CO661 - Theory and Practice of Concurrency

Notes on the Calculus of Communicating Systems (CCS)

Last updated on November 25, 2025

In this module, we examine both the theory and practice of concurrency. The theoretical side brings rigour to our reasoning and understanding of key concepts in concurrent programming. In this module, we utilise CCS, the Calculus of Communicating Systems,[1] as a model for understanding concurrency. CCS is essentially a small programming language (a "*calculus*") which is carefully and formally defined. This allows us to be precise about the meaning of concurrent programs. CCS is often described as a *process calculus* or *process algebra* since it is a calculus where "processes" are the central unit of organisation: its programs are processes, which are made up of further processes.

These notes are open source. [2] If you spot any mistakes or errors or things that need clarifying, please fork and send a pull request.

Contents

1 Syntax

CCS processes comprise *actions*, which we use to abstract the notion of a *command* in a program or a particular function call or a message. For example, the following CCS program describes a process that performs an (abstract) action a followed by an (abstract) action b and then does nothing:

$$a.b.\mathbf{0}$$

Actions are sequenced via the operator "." which is called the *prefixing operator* since it is used to prefix a process with an action. The process $\mathbf{0}$ is the inactive process.

The full syntax of CCS is defined inductively by the following BNF grammar:

$$\begin{array}{rll} P, Q ::= & \alpha.P & \textit{prefixing} \\ \mid & P + Q & \textit{non-deterministic choice} \\ \mid & (P \mid Q) & \textit{parallel composition} \\ \mid & \nu a.P & \textit{name restriction} \\ \mid & \mathbf{0} & \textit{inactive process} \\ \mid & P[b/a] & \textit{relabelling} \\ \mid & A & \textit{process identifiers} \end{array}$$

[1] Robin Milner: *A Calculus of Communicating Systems*, Springer Verlag, ISBN 0-387-10235-3. 1980.

[2] https://github.com/dorchard/co661-notes

where $\alpha ::= a \mid b \mid \ldots \overline{a} \mid \overline{b} \mid \ldots$ ranges over *names* (of actions) where names with an overline are called *co-names* (or *dual names*). We use P and Q to range over processes, i.e., when we want to refer to some process in an abstract sense. We use A to range over identifiers for processes (i.e., concrete names given to processes, see below on "Process definitions").

We often use actions a to model receiving a message "a" (also called an *input action*) whilst its dual action $\overline{a}$ corresponds to sending the message "a" (an *output action*).

The following gives a few examples to give some intuition, but in the next section we will look at the concrete semantics of processes which will make precise what each part of the syntax means.

- $a.b.P$ – does action a then action b then continues to behave as the process P;
- $a.0 + b.0$ – chooses non-deterministically between doing action a or doing action b.
- $(a.b.0 \mid c.d.0)$ – in parallel, one process does action a then b then stops (inactive process), whilst the other does c then d then stops. What is observed is an *interleaving* of these actions, e.g., one possibility is that whole process does a then c then b then d. Another possibility is c then d then a then b. The point is that we may get any interleaving of the actions of the subprocesses.
- $\nu a.(a.P \mid \overline{a}.Q)$ – this is a process where no a action can be observed externally, because of the name restriction νa. The inner process $(a.P \mid \overline{a}.Q)$ can do a *handshake* between the action a and its dual $\overline{a}$ which can happen together leaving a process $P \mid Q$. This notion of handshaking comes from the semantics (later).

Concurrency Workbench The Concurrency Workbench tool (CAAL), which we use in this course, has a slightly different concrete syntax, with the above constructs represented as follows (in order):

$$P ::= \texttt{a}.P \mid \texttt{'a}.P \mid P\texttt{+}Q \mid (P\texttt{|}Q) \mid P\texttt{\{a1,...,an\}} \mid \texttt{0} \mid P\texttt{[b/a]} \mid A$$

Process definitions We will define processes by recursive definitions. For example, the following defines a process named A:

$$A \stackrel{def}{=} a.b.A$$

The process is recursively defined, and performs actions a then b continuously.

Processes can be mutually recursive. For example, we could define a process that performs a then b actions continuously via two definitions:

$$A \stackrel{def}{=} a.B \qquad\qquad B \stackrel{def}{=} b.A$$

Here A performs a then b actions continuously whilst B performs b then a actions continuously (they have different starting points).

2 Semantics

We describe the semantics (the meaning) of CCS processes via a *labelled-transitions system* which effectively describes small "steps" of evaluation for CCS processes along with actions which are made observable by this evaluation. We define these small steps (called *reductions* or *transitions*) by a relation between processes,[3] annotated with a label, of the form:

$$P \xrightarrow{l} Q$$

which describes that a process P can reduce (*evaluate*) to a process Q by doing some action described by the label l. This is a *labelled transition*. Labels are as defined as:

$$l ::= \alpha \mid \tau$$

[3] Viewed set theoretically, $\rightarrow$ is a ternary relation, i.e. $(\rightarrow) \subseteq P \times l \times P$.

i.e., they are either names of actions or they are a special label τ called a *silent action*.

