Computability

The **halting problem** is a program which answers whether a given program, along with its input, would ever terminate.

A problem is said to be **decidable** if it is possible to design a porgram to answer the problem within a finite time.

A problem is **undecidable** if it is impossible to construct an algorithm that leads to the correct answer.

We can prove the halting problem is undecidable: Write an algorithm H such that given:
- the description of an algorithm A (requiring one input)
- an input I

Then H will return 1 if A stops with input I and 0 if it does not.

Assuming H exists, we can then write an algorithm H' such that H' takes an input A and computes $H(A, A)$. If the result is 0 then H' answers 1 and stops, otherwise it loops forever.

Since A is arbitrary, it could be H' itself. We obtain a contradiction: $H'(H')$ stops if and only if $H'(H')$ does not stop.

So H cannot exist.

An **algorithm** has three characteristics:
- has a finite description of a computation in terms of elementary instructions
- is a deterministic procedure: next step ios uniquely defined
- always produces a result no matter the input (i.e. halts)

A **partial function** $f : A \to B$ is a subset of $A \times B$ such that if $(x, y) \in f$ and $(x, z) \in f$ then $y = z$, i.e. there is at most one result in B for each A.

A **total function** $f : A \to B$ is a function for where each element A, there exists an output B.

A function $f : A \to B$ is **computable** if we can construct a deterministic Turing Machine for it or if there is a lambda-term.

The **Church-Turing Thesis** states that any language that can be effectively computed by some finite process (λ-computable) can be recognised by a Turing machine.

Automata

An **alphabet** is a finite set S of symbols, e.g. $\{a, b, c, \ldots, z\}$.

A **word** or **string** is a finite sequence of symbols from S written without commas, e.g. 1110.

The **Comsky hierarchy** categorises languages into collections according to their difficulty.

Type	Abstract Machine
Type-0	Turing Machine (recursively-enumerable languages)
Type-1	Linear-bounded Turing Machine
Type-2	Pushdown Automata (context-free languages)
Type-3	Finite Automata

Finite automata with alphabet $\mathcal{X}$ is defined by a finite number of states and a set of transitions between states, we use the triple $(\mathcal{X}, \mathcal{Q}, q_i, F, \delta)$ (alphabet, states, initial, accepting, transition fn).

Non-deterministic finite automata with alphabet $\mathcal{X}$ is defined by a finite number of states and a set of *some* transitions between states, which may also include ε, the empty transition.

Non-deterministic and deterministic finite automata are computationally equivalent.

The **pumping lemma** says if $L \subseteq \Sigma^*$ is a regular language then there is some $p > 0$ such that every word $w \in L$ of length $|w| \geq p$ can be written in the form $w = xyz$ for $x, y, z \in \Sigma^*$:

$$\text{If } |xy| \leq p \text{ and } |y| > 0$$
$$\text{then } xy^n z \in L \text{ for all } n \geq 0.$$

Pushdown automata are finite automata with a stack, where the stack stores symbols. The stack is a specialised form of memory where only the top of the stack can be read (removing the item from the stack) and new elements must be added to the top.

They are defined using the tuple $(\mathcal{X}, \mathcal{Q}, \Gamma, q_i, F, \delta)$, where Γ is the set of symbols that can be stored on the stack.

The transition function δ maps $(\text{state}, \text{letter}, \text{symbol})$ to a set of pairs of the form $(\text{state}, \text{symbol})$. If the symbol is ε, then nothing is to be popped / pushed.

A word is therefore accepted if there is any trace from initial state to accepting state that leaves the stack empty.

Turing Machines

A **Turing machine** is a tuple $(\mathcal{X}, \mathcal{Q}, \Gamma, q_i, F, \delta)$ where Γ is the set of symbols that can be stored on tape (usually $\{\circ, \bullet\} \in \Gamma$), $F = \left\{q_{\text{accept}}, q_{\text{reject}}\right\} \subseteq \mathcal{Q}$, and $\delta : \mathcal{Q} \times \Gamma \to \mathcal{Q} \times \Gamma \times \{L, R\}$.

The transitions function maps $(\text{state}, \text{symbol})$ to $(\text{state}, \text{symbol}, \text{direction})$.

