}{ccc}
P = (\nu\ b)(a.\overline{b}.0 \mid b.c.0) & & Q = a.\tau.c.0 \\
\Big\downarrow a & & \Big\downarrow a \\
P_1 = (\nu\ b)(\overline{b}.0 \mid b.c.0) & & Q_1 = \tau.c.0 \\
\Big\downarrow \tau & & \Big\downarrow \tau \\
P_2 = (\nu\ b)(0 \mid c.0) & & Q_2 = c.0 \\
\Big\downarrow c & & \Big\downarrow c \\
P_3 = (\nu\ b)(0 \mid 0) & & Q_3 = 0
\end{array}
$$

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 $P \sim Q$. Note that b is a private channel in P_1, meaning that P_1 cannot fire a b-transition.

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 that 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.

Formal definition of bisimulation

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.

We illustrate the issue further. So, consider the process $P = !a.0$. We would like $P \sim P$. Because $a.0 \xrightarrow{a} 0$, we have

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

Applying the above definition of strong bisimulation to P both on the left and the right, 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 the reduced process 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

$$m \xrightarrow{a} n$$

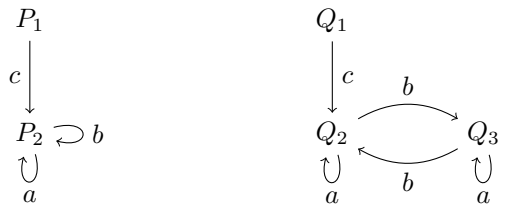
where m and n are nodes in the graph. We can think of an LTS as a transition diagram for processes, in which the nodes have π-calculus processes associated with them.

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 the labelled transition system given below.



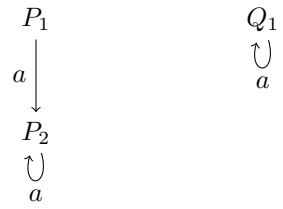
We consider which relations are bisimilar with respect to this LTS.

- 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 match all of its transitions.
- 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 $Q_1 \xrightarrow{c} Q_2$, with $(P_2, Q_2) \in \mathcal{R}$.
- 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. Adding reflexivity (i.e. (P, P) for all processes) to this relation would give us the largest possible set satisfying 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 *strongly 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).

Now, consider the following LTS:



Then, the following are relations satisfying strong bisimulation:

- $\{(P_1, P_2), (P_2, P_2)\}$- we can match $P_1 \xrightarrow{a} P_2$ with $P_2 \xrightarrow{a} P_2$, and we have $(P_2, P_2) \in \mathcal{R}$, and vice versa;
- $\{(P_2, Q_1)\}$- we can match $P_2 \xrightarrow{a} P_2$ with $Q_1 \xrightarrow{a} Q_1$, with $(P_2, Q_1) \in \mathcal{R}$;
- $\{(P_1, Q_1), (P_2, Q_1)\}$- we can match $P_1 \xrightarrow{a} P_2$ with $Q_1 \xrightarrow{a} Q_1$, with $(P_2, Q_1) \in \mathcal{R}$.

By reflexivity and symmetry, the largest strong bisimulation is:

$$\{(P_1, P_1), (P_2, P_2), (Q_1, Q_1), (P_1, P_2), (P_2, P_1), \\ (P_1, Q_1), (Q_1, P_1)(P_2, Q_1), (Q_2, P_1)\}.$$

This implies that every process is strongly bisimilar in this case.

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 know that $(P_1 \mid Q_1, Q_1 \mid P_1) \in \mathcal{R}$.

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. For instance, we can show that $P \mid 0 \sim P$ for all processes P.

Proposition 2.3.4. *Let P be a process. Then, $P \mid 0 \sim P$.*

Proof. Define the relation

$$\mathcal{R} = \{(P \mid 0, P) \mid P \text{ process}\}.$$

We show that $\mathcal{R}$ is a strong bisimulation. So, let $(P \mid 0, P) \in \mathcal{R}$.

- First, let $P \xrightarrow{\alpha} P'$. In that case, the parallel rule tells us that $P \mid 0 \xrightarrow{\alpha} P' \mid 0$. By definition, we have $(P' \mid 0, P') \in \mathcal{R}$.
- Now, let $P \mid 0 \xrightarrow{\alpha} Q$. We know that 0 has no transitions, so the only option we have is $Q \equiv P' \mid 0$. Hence, by the inference rule for transition, we must have $P \xrightarrow{\alpha} P'$. By definition, we have $(Q, P') = (P' \mid 0, P') \in \mathcal{R}$.

So, $\mathcal{R}$ is a strong bisimulation. □

Now, let $P = (\nu\ b, c)((a.b.0 + c.b.0) \mid \overline{b}.0)$ and $Q = a.\tau.0$. We show that P and Q are strongly bisimilar. So, consider the trace diagram for P and Q below.

$$\begin{array}{ccc} P = (\nu\ b, c)((a.b.0 + c.b.0) \mid \overline{b}.0) & & Q = a.\tau.0 \\ \Big\downarrow a & & \Big\downarrow a \\ P_1 = (\nu\ b, c)(b.0 \mid \overline{b}.0) & & Q_1 = \tau.0 \\ \Big\downarrow \tau & & \Big\downarrow \tau \\ P_2 = (\nu\ b, c)(0 \mid 0) & & Q_2 = 0 \end{array}$$

In this case, the relation we want is:

$$\mathcal{R} = \{(P, Q), (P_1, Q_1), (P_2, Q_2)\}.$$

We note that P_2 and Q_2 are strongly bisimilar since they have no transitions. Moreover, since the other processes have just one transition, with the coupled result in $\mathcal{R}$, we conclude that $\mathcal{R}$ is a strong bisimulation. Hence, P and Q are strongly bisimilar.