Since the syntax of processes is itself inductively defined (i.e. processes can contain processes) so the definition of $\rightarrow$ is inductively defined. We leverage the *inference rule* style, with inductive rules having one or more premises and axiom rules with no premises. Each piece of syntax has at least one corresponding rule for $\rightarrow$, giving the semantics of each syntactic construct.

Actions The first simple reduction rule is for action prefixes:

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

We see that a process with an action a prefixing another process P can reduce by emitting that action and then becoming the remaining processes P. e.g.

$$\textit{action}\ \frac{}{a.b.\mathbf{0} \xrightarrow{a} b.\mathbf{0}} \qquad \text{(example)}$$

Definition 1 (Trace)**.** Labelled transitions can be composed together to form a *trace*, which comprises a sequence of reductions steps, e.g.,

$$a.b.\mathbf{0} \xrightarrow{a} b.\mathbf{0} \xrightarrow{b} \mathbf{0}$$

Each step here is a reduction that needs a *derivation*. We will see below that we can derive reductions for more complex processes by "plugging together" inference rules in a derivation tree.

Parallel composition A parallel composition of processes $P \mid Q$ has two possible reductions given by the following inductive rules:

$$\textit{par1}\ \frac{P \xrightarrow{l} P'}{P \mid Q \xrightarrow{l} P' \mid Q} \qquad \textit{par2}\ \frac{Q \xrightarrow{l} Q'}{P \mid Q \xrightarrow{l} P \mid Q'}$$

The *par1* rule says that if we have a process P that can reduce to P' with label l then we can place P in parallel with Q (i.e., $P \mid Q$) and reduce to $P' \mid Q$ with label l. Essentially we are allowing a process within a parallel composition to make some progress. The *par2* provides the symmetric case.

Example 1. We can compose the *action* and *par1* rules to get the following derivation of a single reduction step for the process $a.P \mid b.Q$:

$$\textit{par1}\ \frac{\textit{action}\ \dfrac{}{a.P \xrightarrow{a} P}}{(a.P \mid b.Q) \xrightarrow{a} (P \mid b.Q)} \qquad \text{(example)}$$

Thus, we have reduced on the left-hand side of the parallel composition, emitting an a action to get the resulting process $P \mid b.Q$. This process could then reduce in a separate reduction by applying *par2* :

$$\textit{par2}\ \frac{\textit{action}\ \dfrac{}{b.Q \xrightarrow{b} Q}}{(P \mid b.Q) \xrightarrow{b} (P \mid Q)} \qquad \text{(example)}$$

These two reduction steps, once derived, can be put together to form the following trace:

$$(a.P \mid b.Q) \xrightarrow{a} (P \mid b.Q) \xrightarrow{b} (P \mid Q)$$

This is not the only possible reduction sequence for the process $(a.P \mid b.Q)$. Another possibility is that we could reduce on the right first, then on the left, giving the trace:

$$(a.P \mid b.Q) \xrightarrow{b} (a.P \mid Q) \xrightarrow{a} (P \mid Q)$$

We can put these two traces together into a *transition graph* that shows the possible traces:

$$\begin{array}{ccc} a.P \mid b.Q & \xrightarrow{\ a\ } & P \mid b.Q \\ {\scriptstyle b}\downarrow & & \downarrow{\scriptstyle b} \\ a.P \mid Q & \xrightarrow[\ a\ ]{} & P \mid Q \end{array}$$

Note that this represents a *confluent* reduction, i.e., we take different paths in the reduction but end in the same place.

A further reduction is possible for a parallel composition of processes, but in a restricted setting where two parallel processes reduce by dual actions:

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

In this case we have the processes P and Q making a reduction step *at the same time* to become $P' \mid Q'$ if they reduce by dual actions a and $\overline{a}$. We can use this to model notions of synchronisation, co-ordination, or communication between processes. The action τ is described as a "silent" action because an outsider cannot observe which names were used in the handshake.

Example 2. Consider the following processes representing a vending machine V and a customer C:

$$V \stackrel{\text{def}}{=} \mathsf{coin}.\overline{\mathsf{tea}}.V$$
$$C \stackrel{\text{def}}{=} \overline{\mathsf{coin}}.\mathsf{tea}.C$$

Since V can reduce by emitting the action coin and C can reduce by emitting the action $\overline{\mathsf{coin}}$ we can get the reduction:

$$\textit{handshake}\ \frac{V \xrightarrow{\mathsf{coin}} \overline{\mathsf{tea}}.V \qquad C \xrightarrow{\overline{\mathsf{coin}}} \mathsf{tea}.C}{V \mid C \xrightarrow{\tau} (\overline{\mathsf{tea}}.V \mid \mathsf{tea}.C)} \qquad \text{(example)}$$

Note that the resulting process can further do a handshake because of the dual tea actions, bringing us back to the process $V \mid C$.