The turing machine works as follows:

- Initially, tape stores input word, the head in first symbol of input string, and machine in state q_i.
- As computation proceeds, configuration is described by triple of current state, contents of state, and position of head on tape.
- Word is accepeted if turing machine reaches q_{accept} or rejected if reaches q_{reject}.

  The machine may also loop forever.

A function that can be implemented by a Turing machine is called **Turing computable**.

The **Universal Turing Machine** is a Turing machine that is capable of reading the 'code' of another Turing machine along with its input, simulating the execution of the machine on this input.

Variants of Turing Machines

- Non-deterministic turing machines may have one or more alternative transitions for any given state.
- A turing machine may have one or more tapes.
- A turing machine may have a tape with a starting point, but unlimited space to the right.
- We could define a reading, writing, read/write tape.

Applications of Automata

Syntactic analysis is the process of checking whether a program is snyntactically correct.

The compilation process consists of:
1. **Lexical analysis**: programs decomposed into tokens
2. **Parsing**: tokens checked to ensure they form a word in the programming language

Parsing

A **parser** reads a sequence of tokens and builds a parse/derivation tree, outputting an *abstract syntax tree*.

The **abstract syntax** describes the syntax trees, to which semantics is associated.

In **top-down parsing** (recursive descent parsing), (starting from start symbol) search for a rule that rewrites the non-terminals to yield terminals consistent with input.

A grammar is suitable for RDP (recursive descent parsing) if, wherever there is a choice, the parser can select the correct alternative by looking at the current symbol, otherwise we may end up backgracking a lot which is inefficient.

We can make grammars less expensive/reduce backtracking by **left factoring**: expanding rules to expose the common part and then factoring it out.

To eliminate left recursion from any rule, we apply the following substitution:

$$\langle A \rangle \to \langle A \rangle \alpha \mid \beta$$

$$\begin{aligned} \langle A \rangle &\to \beta \langle A' \rangle \\ \langle A' \rangle &\to \varepsilon \mid \alpha \langle A' \rangle \end{aligned}$$

We may run into infinite loops if the grammar is indirectly left-recursive:

$$\begin{aligned} \langle A \rangle &\to \langle B \rangle \alpha \mid \beta \\ \langle B \rangle &\to \langle A \rangle \gamma \mid \delta \end{aligned}$$

We can rewrite indirect left-recursion as an equivalent direct left-recursion by using the substitution rule:

$$\begin{gathered} \langle A \rangle \to (\langle A \rangle \gamma \mid \delta) \alpha \mid \beta \\ \text{is equivalent to} \\ \langle A \rangle \to \langle A \rangle (\gamma \alpha) \mid \delta \alpha \mid \beta \end{gathered}$$

In **bottom-up parsing** (shift-reduce), we start from the input and compare it against the right-hand side of the rules, to find where a string can be replaced by a non-terminal. Parsing succeeds when the whole input has been replaced by the start of the grammar.

A **parser generator** generates a parser from a given grammar.

Lexical Analysis

A **lexer** reads a string and converts it to a sequence of tokens.

A **regular expression** r denotes a language $L(r)$ (set of acceptable words).

A **regular definition**, over an alphabet $\mathcal{X}$, is a sequence of definitions:

$$\begin{aligned}\langle d_0 \rangle &\to r_0 \\ \langle d_1 \rangle &\to r_1 \\ &\dots \\ \langle d_n \rangle &\to r_n\end{aligned}$$

Where each d_i is a distinct name and each r_i is a regular expression over $\mathcal{X} \cup \{d_0, \dots, d_{i-1}\}$.
Symbols in $\mathcal{X}$ are called terminals, and $\{d_0, \dots, d_n\}$ are called non-terminals.

Lambda Calculus

The Turing Machine is an **imperative** computational model while The Lambda Calculus is a **functional** computational model.

Functions in Lambda Calculus are black-boxes, they do not have an internal representation or implementation and can be seen as pure operations.

Notation

The Lambda Calculus syntax is built up from three components:

1. **Variables**: assume variables are taken from an infinite set $\{x, y, z, ...\}$
2. **Abstractions**: if x is a variable and M is a term, then $(\lambda x.M)$ is a term
3. **Applications**: if M and N are terms then (MN) is a term

Syntax

Application associates to the left, instead of writing $((MN)P)$, we can write MNP.