There are many extensions to 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.

2.4 Types in π-calculus

In general, static typechecking aims to eliminate errors. In terms of π-calculus, we want to ensure that whenever a message is sent, the sender and receiver agree on its structure. In particular, we would like to eliminate processes such as:

- $\overline{a}\langle u, v\rangle.0 \mid a(x).P$, where the two subprocesses do not agree on the number of parameters; and
- $\overline{a}\langle \mathit{true}, \mathit{false}\rangle.0 \mid a(x, y).\overline{a}\langle x + y\rangle.0$, where the two subprocesses do not agree on type.

We will now look at *simple* type system for π-calculus. It deals with these two issues by specifying:

- the number of components of messages on the channel and
- the type of each component.

We illustrate this with an example. So, assume that the channel a is of type $\texttt{Chan}[\mathit{int}, \mathit{bool}]$, denoted by $a\colon \texttt{Chan}[\mathit{int}, \mathit{bool}]$. Then,

- $\overline{a}\langle 1 + 2, \mathit{true}\rangle$ is well-typed;
- $a(x, y).P$ is well-typed if $x\colon \mathit{int}$ and $y\colon \mathit{bool}$ in P; while
- $\overline{a}\langle 1\rangle$, $\overline{a}\langle \mathit{true}, 1, \mathit{false}\rangle$ and $a(x)$ are incorrect.

Instead, if $a\colon \texttt{Chan}[\texttt{Chan}[\mathit{int}]]$, then $a(x).x(y).Q$ is well-typed if $y\colon \mathit{int}$ in Q.

Formally, types are specified by the following grammar:

$$\begin{aligned} T ::= {} & \texttt{Chan}[T, \ldots, T] \\ \mid {} & \mathit{bool} \\ \mid {} & \ldots \end{aligned}$$

where the final line can be used for any types we would want the π-calculus to have, e.g. *int*.

As for λ-calculus, we modify the syntax of processes so that binding occurences are annotated with types, e.g.

$$a(x_1\colon T_1, \ldots, x_n\colon T_n) \qquad (\nu\ x\colon T)$$

We will consider *polyadic π-calculus*. This means that a message has zero or more components. We will also assume that the expression language has a type system with types *int* and *bool*, along with appropriate reduction relations involving them.

Now, we define the modified syntax of π-calculus, including processes P, values v and expressions e:

$$
\begin{array}{lr}
P ::= 0 & \text{terminated process} \\
\quad \mid \overline{x}\langle e_1, \ldots, e_n \rangle.P & \text{output/send} \\
\quad \mid x(y_1 : T_1, \ldots, y_n : T_n).P & \text{input/receive} \\
\quad \mid (\nu\ x : T)P & \text{scope/restriction} \\
\quad \mid P \mid Q & \text{parallel composition} \\
\quad \mid P + Q & \text{choice} \\
\quad \mid \ldots & \\
v ::= x & \text{channel names} \\
\quad \mid \mathit{true}, \mathit{false} & \text{boolean values} \\
\quad \mid \ldots & \\
e ::= v & \text{values} \\
\quad \mid e == e & \text{equality expressions} \\
\quad \mid \ldots &
\end{array}
$$

The dots $(\ldots)$ allow for expansion of the syntax as required.

Assuming *call-by-value*, the reduction rule for communication, with polyadic messages, is:

$$
\begin{array}{c}
(x(y_1 : T_1, \ldots, y_n : T_n).P + \ldots) \mid (\overline{x}\langle v_1, \ldots, v_n \rangle.Q + \ldots) \\
\downarrow \\
P[y_1 := v_1, \ldots, y_n := v_n] \mid Q
\end{array}
$$

If $n = 0$, then we do not send/receive any message- this is synchronisation. If we have a send and a receive on the same channel with different number of components in message, then there is no reduction. This is a *communication error*. We define the type system in a way that avoids such errors.

Let $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ be a *typing context* or *typing environment*. We have typing judgment for processes and expressions.

- The *typing judgment for processes* is $\Gamma \vdash P$ (i.e. under Γ, P is a well-typed process). A process does not have a type- the judgment just says that it uses channels in Γ correctly.
- The *typing judgment for expressions* is $\Gamma \vdash e : T$ (i.e. under Γ, the expression e has type T). This is similar to simply typed λ-calculus. It includes the case where e is just a channel name.

We now define the typing rules. The process 0 is well-typed in any environment:

$$\frac{}{\Gamma \vdash 0}\text{TNil}.$$

Well-typed processes can be put in parallel, or used as alternatives.