Choice A non-deterministic choice between two processes $P + Q$ reduces by reducing just one side, resulting in a process that corresponds to just that one side of the choice, given by the rules:

$$\textit{choice1}\ \frac{P_1 \xrightarrow{l} Q}{P_1 + P_2 \xrightarrow{l} Q} \qquad \textit{choice2}\ \frac{P_2 \xrightarrow{l} Q}{P_1 + P_2 \xrightarrow{l} Q}$$

Example 3. For a process $b.P + c.Q$, the top-level syntactic construct is non-deterministic choice, so a reduction must necessarily use *choice1* or *choice2* . For example:

$$\textit{choice2}\ \frac{\textit{action}\ \dfrac{}{c.Q \xrightarrow{c} Q}}{b.P + c.Q \xrightarrow{c} Q} \qquad \text{(example)}$$

Note that when we make a choice, we lose the other branch— above, $b.P$ disappears in the resulting process term. The lectures have more examples.

Restriction The notion of restricting a name for a process, i.e. $\nu a.P$ can be thought of a bit like binding a name a in the scope of P so that the name is "local" to P and cannot be observed outside of it. Its semantics is given by:

$$\textit{restriction}\ \frac{P \xrightarrow{l} P'}{\nu a.P \xrightarrow{l} \nu a.P'} l \neq a \wedge l \neq \overline{a}$$

Here we are saying that we can observe the action given by label l only if it is not the bound name a or its dual $\overline{a}$. We can use this to enforce handshaking of dual actions.

Example 4. Consider the processes $a.P$ and $\overline{a}.Q$. Their parallel composition has three possible traces:

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

By restricting the process on action a (i.e., the process term $\nu a.(a.P \mid \overline{a}.Q)$) we can make sure that the handshaking trace is the only one possible, preventing the other two traces, with the derivation:

$$\textit{restriction}\ \frac{\textit{handshake}\ \dfrac{\textit{action}\ \dfrac{}{a.P \xrightarrow{a} P} \quad \textit{action}\ \dfrac{}{\overline{a}.Q \xrightarrow{\overline{a}} Q}}{(a.P \mid \overline{a}.Q) \xrightarrow{\tau} P \mid Q}}{\nu a.(a.P \mid \overline{a}.Q) \xrightarrow{\tau} \nu a.(P \mid Q)} \quad \text{(example)}$$

The side condition for *restriction* is satisfied since $a \neq \tau$ and $\overline{a} \neq \tau$. No other derivation is possible. For example, the following is not a valid derivation because it would violate the side condition of *restriction* (that the label does not match the name being restricted over):

$$\text{✗}\quad \textit{restriction}\ \frac{\textit{par}\ \dfrac{\textit{action}\ \dfrac{}{a.P \xrightarrow{a} P}}{(a.P \mid \overline{a}.Q) \xrightarrow{a} (P \mid \overline{a}.Q)}}{\nu a.(a.P \mid \overline{a}.Q) \xrightarrow{a} \nu a.(P \mid \overline{a}.Q)} \quad \text{(example)}$$

We can encapsulate the above notion via a theorem:

Theorem 1. Restriction of a name forces handshaking on dual names. That is:

$$\forall P, P', Q, Q', a. \qquad P \xrightarrow{a} P' \wedge Q \xrightarrow{\overline{a}} Q' \ \wedge P \neq P' \wedge Q \neq Q'$$
$$\Leftrightarrow \quad \nu a.(P \mid Q) \xrightarrow{\tau} \nu a.(P' \mid Q')$$

That is, if two process can reduce by emitting dual names a and $\overline{a}$, then their parallel composition under a restriction of a can only reduce by a handshake, and vice versa.

Finally, a common syntactic shorthand for a several name restrictions is to give a comma-separated list of the restricted names, e.g. $\nu a.\nu b.P$ is written as $\nu a, b.P$.

Process identifiers A process identifier A can reduce given a definition for A which itself can reduce. This is given by the rule:

$$\textit{def}\ \frac{P \xrightarrow{l} Q}{A \xrightarrow{l} Q}\ (A \stackrel{\text{def}}{=} P)$$

Example 5. For example, if we have $A \stackrel{\text{def}}{=} a.b.A$ then we get the derivation:

$$\textit{def}\ \frac{\textit{action}\ \dfrac{}{a.b.A \xrightarrow{a} b.A}}{A \xrightarrow{a} b.A} \quad \text{(example)}$$

Relabelling Relabelling give us a way to rename an emitted action. That is $P[b/a]$ means that for process P, any emitted action a is renamed to the action b. The semantics is given by the reduction:

$$\textit{relabel}\ \frac{P \xrightarrow{l} Q}{P[b/a] \xrightarrow{l[b/a]} Q[b/a]}$$

where relabelling $l[b/a]$ means to relabel name a to b for the label l, defined:

$$l[b/a] = \begin{cases} b & \textit{when } l = a \\ \overline{b} & \textit{when } l = \overline{a} \\ l & \textit{otherwise} \end{cases}$$

Relabelling is useful when reusing definitions for different purposes (abstraction and reuse), but we do not make frequent use of it in the course.