Abstraction associates to the right, instead of writing $\lambda x.(\lambda y.M)$, we can write $\lambda x.\lambda y.M$ or just $\lambda xy.M$.

Free and Bound Variables

A **bound variable** is defined by the given context, e.g. x in $\lambda x.x$.

A **free variable** is defined outside the given context, e.g. y in $\lambda x.xy$.

The set of free variables of a term M, $FV(M)$ is defined as a recursive function:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \end{aligned}$$

Terms without free variables are **combinators/closed terms**.

The set of bound variables of a term M, $BV(M)$ is defined as a recursive function:

$$\begin{aligned} BV(x) &= \emptyset && = \{\} \\ BV(\lambda x.M) &= \{x\} \cup BV(M) \\ BV(MN) &= BV(M) \cup BV(N) \end{aligned}$$

α-equivalence

Recall that $\lambda x.M$ represents a function $f(x) = M$ for all x. Therefore, name of bound occurrences is immaterial.

So $\lambda x.x$ and $\lambda y.y$ represents the exact same function as they only differ on name of bound variable occurrences, so we call them α-equivalent.

$$\lambda x.x \underset{\alpha}{=} \lambda y.y$$

We say a variable has been **captured** when we rename a variable in such a way that a variable that was free before is now bound, this will not preserve the meaning of the term.

Computation

Computation in the Lambda Calculus is composed of a series of substitution rewritings, β-reductions.

A **redex** is a term of the form $(\lambda x.M)N$.

The β-**reduction rule** is $(\lambda x.M)N \underset{\beta}{\to} M\{x \mapsto N\}$.
Where $M\{x \mapsto N\}$ is the term obtained after substituting free occurrences of x by N.

We may apply β-reduction to any redex in a term/in any order.

Examples

To represent numbers, we use the **Church Numerals**:

$$\overline{0} = \lambda xy.y$$
$$\overline{1} = \lambda xy.xy$$
$$\overline{2} = \lambda xy.x(xy)$$
$$\overline{3} = \lambda xy.x(x(xy))$$
$$\dots$$

To define the **successor function**, the function takes a number $\overline{n}$ and returns $\overline{n+1}$:

$$S = \lambda xyz.y(xyz)$$

To define an **addition function**:

$$\text{ADD} = \lambda xyab.(xa)(yab)$$

To encode boolean values:

$$\text{FALSE} = \lambda xy.y$$
$$\text{TRUE} = \lambda xy.x$$

To define the "not" boolean operator:

$$\text{NOT} = \lambda x.x \text{ FALSE TRUE}$$

Curry's fixed point combinator is defined as:

$$Y = \lambda f.(\lambda x.(f(xx))(\lambda x.f(xx)))$$

Used for recursion in Lambda Calculus.

Normal Forms

We can keep applying β-reductions until:

1. **Normal form**: stop when there are no more redexes left to reduce.
2. **Weak-head normal form** (WHNF): stop when all redexes are under an abstraction.

Some terms do not have a normal form such as $(\lambda q.qq)(\lambda q.qq) = Q$ hence $Q \underset{\beta}{\to} Q$.

Such terms represent non-terminating computations.

Confluence

λ-terms may have multiple redexes, therefore multiple β-reductions.

β-reductions are **confluent**.

Church-Rosser Theorem: If $M \overset{*}{\underset{\beta}{\to}} M_1$ and $M \overset{*}{\underset{\beta}{\to}} M_2$ then there exists a term M_3 such that $M_1 \overset{*}{\underset{\beta}{\to}} M_3$ and $M_2 \overset{*}{\underset{\beta}{\to}} M_3$.

Reduction strategies

A **reduction strategy** can make a difference in efficiency of computation, as number of reductions steps may be different.

Strategy	Example
Outermost, Leftmost	$\underline{(\lambda a.a)((\lambda b.b)c)((\lambda d.d)e)}(\lambda f.f)(gh)$
Innermost, Leftmost	$(\lambda a.a)\Big(\underline{(\lambda b.b)c}\Big)((\lambda d.d)e)(\lambda f.f)(gh)$
Outermost, Rightmost	$(\lambda a.a)((\lambda b.b)c)((\lambda d.d)e)\underline{(\lambda f.f)(gh)}$

The **outermost-leftmost** strategy is guaranteed to find the normal form if one exists, but may not be the most efficient strategy.