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q}\text{TPar} \qquad \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P + Q}\text{TPlus}$$

An output must send a message of the correct structure.

$$\frac{\Gamma \vdash x\colon \texttt{Chan}[T_1, \ldots, T_n] \quad \Gamma \vdash e_i\colon T_i \ \forall 1 \leq i \leq n \quad \Gamma \vdash P}{\Gamma \vdash \overline{x}\langle e_1, \ldots, e_n\rangle.P}\text{TOut}$$

In an input, the message must have the correct structure, and its components must be used correctly.

$$\frac{\Gamma \vdash x\colon \texttt{Chan}[T_1, \ldots, T_n] \qquad \Gamma, y_1\colon T_1, \ldots, y_n\colon T_n \vdash P}{\Gamma \vdash x(y_1\colon T_1, \ldots, y_n\colon T_n).P}\text{TIn}$$

When a new channel is created, it must be used correctly.

$$\frac{\Gamma, x\colon \texttt{Chan}[T_1, \ldots, T_n] \vdash P}{\Gamma \vdash (\nu\ x\colon \texttt{Chan}[T_1, \ldots, T_n])P}\text{TNew}$$

We now illustrate the types with an example. So, consider the following processes:

$$\textit{Server} = a(x).x(y).\overline{x}\langle y + 1\rangle.0$$
$$\textit{Client} = (\nu\ c)\overline{a}\langle c\rangle.\overline{c}\langle 2\rangle.c(z).0$$

We can see that:

- a has type $\texttt{Chan}[\texttt{Chan}[\textit{int}]]$;
- x has type $\texttt{Chan}[\textit{int}]$;
- c has type $\texttt{Chan}[\textit{int}]$;
- y (and $y + 1$) has type *int*; and
- z has type *int*.

So, we can define Γ to be the typing context with all these types. Then, $\Gamma \vdash \textit{Server} \mid \textit{Client}$.

The simple type system for π-calculus has issues. In particular,

- deadlock procesess are typeable, e.g. $\overline{b}\langle \textit{true}\rangle.0 \mid c(y\colon \textit{int}).0$; and
- recursive processes are not typeable, even though they should be.

Nonetheless, we will show that the simple type system is safe. This can be done by showing:

- *type preservation*: if $\Gamma \vdash P$ and $P \to Q$, then $\Gamma \vdash Q$;
- *absence of immediate communication errors*: if $\Gamma \vdash P$ at the top level, P does not have an input and an output in parallel on the same channel with different number of message components (and types).

Proving these theorems allows us to show that well-typed processes do not generate communication errors.

To prove these theorems, we need some auxiliary results.

Lemma 2.4.1 (Substitution in Processes)**.** *If* $\Gamma, x_1, : T_1, \ldots, x_n : T_n$ *and for each* $1 \leq i \leq n$, $\Gamma \vdash v_i : T_i$, *then*

$$\Gamma \vdash P[x_1 := v_1, \ldots, x_n := v_n].$$

If v_i are only allowed to be channel names, then we can prove this by induction on the derivation of $\Gamma \vdash P$.

Next, to handle values in the expression language, we need a similar lemma for substitution in expressions.

Lemma 2.4.2 (Substitution in Expressions)**.** *If* $\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash e : T$ *and for* $1 \leq i \leq n$, $\Gamma \vdash v_i : T_i$, *then*

$$\Gamma \vdash e[x_1 := v_1, \ldots, x_n := v_n] : T.$$

This can also be proved using induction.

Because reduction allows the use of structural congruence, we need to prove that this does not affect typeability.

Lemma 2.4.3. *If* $\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q$.

We can prove this by structural induction on $\equiv$. In particular, we consider that P is typeable and then show that Q is typeable, and vice versa.

Proof for scope expansion. The rule for structural congruence tells us that for processes P and Q, and a channel $x \notin FV(P)$,

$$(\nu\ x)(P \mid Q) \equiv P \mid (\nu\ x)Q.$$

If the left hand side is typeable, then we have

$$\frac{\dfrac{\Gamma, x : T \vdash P \quad \Gamma, x : T \vdash Q}{\Gamma, x : T \vdash P \mid Q}}{\Gamma \vdash (\nu\ x : T)(P \mid Q)}$$

using TPar and then TNew. Hence, we need $\Gamma, x : T \vdash P$ and $\Gamma, x : T \vdash Q$ for the left hand side to be typeable. Since $x \notin FV(P)$, we can show that $\Gamma \vdash P$. Hence,

$$\frac{\dfrac{\Gamma, x : T \vdash Q}{\Gamma \vdash P \quad \Gamma \vdash (\nu\ x : T)Q}}{\Gamma \vdash P \mid (\nu\ x : T)Q}$$

using TNew and then TPar. So, if the left hand side is typeable, then so is the right hand side. Using the reverse argument, we can show that if the right hand side is typeable, then so is the left hand side. □

We now prove type preservation.

Theorem 2.4.4 (Type Preservation)**.** *If* $\Gamma \vdash P$ *and* $P \to Q$, *then* $\Gamma \vdash Q$.

We prove this using structural induction on $P \to Q$, by considering each case for the reduction step. In each case, we then analyse the structure of the

derivation of $\Gamma \vdash P$ and use the component parts to show that $\Gamma \vdash Q$. The key case is that of communication, where we have

$$x(y_1 : T_1, \ldots, y_n : T_n).P \mid \overline{x}\langle v_1, \ldots, v_n\rangle.Q \to P[y_1 := v_1, \ldots, y_n := v_n] \mid Q.$$

We ignore non-deterministic choice for simplicity. The typing derivation of the left hand side has the following form:

$$\dfrac{\dfrac{\Gamma \vdash x : \texttt{Chan}[T_1, \ldots, T_n] \quad \Gamma, y_1 : T_1, \ldots, y_n : T_n \vdash P}{\Gamma \vdash x(y_1 : T_1, \ldots, y_n : T_n).P}\text{TIn} \qquad \dfrac{\Gamma \vdash x : \texttt{Chan}[T_1, \ldots, T_n] \quad \Gamma \vdash v_1 : T_1, \ldots, v_n : T_n \quad \Gamma \vdash Q}{\Gamma \vdash \overline{x}\langle v_1, \ldots, v_n\rangle.Q}\text{TOut}}{\Gamma \vdash x(y_1 : T_1, \ldots, y_n : T_n).P \mid \overline{x}\langle v_1, \ldots, v_n\rangle.Q}\text{TPar}$$

Using the Substitution Lemma 1 and the typing derivations, we get that

$$\Gamma \vdash P[y_1 := v_1, \ldots, y_n := v_n]$$

Using TPar, we can derive that

$$\Gamma \vdash P[y_1 := v_1, \ldots, y_n := v_n] \mid Q,$$

hence the right hand side is well-typed. The other cases use induction in a similar manner.