2.1 Collected rules

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

$$\textit{par1}\ \frac{P \xrightarrow{l} P'}{P \mid Q \xrightarrow{l} P' \mid Q} \qquad \textit{par2}\ \frac{Q \xrightarrow{l} Q'}{P \mid Q \xrightarrow{l} P \mid Q'}$$

$$\textit{handshake}\ \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\overline{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \textit{restriction}\ \frac{P \xrightarrow{l} P'}{\nu a.P \xrightarrow{l} \nu a.P'} l \neq a \wedge l \neq \overline{a}$$

$$\textit{choice1}\ \frac{P_1 \xrightarrow{l} Q}{P_1 + P_2 \xrightarrow{l} Q} \qquad \textit{choice2}\ \frac{P_2 \xrightarrow{l} Q}{P_1 + P_2 \xrightarrow{l} Q}$$

$$\textit{def}\ \frac{P \xrightarrow{l} Q}{A \xrightarrow{l} Q}\ (A \overset{def}{=} P) \qquad \textit{relabel}\ \frac{P \xrightarrow{l} Q}{P[b/a] \xrightarrow{l[b/a]} Q[b/a]}$$

where relabelling $l[b/a]$ means to relabel name a to b for the label l, defined:

$$l[b/a] = \begin{cases} b & \textit{when } l = a \\ \overline{b} & \textit{when } l = \overline{a} \\ l & \textit{otherwise} \end{cases}$$

Example 6. Here is an extended example of a derivation of one possible reduction for the process defined $A \overset{\text{def}}{=} a.\mathbf{0} \mid (b.P + c.Q)$:

$$\textit{def}\ \frac{\textit{par2}\ \dfrac{\textit{choice2}\ \dfrac{\textit{action}\ \dfrac{}{c.Q \xrightarrow{c} Q}}{b.P + c.Q \xrightarrow{c} Q}}{a.\mathbf{0} \mid (b.P + c.Q) \xrightarrow{c} a.\mathbf{0} \mid Q}}{A \xrightarrow{c} a.\mathbf{0} \mid Q} \qquad \text{(example)}$$

3 Modelling locks, semaphores, and race conditions

We can use CCS to model common concurrent programming constructs, helping us to understand their behaviour. The following accompanies the lectures for this concept.

The following is a simple CCS model for the vending machine of `TeaRoomInitial.java`:

$$
\begin{array}{rrcl}
(\textit{accounting}) & \mathit{Inc}(p).P & \stackrel{\text{def}}{=} & \mathsf{r}(p).\overline{\mathsf{w}}(p+1).P \\
& \mathit{Dec}(s).P & \stackrel{\text{def}}{=} & \mathsf{r}(s).\overline{\mathsf{w}}(s-1).P \\
& A.P & \stackrel{\text{def}}{=} & \mathit{Dec}(s).\mathit{Inc}(p).P \\
(\textit{vend}) & V & \stackrel{\text{def}}{=} & \mathsf{coin}.\overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \\
(\textit{machine}) & M & \stackrel{\text{def}}{=} & V \mid M
\end{array}
$$

The "accounting" agent A provides the behaviour of reading and then updating two variables: incrementing the profit p and decrementing the supply s. Note that we abuse notation slightly here and definition *parametric* processes $\mathit{Inc}(p)$ and $\mathit{Dec}(s)$ parameterised by the name of some variable we are changing. This is not real CCS syntax, but a bit of meta-level syntactic sugar to simplify the model.

The vending machine receives a coin, then requests the lock before trying to behave as an accounting agent, then unlocking and sending tea. Since a machine is a shared object for which the `vend` method can be called arbitrarily and concurrently, a machine M is an infinite parallel composition of V.

A customer and a lock (a shared object) are defined:

$$
\begin{array}{rrcl}
(\textit{lock}) & L & \stackrel{\text{def}}{=} & \mathsf{lock}.\mathsf{unlock}.L \\
(\textit{customer}) & C & \stackrel{\text{def}}{=} & \overline{\mathsf{coin}}.\mathsf{tea}.C
\end{array}
$$

The following CCS agent then models two consumers at the vending machine:

$$\nu\mathsf{coin}.\ \nu\mathsf{tea}.\ \nu\mathsf{lock}.\ \nu\mathsf{unlock}.\ (C \mid C \mid M \mid L)$$

We are *restricting* the names coin, tea, lock and unlock so that handshaking is guaranteed (see Theorem 1). The following gives a trace of the model, but we elide the restriction since it is not necessary to get the following trace.

A process P is highlighted as P when we are going to expand its definition in the next line, where P marks the expanded definition. Pairs of dual (complementary) actions are highlighted as a and $\overline{\mathsf{a}}$ when they are about to handshake via a $\xrightarrow{\tau}$ step.

$$
\begin{aligned}
C \mid C \mid M \mid L &\equiv \overline{\mathsf{coin}}.\mathsf{tea}.C \mid C \mid \mathsf{coin}.\overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid L \\
&\xrightarrow{\tau} \mathsf{tea}.C \mid C \mid \overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid L \\
&\equiv \mathsf{tea}.C \mid C \mid \overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{lock}.\mathsf{unlock}.L \\
&\xrightarrow{\tau} \mathsf{tea}.C \mid C \mid A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{unlock}.L \\
&\equiv \mathsf{tea}.C \mid \overline{\mathsf{coin}}.\mathsf{tea}.C \mid A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid \mathsf{coin}.\overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{unlock}.L \\
&\xrightarrow{\tau} \mathsf{tea}.C \mid \mathsf{tea}.C \mid \underbrace{A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}}}_{\text{vend 1}} \mid \underbrace{\overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}}}_{\text{vend 2}} \mid M \mid \mathsf{unlock}.L
\end{aligned}
$$