Interaction Nets

Agents are nodes with:
- one **principal port** (outgoing edge)
- set of n **auxiliary ports** (other connected edges, **ordered**)
- a **type** that determines its interactions (node symbol)

Agents correspond with functions.

An **interaction net** is a graph of agents, where agents are connected at their ports (1:1).
If a port is not connected to another agent, we say it is free.
Nets correspond with programs.

The **interface** of a net is its set of free ports.

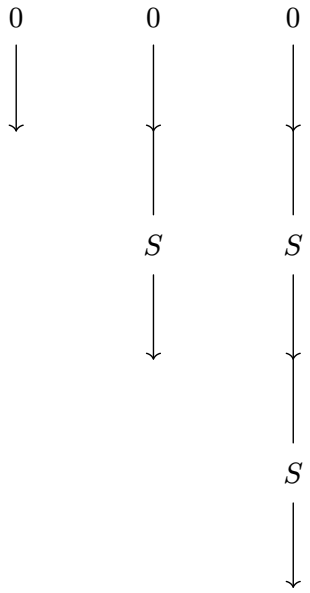
There are two special nets:
- the empty net
- wirings (nets with only edges)

An **active pair** is a pair of agents connected by their primary ports.

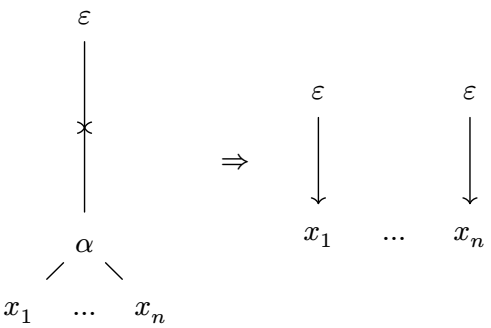
An **interaction rule** replaces an active pair of agents (α, β) by a new net N which must have the same interface as the active pair.
There may be at most one interaction rule for each pair of agent types.
Reduction is local so only agents in active pair are rewritten.

Examples

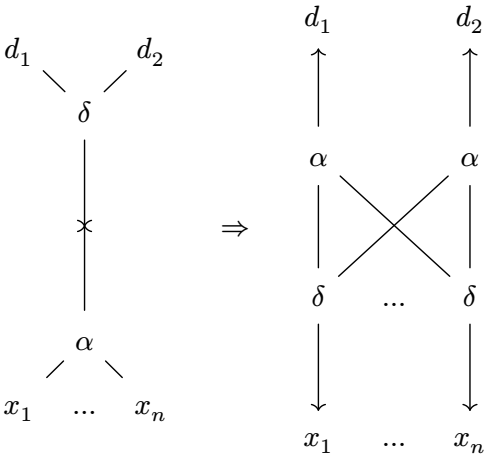
Define natural numbers using an interaction net: use a 0 agent and successor agent S.



Define erasing agent (erase sub-nets):



Define duplicating agent (copy same operation α to two sub-nets d_1, d_2):



Normal Forms

We stop execution when there are no more active pairs, we are in the **normal form**.

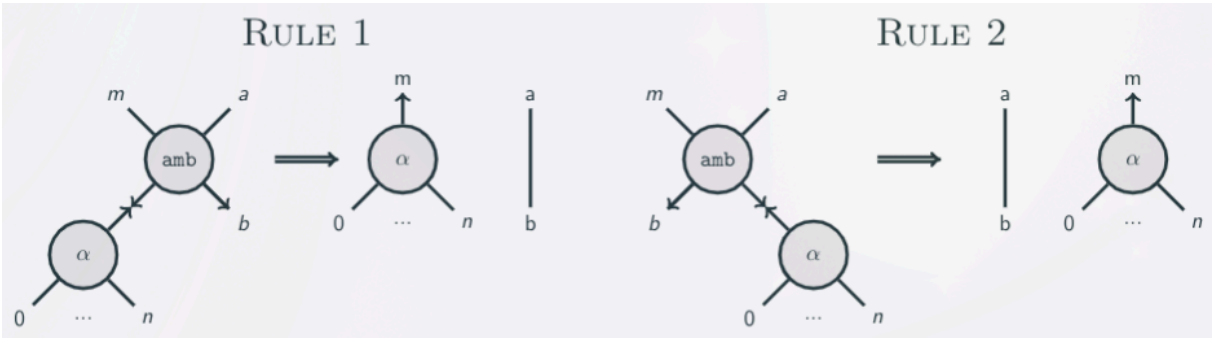
Not all nets have a normal form, they may be non-terminating.