We will now give a precise statement that *a well-typed process does not have immediate communication errors*. First, we can use scope expansion to write any process in the form

$$(\nu\ x_1) \ldots (\nu\ x_n)(P_1 \mid \cdots \mid P_n),$$

where immediate communications among the P_i are at the top level and not within ν operators.

Theorem 2.4.5 (Absence of immediate communication errors)**.** *If*

$$\Gamma \vdash (\nu\ x_1 : T_1, \ldots, x_k : T_k)(x(y_1 : U_1, \ldots, y_r : U_r).P \mid \overline{x}\langle v_1, \ldots, v_n\rangle.Q)$$

then

- $n = r$ *and*
- $\Gamma \vdash v_i : U_i$ *for all* $1 \leq i \leq n$.

So, the number of message components is the same as in the input and the output, and the messages/values components are of the types expected by the input.

Proof. The typing derivation must have the form

$$\Gamma_1 = \Gamma, x_1 : T_1, \ldots, x_n : T_m.$$

Considering $\Gamma \vdash x(y_1 : U_1, \ldots, y_r : U_r).P$, Γ_1 must contain $x : \texttt{Chan}[U_1, \ldots, U_r]$. Moreover, since $\Gamma \vdash \overline{x}\langle v_1, \ldots, v_n\rangle.Q$, we see that $x : \texttt{Chan}[U_1, \ldots, U_r]$ in the derivation. Hence, $n = r$ and $\Gamma \vdash v_i : U_i$ for $1 \leq i \leq n$. □

The simple type system for π-calculus provides a way of statically checking that communication channels are used consistently throughout a program. This eliminates a certain class of communication errors.

Several type systems have been developed for the π-calculus. Some aim to classify channels in more precise ways (input/output, linear, polymorphic, etc.). Others aim to eliminate more complex behavioural errors, such as deadlocks and livelocks. Lastly, behavioural type systems specify *order* and *sequence* of send and receive.

Advanced Types

We will now look at an advanced type system for π-calculus that aims to classify channels in a more precise way.

A fairly straightforward refinement of the simple type system introduces two new channel type constructors:

- $\texttt{In}[T_1, \ldots, T_n]$ is the type of channels that can be used to *input* messages of type $T_1, \ldots, T_n$; and
- $\texttt{Out}[T_1, \ldots, T_n]$ is the type of channels that can be used to *output* messages of type $T_1, \ldots, T_n$.

Now, types are specified by the following grammar:

$$\begin{aligned} T ::= & \ \texttt{Chan}[T_1, \ldots, T_n] \\ & \mid \texttt{In}[T_1, \ldots, T_n] \\ & \mid \texttt{Out}[T_1, \ldots, T_n] \\ & \mid \ldots \end{aligned}$$

Input/output channel types introduce a notion of *subtyping*. A subtyping relation connects the new constructors and the original constructor:

$$\begin{aligned} \texttt{Chan}[T_1, \ldots, T_n] &<: \texttt{In}[T_1, \ldots, T_n] \\ \texttt{Chan}[T_1, \ldots, T_n] &<: \texttt{Out}[T_1, \ldots, T_n] \end{aligned}$$

The following defines the subtyping rules in general- we have reflexivity and transitivity, which make it a partial order:

$$\frac{}{T <: T}\text{SRefl} \qquad \frac{R <: S \qquad S <: T}{R <: T}\text{STrans}$$

We also have the rules we saw above:

$$\frac{}{\texttt{Chan}[T_1, \ldots, T_n] <: \texttt{In}[T_1, \ldots, T_n]}\text{SChanIn}$$

$$\frac{}{\texttt{Chan}[T_1, \ldots, T_n] <: \texttt{Out}[T_1, \ldots, T_n]}\text{SChanOut}$$

We have covariance in subtyping with respect to input:

$$\frac{S_1 <: T_1 \qquad \ldots \qquad S_n <: T_n}{\texttt{In}[S_1, \ldots, S_n] <: \texttt{In}[T_1, \ldots, T_n]}\text{SInIn}$$

In terms of programming, consider the types *num* and its subtype *num*. Then, a channel accepting num can accept a value of type *int*. We have contravariance in subtyping with respect to output:

$$\frac{T_1 <: S_1 \qquad \dots \qquad T_n <: S_n}{\texttt{Out}[S_1, \dots, S_n] <: \texttt{Out}[T_1, \dots, T_n]}\text{SOutSOut}$$

In terms of programming, a channel that outputs an *int* can be thought of as a channel that outputs *num*. For subtyping with a general channel, we require both covariance and contravariance:

$$\frac{S_1 <: T_1 \qquad \dots \qquad S_n <: T_n \qquad T_1 <: S_1 \qquad \dots \qquad T_n <: S_n}{\texttt{Chan}[S_1, \dots, S_n] <: \texttt{In}[T_1, \dots, T_n]}\text{SChanChan}$$

To make use of subtyping, the typing judgment can be extended with the subsumption rule:

$$\frac{\Gamma \vdash x : S \qquad S <: T}{\Gamma \vdash T}\text{TSub}$$

An output must send a message of the correct structure on a channel having an output channel type. The inference rule is now:

$$\frac{\Gamma \vdash x : \texttt{Out}[T_1, \dots, T_n] \quad \Gamma \vdash e_i : T_i \ \forall 1 \leq i \leq n \quad \Gamma \vdash P}{\Gamma \vdash \overline{x}\langle e_1, \dots, e_n \rangle.P}\text{TOut}$$

The change is highlighted in red- we have `Out` instead of `Chan`.