The second vending process is not able to acquire the lock since there is no dual lock action that is the head prefix of any other process. The "vend 2" process has to wait for the first one ("vend 1")

to unlock via $\overline{\mathsf{unlock}}$ which then puts the lock L back to its initial configuration. That is, after some steps dealing with accounting (which is now atomic) we get:

$$\begin{aligned} & \mathsf{tea}.C \mid \mathsf{tea}.C \mid \underbrace{\overline{\mathsf{tea}}}_{\text{vend 1}} \mid \underbrace{\overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}}}_{\text{vend 2}} \mid M \mid L \\ \equiv\ & \mathsf{tea}.C \mid \mathsf{tea}.C \mid \overline{\mathsf{tea}} \mid \overline{\mathsf{lock}}.A.\overline{\mathsf{unlock}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{lock}.\mathsf{unlock}.L \end{aligned}$$

We define the following CCS model of `TeaRoomSem.java` which reuses the accounting agent A:

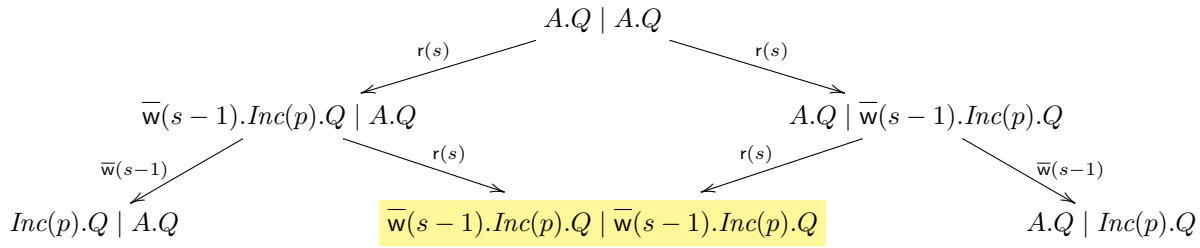
$$\begin{aligned} (\textit{1-semaphore}) \quad & S_1 \stackrel{\text{def}}{=} \mathsf{acq}.\mathsf{rel}.S_1 \\ (\textit{n-semaphore}) \quad & S_n \stackrel{\text{def}}{=} S_1 \mid S_{n-1} \\ (\textit{vend}) \quad & V \stackrel{\text{def}}{=} \mathsf{coin}.\overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \\ (\textit{machine}) \quad & M \stackrel{\text{def}}{=} V \mid M \\ (\textit{customer}) \quad & C \stackrel{\text{def}}{=} \overline{\mathsf{coin}}.\mathsf{tea}.C \end{aligned}$$

A semaphore with n permissions (capacity) is modelled by S_n which composes in parallel n copies of the binary semaphore agent S_1. We can see that a binary semaphore is basically the same as our previous model of a lock L. However, now n processes can be in their critical section.

The following trace models two consumers at a vending machine with a semaphore that has two permissions:

$$\begin{aligned} C \mid C \mid M \mid S_2 \ &\equiv\ \overline{\mathsf{coin}}.\mathsf{tea}.C \mid C \mid \mathsf{coin}.\overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid S_2 \\ &\xrightarrow{\tau}\ \mathsf{tea}.C \mid C \mid \overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid S_2 \\ &\equiv\ \mathsf{tea}.C \mid C \mid \overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{acq}.\mathsf{rel}.S_1 \mid S_1 \\ &\xrightarrow{\tau}\ \mathsf{tea}.C \mid C \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{rel}.S_1 \mid S_1 \\ &\equiv\ \mathsf{tea}.C \mid \overline{\mathsf{coin}}.\mathsf{tea}.C \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid \mathsf{coin}.\overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{rel}.S_1 \mid S_1 \\ &\xrightarrow{\tau}\ \mathsf{tea}.C \mid \mathsf{tea}.C \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid \overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{rel}.S_1 \mid S_1 \\ &\equiv\ \mathsf{tea}.C \mid \mathsf{tea}.C \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid \overline{\mathsf{acq}}.A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{rel}.S_1 \mid \mathsf{acq}.\mathsf{rel}.S_1 \\ &\xrightarrow{\tau}\ \mathsf{tea}.C \mid \mathsf{tea}.C \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid A.\overline{\mathsf{rel}}.\overline{\mathsf{tea}} \mid M \mid \mathsf{rel}.S_1 \mid \mathsf{rel}.S_1 \end{aligned}$$

We now have a race condition in the reduction of the yellow highlighted sub-process, which we show separate from the rest of the context (which is independent) setting $Q = \overline{\mathsf{rel}}.\overline{\mathsf{tea}}$:



Note that the middle term is achieved when the left and right hand processes interleave their reads, leading to a state where both processes have read the same value of s (the supply) and will write the same value of $s-1$, shadowing the fact that **two** processes are actually reducing the supply.