Confluence

Interaction nets have **strong confluence** meaning if $M \Rightarrow M_1$ and $M \Rightarrow M_2$, there exists M_3 such that $M_1 \Rightarrow M_3$ and $M_2 \Rightarrow M_3$.

Non-deterministic extension

We introduce the `amb` agent which has two principal ports. The agent may resolve the interaction at random.



Concurrent Computational Models

Properties of concurrent computation

- **Non-termination**: the program(s) may not terminate, and additionally the output of a program may not be the focus of the computation
- **Parallelism**: steps of computation can occur simultaneously
- **Interference**: the meaning of a program may depend on the behaviour of other programs that are being executed
- **Non-determinism**: the same computations do not always produce the same results

The calculus of communicating systems (CCS)

Milner's general model:

- A **concurrent system** is a collection of processes
- A **process** is an independent agent that may perform internal activities in isolation, or may interact with the environment to perform shared activities

Example of CCS

A vending machines can be represented as follows:

$$\text{coke.coin.}\overline{\text{coke\_can}}.0 + \text{chocolate.coin.}\overline{\text{chocolate\_bar}}.0$$

With the following syntax:

1. Actions: coke, $\overline{\text{coke\_can}}$, ...
2. Sequential computation: . (the dot)
3. Non-deterministic choice: $+$
4. Terminal process: 0

Processes perform actions ($\overset{\text{action}}{\rightarrow}$) and become a new process:

$$\text{coke.coin.}\overline{\text{coke\_can}}.0 + \text{chocolate.coin.}\overline{\text{chocolate\_bar}}.0$$

$$\overset{\text{coke}}{\rightarrow}\ \text{coin.}\overline{\text{coke\_can}}.0$$

$$\overset{\text{coin}}{\rightarrow}\ \overline{\text{coke\_can}}.0$$

$$\overset{\overline{\text{coke\_can}}}{\rightarrow}\ 0$$

Syntax

The simplest possible process is a terminal process, represented by 0, which is a **terminated**, **inactive**, or **deadlocked** process.

A set of **labels** for actions: $\mathcal{A} = \mathbb{N} \cup \overline{\mathbb{N}} \cup \{\tau\} = \{a, b, c, ...\} \cup \left\{\overline{a}, \overline{b}, \overline{c}, ...\right\} \cup \{\tau\}$.

- **Input actions**: $i \in \mathbb{N}$ represents the receiving of an input
- **Output actions**: $o \in \overline{\mathbb{N}}$ represent the sending of an output
- **Internal actions**: τ represents an internal action

The simplest behaviour is **sequential action**. If P is a process, we write cloze($\alpha.P$) to denote the **prefixing** of P with action α.
$\alpha.P$ models a system that is ready to perform the action α and then it will behave as P.

$$\text{i.e. } \alpha.P \xrightarrow{\alpha} P$$

Alternative behaviours can be used to model **non-deterministic choice**. If P and Q are processes, then we can use $P + Q$ to denote the non-deterministic choice between the two.
$P + Q$ models a process that can behave as P (discarding Q) or behave as Q (discarding P).

$$\text{coke.coin.}\overline{\text{coke\_can}}.0 + \text{chocolate.coin.}\overline{\text{chocolate\_bar}}.0 \xrightarrow{\text{coke}} \text{coin.}\overline{\text{coke\_can}}.0$$

We can define **process constants**, that can be defined recursively.

$$P ::= a.b.c.0$$
$$P \text{ is defined as a constant for the process } a.b.c.0$$

Example: a ticking clock $C ::= \text{tick.tock.}C$:

$$\text{tick.tock.}C \xrightarrow{\text{tick}} \text{tock.}C \xrightarrow{\text{tock}} C = \text{tick.tock.}C \rightarrow ...$$

This allows process to continue ad infinitum.