In an input, the message must have the correct structure and its components must be used correctly.

$$\frac{\Gamma \vdash x : \texttt{In}[T_1, \dots, T_n] \qquad \Gamma, y_1 : T_1, \dots, y_n : T_n \vdash P}{\Gamma \vdash x(y_1 : T_1, \dots, y_n : T_n).P}\text{TIn}$$

A channel can only be created of type `Chan` in TNew; it cannot be the other way round.

In addition to capabilities of input/output, channel types can be refined by introducing *multiplicities* for linear and shared channels. *Linearity* restricts channel usage exactly once, in accordance with its capability:

- linear input type $\ell_i[\cdot]$ can only be used once in input;
- linear output type $\ell_o[\cdot]$ can only be used once in out; and
- linear channel type $\ell_{Chan}[\cdot]$ can only be used once in input and output.

Linear types allow for resource-aware programming and have inspired ownership types and unique types in PLs. The updated types are the following:

$$\begin{aligned} T ::= \ & \texttt{Chan}[T_1, \dots, T_n] \\ & \mid \ell_i[T_1, \dots, T_n] \\ & \mid \ell_o[T_1, \dots, T_n] \\ & \mid \ell_{Chan}[T_1, \dots, T_n] \\ & \mid \dots \end{aligned}$$

In order to respect linearity, a linear type system must keep track of the resources used, without duplicating or discarding any. We need a notion of

type combination on types, contexts and ultimately on typing rules. We will use the symbol $\uplus$ for the type combination operator, or simply $+$.

The combination '$\uplus$' of types is a symmetric operation and is defined by the following rules:

$$\begin{aligned}
\ell_i[T_1,\ldots,T_n] \uplus \ell_o[T_1,\ldots,T_n] &\triangleq \ell_{Chan}[T_1,\ldots,T_n] \\
\texttt{Chan}[T_1,\ldots,T_n] \uplus \texttt{Chan}[T_1,\ldots,T_n] &\triangleq \texttt{Chan}[T_1,\ldots,T_n] \\
\mathit{bool} \uplus \mathit{bool} &\triangleq \mathit{bool} \\
T \uplus T' &\triangleq \texttt{undef} \qquad \text{otherwise.}
\end{aligned}$$

Now, we extend '$\uplus$' on typing contexts based on these rules:

$$(\Gamma_1 \uplus \Gamma_2)(x) \triangleq \begin{cases} \Gamma_1(x) \uplus \Gamma_2(x) & \text{if both } \Gamma_1(x) \text{ and } \Gamma_2(x) \text{ are defined} \\ \Gamma_1(x) & \text{if } \Gamma_1(x) \text{ is defined but } \Gamma_2(x) \text{ is not defined} \\ \Gamma_2(x) & \text{if } \Gamma_2(x) \text{ is defined but } \Gamma_1(x) \text{ is not defined} \\ \texttt{undef} & \text{otherwise.} \end{cases}$$

We also add typing rules for linear output and input.

$$\frac{\Gamma_0 \vdash x\colon \ell_o[T_1,\ldots,T_n] \qquad \Gamma_1 \vdash e_1\colon T_1 \quad \ldots \quad \Gamma_n\colon e_n\colon T_n \qquad \Gamma_m \vdash P}{\Gamma_0 \uplus \Gamma_i \uplus \Gamma_m \vdash \overline{x}\langle e_1,\ldots,e_n\rangle.P}\text{TOut}$$

$$\frac{\Gamma_0 \vdash x\colon \ell_i[T_1,\ldots,T_n] \qquad \Gamma_1, y_1\colon T_1,\ldots,y_n\colon T_n \vdash P}{\Gamma_0 \uplus \Gamma_1 \vdash x(y_1\colon T_1,\ldots,y_n\colon T_n).P}\text{TIn}$$

Also, when a new channel is created, this must be a linear channel.

$$\frac{\Gamma, x\colon \ell_{Chan}[T_1,\ldots,T_n] \vdash P}{\Gamma \vdash (\nu\ x)\colon \ell_{Chan}[T_1,\ldots,T_n]P}\text{TNew}$$

Linear type systems guarantee that resources will be used exactly once. However, a process can be built in such a way that resources are not used at all! For instance, consider the following linear process:

$$a\colon \ell_i[\mathit{bool}] \vdash (\nu\ b\colon \texttt{Chan}[\mathit{bool}])\overline{b}\langle \mathit{false}\rangle.a(x).0$$

Although this is a well-typed process under the linear type system, it is deadlocked. Further extension to types and type systems for π-calculus deal with this, along with lock freedom and termination.