Furthermore, this can hide the situation where the supply is only one ($s = 1$) and therefore only one process is allowed to consume from the supply. But if both processes interleave their reads as in the above, they will both see a view of the world where $s = 1$ and erroneously assume there is enough in the supply for them both to proceed.

Thus we see that semaphores alone cannot prevent *race conditions*. If semaphore-guarded code contains shared state, we must also enforce mutual exclusion on that shared state via an additional binary semaphore or mutual-exclusion lock.

4 Equivalence of Processes

In Milner's CCS calculus, the notion of equivalence captures when two processes should be regarded as "the same" despite having different syntactic presentations. Several approaches exist, however, it is important that whatever notion of equivalence we choose this has to be a *congruence* with respect to the constructors of the language.

To give an idea of what this means imagine that you have a program written in some imperative language

$$f(x) \triangleq m := x;\ \texttt{return}\ (x + 1);$$

This program is not (and should not) be equivalent to the program $inc(x) =$ `return` $(x + 1)$;. The reason is the following. Say that you have a program $\mathbb{C}(g)$ which relies on a function $g : \mathbb{N} \to \mathbb{N}$ to calculate the increment of a natural number. We notice that there exists a program $\mathbb{C}$ such that if we give it the increment function it will work correctly and if we give it the function f it will crash.

This program can be defined as

$$\begin{aligned}\mathbb{C}[g] \triangleq\ & m := 0;\\ & g(1);\\ & \texttt{if}\ m! = 0\ \texttt{then}\ \mathit{crash}\ \texttt{else return}\ \mathit{ok}\end{aligned}$$

This is a toy example to illustrate a broader problem with the choice of our equivalence relation.

This toy example illustrates a broader issue with how we choose our equivalence relation. A more realistic scenario involves a program $\mathbb{C}$ that depends on an interface for sorting an array. Whether $\mathbb{C}$ can distinguish between two implementations of this interface depends on their behaviour. If the implementation is *quicksort* or *bubblesort*, then $\mathbb{C}$ cannot tell the difference—the input/output behaviour of the two functions is the same. But if the interface is implemented incorrectly, for example as the identity function on arrays, then $\mathbb{C}$ will begin to behave incorrectly as it assumes the output of the function will be an ordered array. In this case the choice of implementation becomes observable, revealing that the equivalence relation must be sensitive enough to detect such behavioural differences. Naturally, depending on the language we choose to model we need to take care of different behaviors.

The moral here is that an equivalence relation should be a *congruence* in the sense that if two programs P, Q are equivalent, written $P \approx Q$ then for all contexts $\mathbb{C}$, also $\mathbb{C}[P] \approx \mathbb{C}[Q]$.

For example, in CCS, it must be true that for all $P \approx Q$, then $(P \parallel R) \approx (P \parallel R)$ where the context here is $\mathbb{C}[X] = X \parallel R$. With this in mind we now go through various different ways of comparing processes and eventually we will find our right notion of equivalence.

4.1 Structural congruence

A basic notion of equality can be ascribed to processes which explains when processes are treated as being "the same". This equality relation, which we write as $\equiv$, is known as *structural congruence*, which we define in chunks here based on the main syntactic construct of interest.

Firstly, for parallel composition, structural congruence is defined:

$$P \mid Q \equiv Q \mid P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$P \mid \mathbf{0} \equiv P$$

i.e., parallel composition is commutative and associative, and has the inactive process $\mathbf{0}$ as a unit; parallel composition is a commutative monoid with $\mathbf{0}$.

For choice, we get the equations:

$$P + Q \equiv Q + P$$
$$(P + Q) + R \equiv P + (Q + R)$$
$$P + \mathbf{0} \equiv P$$
$$P + P \equiv P$$

Thus, choice is commutative, associative, has $\mathbf{0}$ as a unit, and is also idempotent (lat equation): choosing between the same thing is no choice at all.

Finally, for restriction we have some more involved rules with side conditions, that relate to how we can distribute bindings (that is, move them with respect to other pieces of syntax). In these equations we make use of a meta-level operation on processes $\mathsf{FN}(P)$ which returns the set of free names (i.e. unrestricted names) in P. For example $\mathsf{FN}(a.b.\mathbf{0}) = \{a, b\}$ but $\mathsf{FN}(a.\mathbf{0} \mid \nu b.(b.\mathbf{0})) = \{a\}$.

$$\begin{aligned}
\nu a.P &\equiv P && \text{if } a, \overline{a} \notin \mathsf{FN}(P) \\
\nu a.(P + Q) &\equiv \nu a.P + \nu a.Q \\
\nu a.(P \mid Q) &\equiv \nu a.P \mid \nu a.Q && \text{if } (a \in \mathsf{FN}(P) \Rightarrow \overline{a} \notin \mathsf{FN}(Q)) \wedge (a \in \mathsf{FN}(Q) \Rightarrow \overline{a} \notin \mathsf{FN}(P)) \\
\nu a.(P \mid Q) &\equiv (\nu a.P) \mid Q && \text{if } a, \overline{a} \notin \mathsf{FN}(Q)
\end{aligned}$$

The first allows us to drop redundant restrictions. The second allows us to distribute restriction inside choice. The third allows us to distribute restriction inside parallel composition only if P and Q cannot handshake via a. The fourth equation states that we can shrink or grow the scope of a restriction over processes that don't mention the restricted name (Q in this case).