A **program** is a collection of process constants.

Concurrent behaviour can be modelled with **parallel composition**. If P and Q are processes, we write $P|Q$ to denote the parallel composition of the two.

- Each process may proceed independently.
- If P is ready to perform action a and Q is ready to perform the complementary action $\overline{a}$, they may interact ($\xrightarrow{\tau}$, perform step internal to the process).

We can control unwanted interactions between processes by **restricting** action/co-action pairs.
If P is a process and a is an action, we write $\nu a.P$ for the restriction of a in P.

- P can no longer take action a or $\overline{a}$ with external processes.
- P can take actions a and $\overline{a}$ together as an internal action (τ-action).

CCS Grammar

$P \rightarrow K$	| Process constants
$\alpha.P$	| Action prefixing, $\alpha \in \mathcal{A}$
$\sum_{i \in I} P_i$	| Summation, $P_0 + P_1 + ... + P_i$
$P_1 \mid P_2$	| Parallel composition
$\nu a.P$	| Restriction, $\alpha \in \mathcal{A}$
0	Terminal process

Labelled Transition System

We are able to describe concurrent programs using CCS, and we introduce LTS to capture their behaviour.

A **labelled transition system** (LTS) is a triple $\left(\mathcal{P}, \mathcal{A}, \left\{\xrightarrow{\alpha} \mid \alpha \in \mathcal{A}\right\}\right)$ where

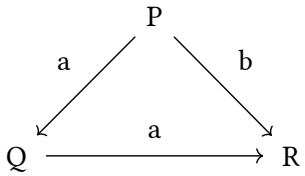
- $\mathcal{P}$ is a set of processes (nodes)
- $\mathcal{A}$ is a set of actions (labels)
- For every $\alpha \in$

Example LTS

Given the following LTS:

$$\left\{\{P, Q, R\}, \{a, b, \tau\}, \left\{P \xrightarrow{a} Q, P \xrightarrow{b} R, Q \xrightarrow{a} R\right\}\right\}$$

Corresponds to the following directed graph:

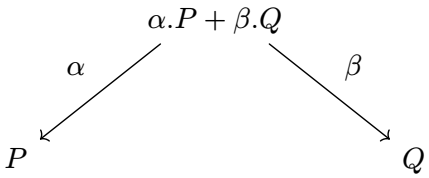


Converting CCS to LTS

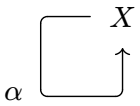
Action Prefix: $\alpha.P$

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

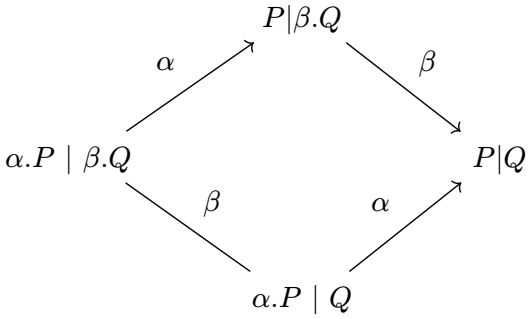
Non-deterministic choice: $\alpha.P + \beta.Q$



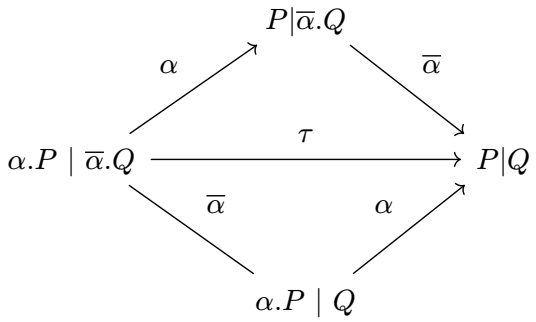
Recursion: $X \coloneqq \alpha.X$



Parallel composition: $\alpha.P \mid \beta.Q$



Parallel composition (interaction): $\alpha.P \mid \overline{\alpha}.Q$



Terminal process: 0

$$0$$

Restriction: $\nu\alpha.(\alpha.P)$

$$\nu\alpha.(\alpha.P)$$

Restriction and interaction: $\nu\alpha.(\alpha.A \mid \overline{\alpha}.B)$

$$\nu\alpha.(\alpha.P)$$

$$\nu\alpha.(\alpha.A \mid \overline{\alpha}.B) \xrightarrow{\tau} \nu\alpha.(A|B)$$