2.5 Session Types

In complex distributed systems, communicating participants agree on a protocol to follow, specifying *type* and *direction* of data exchanged. *Session types* are a type formalism used to model structured communication-based programming. They guarantee *privacy*, *communication safety* and *session fidelity*. It was initially designed for π-calculus, but has now been designed for functional languages, object-oriented languages, and more!

Session types were introduced in the early 90s. Since their appearance, they have developed into a significant theme in programming languages. This is because computing has now moved from the era of data processing to the era of communication. *Data types* codify the structure of *data* and make it available to programming tools. Similarly, *session types* codify the structure of *communication* and make it available to programming tools.

We will now have a look at an example of session types using a server and a client. The session type for the server is given as follows:

$$\begin{aligned} S \triangleq \&\{&add\colon ?int.?int.!int.S \\ &neg\colon ?int.!int.S \\ &quit\colon end\} \end{aligned}$$

The symbol & denotes that the server S is offering 3 services- *add*, *neg* and *quit*. This is called *external choice* (i.e. the client chooses). The *quit* service would end in termination of the protocol. On the other hand, the *add* service takes in 2 integers (as denoted by $?int$) and then returns an integer (as denoted by $!int$). Similarly, *neg* takes an integer and returns an integer.

From the client endpoint, the session type is the following:

$$\begin{aligned} C \triangleq \oplus\{&add\colon !int.!int.?int.C \\ &neg\colon !int.?int.C \\ &quit\colon end\} \end{aligned}$$

The symbol $\oplus$ denotes that the client must select one of the 3 services. This is called *internal choice*. Note that a send in the server is a receive in the client and vice versa. This is called *duality*, and is denoted $S = \overline{C}$.

We will now instantiate a server *srv*. It is parametrised on the channel endpoint:

$$\begin{aligned} srv(x\colon S) = x \rhd \{&add\colon x(a\colon int).x(b\colon int).\overline{x}\langle a+b\rangle.srv(x) \\ &neg\colon x(a\colon int).\overline{x}\langle -a\rangle.srv(x) \\ &quit\colon 0\} \end{aligned}$$

Next, we instantiate a client *clt*. It is of type C:

$$clt(x\colon C) = x \lhd neg.\overline{x}\langle 2\rangle.x(a\colon int).x \lhd quit.P(a)$$

Note that since $\oplus$ denotes an internal choice, the client *clt* has already chosen a sequence- it first chooses the *neg* service and then *quit*.

We can put srv and clt in parallel for communication to occur, using the reduction rules:

$$
\begin{gathered}
(\nu\ c\colon S)(srv(c^+) \mid clt(c^-)) \\
\downarrow \\
(\nu\ c\colon ?int.!int.S)(c^+(a\colon int).\overline{c^+}\langle -a\rangle.srv(c^+) \mid \overline{c^-}\langle 2\rangle.c^-(a\colon int).c^- \lhd quit.P(a)) \\
\downarrow \\
(\nu\ c\colon !int.S)(\overline{c^+}\langle -2\rangle.srv(c^+) \mid c^-(a\colon int).c^- \lhd quit.P(a)) \\
\downarrow \\
(\nu\ c\colon S)(srv(c^+) \mid c^- \lhd quit.P(-2)) \\
\downarrow \\
(\nu\ c\colon end)(0 \mid P(-2)) \\
\equiv \\
P(-2)
\end{gathered}
$$

The communication occurs at channel c. The server and the client are at the opposite ends of the channel. To distinguish this, we make use of polarities- c^+ for one and c^- for the other. We keep track of the type using the server; the client type could have also been used here. Note that the type of the channel c is given as the type of the server, and changes during the reduction process.

As we can see, the communication occurs in a private channel c. To accommodate this, the server listens on a standard channel a of type $\texttt{Chan}[S]$ and receives a session channel for srv to use. That is,

$$server(a) = a(x\colon S).srv(x)$$

The global declaration $a\colon \texttt{Chan}[S]$ advertises the server and its protocol. The client creates a session channel and sends it to the server, i.e.

$$client(a) = (\nu\ c\colon S)(\overline{a}\langle c^+\rangle.clt(c^-))$$

After one step reduction of $server(a) \mid client(a)$, execution proceeds as above.

The main features of session types are the following:

- *Duality*- the relationship between types of opposite endpoints of a session channel;
- *Linearity*- each channel endpoint occurs exactly once in a collection of parallel processes;
- The *structure* of session types matches the structure of communication;
- Session types *change* as communication occurs;
- *Connection* is established among participants before communication begins.

Session types guarantee the following:

- *communication safety*- the exchanged data has the expected type;

- *session fidelity*- the session channel has the expected structure; and
- *privacy*- the session channel is owned only by the communicating parties.

This allows us to conclude that, at runtime, communication follows the protocol.

We will now define session types using a grammar:

$S ::= end$	termination
$!T.S$	send
$?T.S$	receive
$\oplus \{I_i \colon S_i\}_{i \in I}$	select
$\& \{I_i \colon S_i\}_{i \in I}$	branch

The value T refers to a type. Now, session types are also types in π-calculus. The syntax of the processes has also changed, to the following grammar:

$P, Q ::= 0$	inaction
$P \mid Q$	composition
$(\nu\ x)P$	restriction
$\overline{x^p}\langle v^q \rangle.P$	output
$x^p(y).P$	input
$x^p \lhd l_j.P$	selection
$x^p \rhd \{I_i \colon P_i\}_{i \in I}$	branching

The value v refers to values, which includes channel names and other values (e.g. booleans). The values p and q are optional polarities for channels (i.e. $+$ or $-$).