We can apply structural congruences to simplify processes by algebraic manipulation. For example:

$$P + (\mathbf{0} \mid P) \equiv P \qquad \text{(example)}$$

4.2 Trace Equivalence

A *trace* of a process captures the sequence of *visible actions* it can perform along some execution path. Formally, a trace is a finite sequence $\alpha_1\alpha_2 \cdots \alpha_n$ of visible actions (a or $\overline{a}$) such that there exists a sequence of transitions

$$P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots \xrightarrow{\alpha_n} P_n$$

in the operational semantics of CCS. The set of all finite traces of a process P is denoted $\mathrm{Traces}(P)$. The empty trace ε is always included. Internal actions τ are ignored in traces because they are invisible to the environment.

To compute traces for a given process, one typically constructs its labelled transition system (LTS), ignores τ-transitions, and enumerates all sequences of visible actions along finite paths starting from the initial state.

Example 7. Consider the processes

$$P = a.b.P \quad \text{and} \quad R = a.Q \quad \text{with} \quad Q = b.a.Q.$$

Intuitively, P repeatedly performs a then b, while R performs a followed by b then a, and so on. The traces of P are

$$\varepsilon, a, ab, aba, abab, \ldots$$

corresponding to any finite prefix of the infinite repetition $ab\,ab\,ab\ldots$. For R, the first action is a, and Q then produces $ba\,ba\,ba\ldots$, so the finite prefixes are

$$\varepsilon, a, ab, aba, abab, \ldots$$

which coincides with the traces of P.

This shows that although P and R have different internal structures in their LTSs, their traces are identical. Hence *trace equivalence* is coarser than structural congruence, as it abstracts away from the precise arrangement of states and only records sequences of visible actions.

Now consider the following example.

Example 8. Consider the two CCS processes

$$P' = a.(b + c) \quad \text{and} \quad Q' = a.b + a.c.$$

The process P' performs the action a first and then makes an internal choice between b and c, while Q' makes the choice between $a.b$ and $a.c$ immediately. Computing the traces, we see that for P' the possible sequences of visible actions are: doing nothing ε, performing only a, performing a then b, and performing a then c, giving

$$\text{Traces}(P') = \{\varepsilon, a, ab, ac\}.$$

For Q', the choice occurs before a, but the observable sequences of visible actions are identical: doing nothing ε, performing a alone, then a followed by b, and a followed by c, yielding

$$\text{Traces}(Q') = \{\varepsilon, a, ab, ac\}.$$

Hence, although the internal branching structure of the two processes differs, the sets of traces are the same, and therefore P' and Q' are *trace-equivalent*.

The problem with this example, is that while the process $(P'||\overline{a}.\overline{b})$ can synchronise with a and then b, the process $(Q' \parallel \overline{a}.\overline{b})$ might deadlock in case Q' decides to to choose to branch right. Hence the context $\mathbb{C}[\cdot] = [\cdot] \parallel \overline{a}.\overline{b}.0$ can tell the difference between P' and Q'.

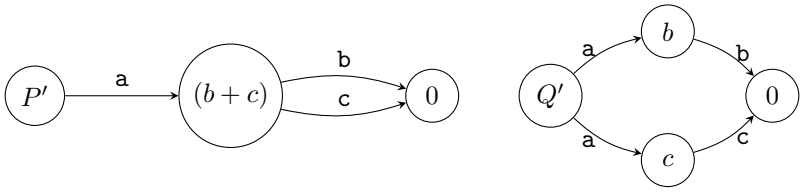
Hence trace equivalence is not a congruence. The issue here is that trace equivalence does not distinguish between P and Q and so we need a stricter notion of equivalence.

4.3 Graph Isomorphisms

If we examine the processes from the perspective of *graph isomorphism*, a stricter notion, the situation differs. A process can be represented as an LTS, a directed graph in which nodes correspond to process states, edges are labelled by actions, and the initial node corresponds to the process itself.

Intuitively, two processes are graph-isomorphic if their LTSs have the *same structure*: the same number of states connected in the same way by the same labelled actions. Graph isomorphism is a stricter equivalence than trace equivalence because it considers the *branching and connectivity* of states, not just the sequences of observable actions.

Now consider again the processes P' and Q' from previous section. These are not graph-isomorphic as the following picture depicts



hence there is no requirement for them to be graph-isomorphic under any context. Therefore the condition that graph isomorphisms constitute a congruence is satisfied[1].

The problem is that this notion of equivalence is so strict that it distinguishes the previous example where P and R were trace equivalent. These two processes continue to be trace equivalent under all contexts, but they are not graph-isomorphic as their structure does not look like to have the same shape:

This is now a problem because these two processes really should be regarded as equivalent.

Therefore we need something in between.

