PROCESS ALGEBRA

What we have seen so far:

Petri Nets

- INVARIANTS, LINEAR ALGEBRA FOR INCOMPLETE VERIFICATION
- ALGORITHMS FOR
  - TERMINATION — K&M
  - BOUNDEDNESS — K&M
  - COVERABILITY — K&M; Rakoff + Lipton, EXPACE-COMPLETE
  - REACHABILITY — via SEMILINEAR FWD. IND. INV.
- TRANSFER, RESET, INHIBITOR NETS
  - ↳ BOUNDEDNESS UNDECIDABLE
  - ↳ TURING-COMPLETE

Well Structured Transition Systems

- WQOs CHARACTERISATIONS AND CONSTRUCTIONS
- SIMULATION/MONOTONICITY PROPERTY
- BACKWARD SEARCH
  - ↳ upclosed sets, minimal elem., minpre
- FORWARDS SEARCH
  - ↳ dwnclosed sets, ideals, $\widehat{post}$
- INSTANCES of FWD/BWD SEARCH for
  - PN VARIANTS
  - LCS
    - ↳ Downward closed sets of $\Sigma^* =$ SRE
    - Extension to wqo alphabet

Until now we always saw models of concurrency with the goal of devising algorithms for infinite-state verification:

- The models were automata-like and specified only the INTERNAL MECHANICS of a system
- There was no notion of:
  - interaction and composition of sub-components:
    for example a PN was seen as a collection of atomic entities (places and transitions) rather than sub-PN plugged together
  - interaction with an environment:
    by describing a system with a PN all we are doing is specifying its internal dynamics in every detail, as a closed system.
  - equivalence of behaviour:
    it is not clear what it means for two PN to implement the same behaviour.

In considering the topic of PROCESS ALGEBRA we now need a radical shift in our point-of-view: these issues now become fundational, and for the moment we do not worry about verification algorithms.

The crucial question here is: WHAT IS (CONCURRENT) BEHAVIOUR?

Process Algebra is chiefly concerned with understanding these key aspects of concurrent computation:

1. INTERACTION — As in Reactive System: what does it mean for a system to interact with an environment?
   Key concept: for the environment, the system is a BLACK BOX that can only be probed by direct interaction/experiment

2. BEHAVIOURAL EQUIVALENCE — What does it mean for two systems to be equivalent? To implement the same 'black box' behaviour? To be indistinguishable from the environment's point-of-view?

3. CONCURRENCY & COMMUNICATION — How do two concurrent systems interact and synchronise with each others? How can we (compositionally) describe a concurrent system?

4. MOBILITY & COMMUNICATION TOPOLOGIES — How do concurrent agents acquire knowledge of each other? How do they form networks and how do these networks evolve during time?

In this part of the lecture we will conceptualise and formalise each of these four aspects, and use the corresponding theory to prove properties of systems using manual algebraic methods or verification procedures.

1. INTERACTION: We introduce Labelled Transition Systems to represent the behaviour of a system from an external observer's point of view. The observer, or environment, sees the system as a black box offering possible ACTIONS that the environment can trigger (thus performing an 'experiment'). The result of triggering an action is a change in the internal (unobservable) state of the system, which will offer another set of actions and so on.

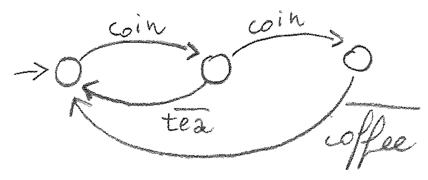
Def A Labelled Transition System (LTS) over a set of actions Act, is a tuple $(S, \rightarrow)$ where

- S is a (typically infinite) set of configurations (the internal states of the system)
- $\rightarrow \subseteq S \times Act \times S$ is the transition relation

We write $s \xrightarrow{\alpha} t$ when $(s, \alpha, t) \in \rightarrow$

An LTS is finitely branching if for every $s \in S$ there are only finitely many $t \in S$ and $\alpha \in Act$ such that $s \xrightarrow{\alpha} t$.
The LTS we will consider in this course are all finitely branching.