The typing rules are given as follows:

$$\frac{\Gamma_1 \vdash P \quad \Gamma_2 \vdash Q}{\Gamma_1 + \Gamma_2 \vdash P \mid Q}\text{TPar}$$

$$\frac{\Gamma, x^p \colon S, x^{\overline{p}} \colon \overline{S} \vdash P}{\Gamma \vdash (\nu\ x)P}\text{TRes}$$

$$\frac{\Gamma, x^p \colon S, y \colon T \vdash P}{\Gamma, x^p \colon ?T.S \vdash x^p(y).P}\text{TIn}$$

$$\frac{\Gamma_1, x^p \colon S \vdash P \quad \Gamma_2 \vdash v^q \colon T}{(\Gamma_1, x^p \colon !T.S) + \Gamma_2 \vdash \overline{x^p}\langle v^q \rangle.P}\text{TOut}$$

$$\frac{\Gamma, x^p \colon S_i \vdash P_i}{\Gamma, x^p \colon \& \{I_i \colon S_i\}_{i \in I} \vdash x^p \rhd \{I_i \colon P_i\}_{i \in I}}\text{TBrch}$$

$$\frac{j \in I \quad \Gamma, x^p \colon S_j \vdash P}{\Gamma, x^p \colon \oplus \{I_i \colon S_i\}_{i \in I} \vdash x^p \lhd I_j.P}\text{TSel}$$

For a polarity p, its dual is denoted by $\overline{p}$. Note that the types change as we saw above.

We define the combination + operation on typing judgments as follows:

$$\begin{aligned}
\Gamma + x^+ \colon S &= \Gamma, x^+ \colon S && \text{if } x, x^+ \notin \mathrm{dom}(\Gamma) \\
\Gamma + x^- \colon S &= \Gamma, x^- \colon S && \text{if } x, x^- \notin \mathrm{dom}(\Gamma) \\
\Gamma + x \colon S &= \Gamma, x \colon S && \text{if } x, x^+, x^- \notin \mathrm{dom}(\Gamma) \\
(\Gamma, x \colon T) + x \colon T &= \Gamma, x \colon T && \text{if } T \text{ is not a session type}
\end{aligned}$$

We will now have a look at some terms and see whether they are well-typed:

- The process
$$(\nu\ x)(x^+(t \colon \mathit{bool}).0 \mid \overline{x^-}\langle \mathit{true} \rangle.0)$$
is well-typed since the types of x^+ and x^- are dual.

- The process
$$(\nu\ x)(\overline{x^+}\langle t \rangle.0 \mid \overline{x^-}\langle \mathit{true} \rangle.0)$$
is not well-typed since both x^+ and x^- send data, and hence are not dual of each other.

- The process
$$(\nu\ x)(\overline{x^-}\langle \mathit{false} \rangle.0 \mid x^+(t \colon \mathit{bool}).0 \mid x^+(w \colon \mathit{bool}).0)$$
is not well-typed since it violates linearity- there can only be one occurrence of x^+ and x^-.

- The process
$$(\nu x)(\nu y)(x^+(z \colon \mathit{int}).\overline{y^-}\langle 42 \rangle.0 \mid \overline{x^-}\langle 11 \rangle.y^+(w \colon \mathit{int}).0)$$
is well-typed since the types of x^-, x^+ and y^-, y^+ are dual.

- The process
$$(\nu x)(\nu y)(\overline{y^-}\langle 42 \rangle.x^+(z \colon \mathit{int}).0 \mid \overline{x^-}\langle 11 \rangle.y^+(w \colon \mathit{int}).0)$$
is well-typed since the types of x^-, x^+ and y^-, y^+ are dual, even though the process is actually deadlocked.

- The process
$$(\nu\ x)(x^- \lhd k.0 \mid x^+ \rhd \{l_i \colon P_i\}_{i \in I}.0)$$
is well-typed if and only if $k = l_i$ for some $i \in I$.

Standard types and session types

We will now aim to connect standard types (e.g. the simple type system π-calculus) on π-calculus and session types. We saw the following types:

- $\texttt{Chan}[T_1, \ldots, T_n]$ for input/output for data of type $T_1, \ldots, T_n$;

- $\texttt{In}[T_1, \ldots, T_n]$ and $\texttt{Out}[T_1, \ldots, T_n]$ for only input or only output for data of type $T_1, \ldots, T_n$;

- $\ell_i[T_1, \ldots, T_n]$ and $[\ell_o[T_1, \ldots, T_n]]$ for only linear input or linear output (i.e. transmit data exactly once); and
- *variant types* $\langle l_1 : T_n, \ldots, l_n : T_n \rangle$ for a disjoint union of types that are labelled.

On the other hand, session types have:

1. structure;
2. duality;
3. restriction; and
4. branch/select.

This can also be seen in standard π-calculus. In particular,

1. *linearity* forces a channel to be used exactly once;
2. we have the *capability* to split a channel into input/output channels;
3. we have *restriction* that allows for creation of private channels; and
4. *variant type* allows for choice.

Because of the similarities, we might ask about the difference in expressivity of standard π-calculus and session types. It turns out that they have equal expressivity.

We can make use of standard π-calculus within session types. Hence, to prove that they have equal expressivity, we just need to show that we can convert a process that makes use of session types to a process without the session types, but the same functionality.

The encoding is based on:

- *linearity* of channel types;
- *input* and *output* channel capabilities;
- the *continuation-passing* principle; and
- *variant types*.