4.4 Bisimulation

Often we want to be able to reason about whether two processes behave in the same way. For example, when programming, we might come up with a refactoring or simplification which we want to ensure behaves in the same way as the old program. Or, we might want to use one process as a specification for another, giving a simple but inefficient definition as the specification, against which we want to compare the complicated but efficient implementation that we use in production. We can describe process equivalence by comparing process behaviour: i.e., do two processes behave in the same way?

The notion of behavioural equivalence we study in this course is called *strong bisimulation*. This notion of behavioural equivalence compares step-by-step what two processes can do. Roughly, we say that a process P is *bisimilar* to a process Q if whenever P can make a reduction step with label l, then so can Q (and vice versa), and the processes reached after reduction are also bisimilar. This is encapsulated by the following definition, which is actually a property of a relation:

Definition 2 (Bisimulation)**.** A relation $\mathcal{R}$ between processes is a bisimulation relation iff it satisfies the following: for all processes P and Q then:

$$\begin{aligned} P\mathcal{R}Q \quad & \Rightarrow \forall P', l.\ \ P \xrightarrow{l} P' \ \Rightarrow\ (\exists Q''.Q \xrightarrow{l} Q'' \ \wedge\ P'\mathcal{R}Q'') \\ & \wedge\ \ \forall Q', l.\ \ Q \xrightarrow{l} Q' \ \Rightarrow\ (\exists P''.P \xrightarrow{l} P'' \ \wedge\ P''\mathcal{R}Q') \end{aligned}$$

Definition 3 (Strong bisimulation)**.** Two processes P and Q are *bisimilar* iff $P \sim Q$ where $\sim$ is the largest bisimulation, defined:

$$\sim = \bigcup \{\mathcal{R} \mid \mathcal{R} \textit{ is a bisimulation}\}$$

(i.e. $\sim$ incorporates all possible bisimulation relations over processes).

The following are some small examples:

- $(a.\mathbf{0} \mid \overline{a}.\mathbf{0}) \ \sim\ a.\overline{a}.\mathbf{0} + \overline{a}.a.\mathbf{0} + \tau.0$;
- $P \ \sim\ a.Q$ where $P \stackrel{\text{def}}{=} a.b.P$ and $Q \stackrel{\text{def}}{=} b.a.Q$;
- $a.(b + c) \not\sim a.b + a.c$ (i.e., distributing a prefix inside a choice leads to different behaviour).

The lecture gives more examples and motivates this definition further.

[1] One can easily verify that graph isomorphisms are congruences in the category of graphs

Example 9. Why is it the case that $a.(b.\mathbf{0} + c.\mathbf{0}) \not\sim a.b.\mathbf{0} + a.c.\mathbf{0}$? Let's take it a step at a time.

Let $P = a.(b.\mathbf{0} + c.\mathbf{0})$ and let $Q = a.b.\mathbf{0} + a.c.\mathbf{0}$. We must consider every transition that P can make. In this case, P can just do one transition $a.(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{a} (b.\mathbf{0} + c.\mathbf{0})$. Let $P' = b.\mathbf{0} + c.\mathbf{0}$.

Now we need some transition $Q \xrightarrow{a} Q'$ such that $P' \sim Q'$. There are two such possible transitions for Q— we need just one to work. Let's try each in turn.

- $(a.b.\mathbf{0} + a.c.\mathbf{0}) \xrightarrow{a} b.\mathbf{0}$ (by *choice1*) thus $Q' = b.\mathbf{0}$.

  We now consider the recursive case of the bisimulation argument: $P' \sim Q'$? i.e. $b.\mathbf{0} + c.\mathbf{0} \sim b.\mathbf{0}$? We need to look at all possible transitions that P' can make. Since $P' = b.\mathbf{0} + c.\mathbf{0}$ then there are two possibilities both of which we need to consider:

  - $(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{b} \mathbf{0}$.

    Can Q' do this transition? Yes. $Q' = b.\mathbf{0} \xrightarrow{b} \mathbf{0}$. And trivially $\mathbf{0} \sim \mathbf{0}$.
  - $(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{c} \mathbf{0}$.

    Can Q' do this transition? No, since Q' can only do a b transition.

  Therefore, $P' \not\sim Q'$ so applying *choice1* to Q does not lead to a bisimilar situation. Our only hope is the other possibility for Q' which we try in the next case:

- $(a.b.\mathbf{0} + a.c.\mathbf{0}) \xrightarrow{a} c.\mathbf{0}$ (by *choice2*) thus $Q' = c.\mathbf{0}$.

  Again we consider the recursive case of the bisimulation argument, i.e., does $P' \sim Q'$? Since $P' = b.\mathbf{0} + c.\mathbf{0}$ we again have two possibility that we need to check:

  - $(b.\mathbf{0} + c.\mathbf{0}) \xrightarrow{b} \mathbf{0}$.

    Can Q' do this transition? No since $Q' = c.\mathbf{0}$ it can only do a c transition. So $P' \not\sim Q'$.

Thus overall $P \not\sim Q$, i.e. $a.(b.\mathbf{0} + c.\mathbf{0}) \not\sim a.b.\mathbf{0} + a.c.\mathbf{0}$.