Example Consider a Vending Machine delivering tea (for the cost of one coin) or coffee (costing 2 coins). We can describe its behaviour with the following LTS

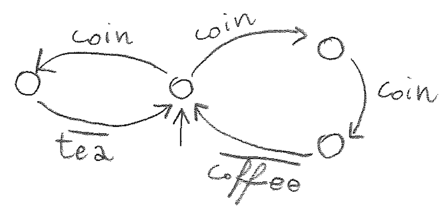


[Here we adopt a convention that will become clearer later: overlined actions are "outputs" and the others are "inputs". We also marked with $\rightarrow\circ$ the intended initial config.]

A user interacting with the machine knows nothing about the internals of it but can observe that after inserting a coin, only tea can be ordered, and after inserting the second coin, only coffee can be ordered.

2 BEHAVIOURAL EQUIVALENCE

Now consider a second model for the vending machine:



Will the user be able to distinguish the two just by interacting with them? Yes! In fact, by inserting a coin in the second machine, the user triggers a choice MADE BY THE MACHINE on which beverage will be offered!

The following definition formalises the idea of being able to distinguish two black boxes, thus embodying in the form of an equivalence, the concept of behaviour.

Def A (strong) simulation over an LTS $(S, \rightarrow)$ is a binary relation $\mathcal{S}$ over S such that $\forall s, s', t \in S\ \forall \alpha \in Act$ if $s \xrightarrow{\alpha} s'$ and $s \mathcal{S} t$ then there is a $t' \in S$ such that $t \xrightarrow{\alpha} t'$ and $s' \mathcal{S} t'$.
We say that s is simulated by t (written $s \lesssim t$) if there is a simulation relating s and t.

A (strong) bisimulation over $(S, \rightarrow)$ is a binary relation $\mathcal{B}$ over S such that both $\mathcal{B}$ and its inverse $\mathcal{B}^{-1}$ are simulations.
We say s is bisimilar to t (written $s \sim t$) if there is a bisimulation relating s and t.

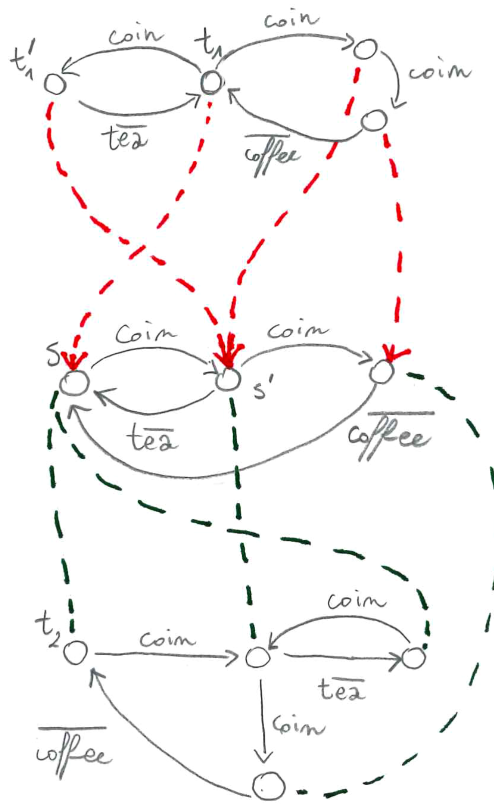
When we relate two distinct LTS $(S_1, \rightarrow_1)$ and $(S_2, \rightarrow_2)$ using (bi)simulations, we implicitly consider (bi)simulations over their union $(S_1 \cup S_2, \rightarrow_1 \cup \rightarrow_2)$.

Remark $s \sim t \Rightarrow s \lesssim t$ and $t \lesssim s$ but the other direction does not hold (see SHEET 9 for a counterex.) Indeed the two simulations proving $s \lesssim t$ and $t \lesssim s$ may be different, while $s \sim t$ requires the same rel. to be a sim. in both directions.

Unless specified otherwise, when we write "(bi)simulation" we mean strong (bi)simulation.

Example Consider the two models of the vending machine and a third one (displayed below)

The red relation is a simulation showing that $t_1 \lesssim s$.
The red relation however fails to be a simulation in the other direction: from s' you can take a tea action which is not available from t_1'.
In fact, no relation is a bisimulation relating t_1 and s, so $s \not\sim t_1$ as we hoped.