The way the encoding works, informally, is as follows:

- session types are encoded as linear channels;
- ? and ! are encoded as ℓ_i and ℓ_o respectively;
- $\&\{I_i \mid S_i\}_{i \in I}$ and $\oplus\{I_i \mid S_i\}_{i \in I}$ are encoded using variant types;
- continuation of a session type becomes the carried type; and
- dual operations in continuation become equal when carried.

We will now consider the encoding process with an example. So, assume that S is a process with session type:

$$S = ?int.?int.!bool.end$$

Then, it is encoded as the following type in standard π-calculus:

$$\|S\| = \ell_i[int, \ell_i[int, \ell_o[bool, \varnothing]]]$$

As we can see, an input type $?int$ becomes $\ell_i[int, \cdot]$, where $\cdot$ contains the continuation process. Similarly, an output type $!int$ becomes $\ell_o[int, \cdot]$. Finally, end is denoted by the empty symbol $\varnothing$.

Now, consider the dual type of S:

$$\overline{S} = !int.!int.?bool.end$$

Then, it is encoded as the following type:

$$\|\overline{S}\| = \ell_o[int, \ell_i[int, \ell_o[bool, \varnothing]]]$$

Note that although the first $?int$ has been converted into $\ell_o[int, \cdot]$, this is not the case for the second one. This is because both S and its dual $\overline{S}$ have to agree on the type of input/output for them to be able to communicate. Hence, the second parameter of the dual $\overline{S}$ must be the same as the second parameter of S. We can also think of this as encoding its dual after a receive type. We can think of this in general- in session types, dual types have opposite types throughout, whereas in normal π-calculus, only the topmost type is of opposite type.

Formally, the algorithm for encoding is given as follows:

$$\begin{aligned}
\|end\| &\triangleq \varnothing[] \\
\|?T.S\| &\triangleq \ell_i[\|T\|, \|S\|] \\
\|!T.S\| &\triangleq \ell_o[\|T\|, \|\overline{S}\|] \\
\|\&\{I_i \mid S_i\}_{i \in I}\| &\triangleq \ell_i[\langle l_i \mid \|S_i\| \rangle_{i \in I}] \\
\|\oplus\{I_i \mid S_i\}_{i \in I}\| &\triangleq \ell_o[\langle l_i \mid \|\overline{S_i}\| \rangle_{i \in I}]
\end{aligned}$$

The encoding obeys many properties, such as:

- it preserves *typeability* of programs;
- it preserves *evaluation* of programs; and
- encoding of dual session types gives *dual linear π-types*.

The encoding has quite powerful applications, including:

- large usability of standard typed π-calculus theory;
- derivation of processes for session π-calculus from the standard π-calculus (e.g. type safety);
- elimination of redundancy in the syntax of types and terms, and in theory;
- the robustness of the encoding (subtyping, polymorphism and higher-order);
- expressivity result for session types;
- implementation of session types in mainstream programming languages.

Curry-Howard Isomorphism

We will now look at the Curry-Howard isomorphism for session types. Since the beginning of linear logic, there were suggestions that it should be relevant to concurrency. The logically approach for session types has been studied since the 1990s. It has been extended to dependent types, failures, sharing and races.

The correspondence between session types as linear logic is the following:

- propositions are equivalent to session types;
- proofs are equivalent to π-procesess; and
- proof normalisation and cut elimination are equivalent to communication.

Moreover,

- the proposition $A \parr B$ is interpreted as 'input A then behave like B', i.e. $?A.B$;
- the proposition $A \otimes B$ is interpreted as 'output A then behave like B', i.e. $!A.B$;
- $\&$ and $\oplus$ are interpreted to branch and select.

The correspondence has lead to a re-examination of all aspects of session types from a logical viewpoint.

These are the connections between session types and classical linear logic.

$$\frac{P \vdash_\Delta y\colon A, x\colon B}{x(y).P \vdash_\Delta, x\colon A \parr B}\mathrm{T}\parr$$

$$\frac{P \vdash_\Delta, y\colon A \quad Q \vdash_{\Delta'}, x\colon B}{\overline{x}\langle y\rangle.(P \mid Q) \vdash_\Delta, \Delta', x\colon A \otimes B}\mathrm{T}\otimes$$

$$\frac{P \vdash_\Delta, x\colon \overline{A} \quad Q \vdash_{\Delta'}, x\colon A}{(\nu\ x)(P \mid Q) \vdash_\Delta, \Delta'}\mathrm{TCut}$$

$$\frac{P_i \vdash_\Delta, x\colon A_i}{x \rhd \{l_i\colon P_i\}_{i\in I} \vdash_\Delta, x\colon \&\{I_i\colon A_i\}_{i\in I}}\mathrm{T}\&$$

$$\frac{P \vdash_\Delta, x\colon A_j \quad j \in I}{x \lhd l_j.P \vdash \Delta, x\colon \oplus\{I_i \mid A_i\}_{i\in I}}\mathrm{T}\oplus$$

The red values are propositions, while the black ones are classical linear logic. Note that the judgment is reversed.

The session type system based on classical linear logic propositions guarantees:

- type preservation (or subject reduction)- well-typed processes reduce to well-typed processes;
- deadlock-freedom (by design)- if a process P is well-typed and it is a *cut*, then there is some Q such that P reduces to Q and Q is not a *cut*.

Curry-Howard correspondences between session-typed languages and linear logic are a vivid research area. Recent work includes extending these correspondences to include more expressive processes-containing 'good' cycles, sharing, recursion, dependent types and so on.