Formal semantics for CCS

The LTS $\left(\mathcal{P}, \mathcal{A}, \left\{\overset{\alpha}{\to} \mid \alpha \in \mathcal{A}\right\}\right)$ of a CCS process is as follows:

- $\mathcal{P}$ is the set of all CCS process expressions
- $\mathcal{A}$ is the set of all actions (including τ)
- the transition relation is given by rules in table below

RULE	Premise	Transition	Conditions
ACT		$\alpha.P \overset{\alpha}{\to} P$	
SUM	$P \overset{\alpha}{\to} P'$	$M = P \overset{\alpha}{\to} P'$	
CON	$P \overset{\alpha}{\to} P'$	$K \overset{\alpha}{\to} P'$	$K \coloneqq P$
COM_1	$P \overset{\alpha}{\to} P'$	$P \mid Q \overset{\alpha}{\to} P' \mid Q$	
COM_2	$Q \overset{\alpha}{\to} Q'$	$P \mid Q \overset{\alpha}{\to} P \mid Q'$	
COM_3	$P \overset{\alpha}{\to} P', Q \overset{\overline{\alpha}}{\to} Q'$	$P \mid Q \overset{\tau}{\to} P' \mid Q'$	
RES	$P \overset{\alpha}{\to} P'$	$\nu\beta.P \overset{\alpha}{\to} \nu\beta.P'$	$\alpha, \overline{\alpha} \neq \beta$

Equivalence

Bisimilarity: two concurrent processes are equivalent iff an external observer cannot tell their behaviour apart.

Given an LTS $\left(\mathcal{P}, \mathcal{A}, \left\{\overset{\alpha}{\to} \mid \alpha \in \mathcal{A}\right\}\right)$, a binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a **strong bisumulation** iff whenever $(P, Q) \in S$ then for each $\alpha \in (\mathcal{A} - \{\tau\})$:

1. if $P \overset{\alpha}{\to} P'$ then $Q \overset{\alpha}{\to} Q'$ for some Q' such that $(P', Q') \in S$
2. if $Q \overset{\alpha}{\to} Q'$ then $P \overset{\alpha}{\to} P'$ for some P' such that $(P', Q') \in S$

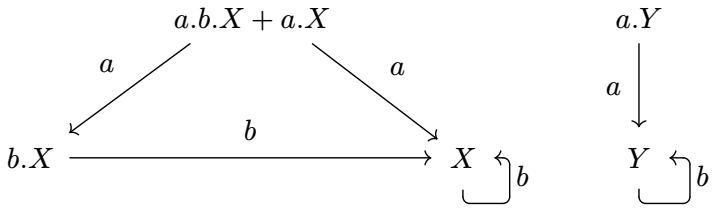
We don't consider τ-actions as an observer can't see internal actions, so they are not part of visible behaviour.

Two processes are **strongly bisimilar** iff there exists a strong bisimulation of their LTSs.

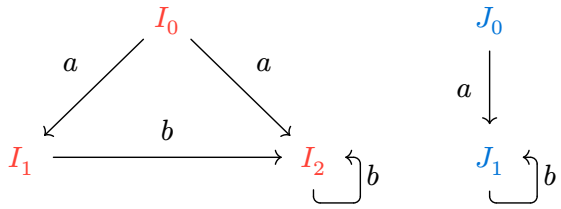
Are the given CCS process expressions equivalent?

$$
\begin{aligned}
I &= a.b.X + a.X \\
J &= a.Y \\
&\text{where} \\
X &::= b.X \\
Y &::= b.Y
\end{aligned}
$$

We begin by constructing the LTSs.



Take note of each state:



We can find a suitable binary relation S that maps between the two:

$$S = \{(I_0, J_0), (I_1, J_1), (I_2, J_1)\}$$

Hence, the processes are equivalent.