The green relation is a bisimulation showing $s \sim t_2$.
The reader can check that the simulation condition is satisfied in both direction by every transition.

Remark It is important to observe that the language from s and the language from t_1 are identical.
Thus an equivalence based on the generated language would be inadequate as it would not be able to differentiate the good behaviour of s and the nasty choice of beverage operated by t_1.

Theorem 1) $\sim$ is an equivalence relation
2) $\sim$ is a bisimulation relation (the largest one)

Proof see NOTES on BISIMILARITY

Now we start exploring ③ CONCURRENCY & COMMUNICATION gradually introducing the key elements.

An important characteristics of Process Algebra is that systems are described by composing subsystems using linguistic constructs, i.e. operators, that model few fondamental ways in which components can be composed.

We start by considering the most basic kind of component: sequential processes.

For reasons that will become clear later, we put some structure on the set of actions Act:

- NAMES:
  We fix a 'universe' of names $\mathcal{N} = \{x, y, z \ldots a, b, c \ldots\}$ which we generally write using lowercase names.
  We use the vector notation $\vec{a} = a_1, \ldots, a_n$ for sequences of names. Generally we use $\vec{x}, \vec{y}, \vec{z}$ for sequences of bound names (see below) so in those cases we assume the sequences do not repeat names.

  if $\vec{a} = a_1, \ldots, a_n$ then $|\vec{a}| = n$ is the length of $\vec{a}$

  We abuse of notation by using sequences of names as sets, e.g. $x \in \vec{a}$.

- ACTIONS:
  We structure the set of actions as $Act := \Lambda \uplus \{\tau\}$
  where $\Lambda = \mathcal{N} \cup \{\bar{a} \mid a \in \mathcal{N}\}$ [Notation: $\alpha \in Act$, $\lambda \in \Lambda$]
  The special action τ is called the 'internal' action.
  Note $\tau \notin \mathcal{N}$.

We also fix a set of Process Identifiers $PID = \{A, B, C \ldots\}$ which we write with titlecase names.
These can be understood as the local control states / program counters of each sequential component / thread.

The syntax of SEQUENTIAL PROCESSES follows the grammar

$$Seq \ni M ::= \underbrace{A[\vec{a}]}_{\text{Process instance}} \mid \underbrace{\alpha_1.M_1 + \ldots + \alpha_n.M_n}_{\text{SUMS}}$$

SUMS often abbreviated with $\sum_{i\in I} \alpha_i.M_i$ for some finite I.

The empty sum is written $\mathbf{0}$ (bold zero, indicating inactivity)

A set of definitions Δ is a set of expressions of the form

$$\underbrace{A[\vec{x}]}_{\text{head}} := \underbrace{\sum_{i\in I} \alpha_i.M_i}_{\text{body}}$$

(parameters: $\vec{x}$)

were all the names occurring in the body occur in $\vec{x}$ and $\vec{x}$ does not repeat names.

We assume Δ contains at most one definition for each $A \in PID$.
Note that Δ is not necessarily finite.
Definitions give meaning to process instances and enable the expression of recursive behaviour.
Technically, we define the relation $\triangleq$ to be the smallest satisfying

$$M \triangleq M \quad \text{and} \quad A[\vec{a}] \triangleq Q[\vec{a}/\vec{x}] \text{ if } (A[\vec{x}] := Q) \in \Delta$$

Where by $[\vec{a}/\vec{x}]$ we denote the substitution mapping $x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$.
Substitutions σ are mappings from $\mathcal{N}$ to $\mathcal{N}$ and apply to Act by substituting the underlying name: $x\sigma = \sigma(x)$, $\overline{x}\sigma = \overline{\sigma(x)}$, $\tau\sigma = \tau$
Substitutions apply to processes as expected $A[a_1, \ldots, a_m]\sigma := A[\sigma(a_1), \ldots, \sigma(a_m)]$ and $(\sum_{i\in I} \alpha_i.M_i)\sigma := \sum_{i\in I} \alpha_i\sigma.M_i\sigma$

<u>Example</u> Consider the definition

$$A[a,b] := \overline{a}.A[b,a]$$

We have

$$A[c,d] \triangleq (\overline{a}.A[b,a])[c,d/a,b] = \overline{c}.A[d,c]$$

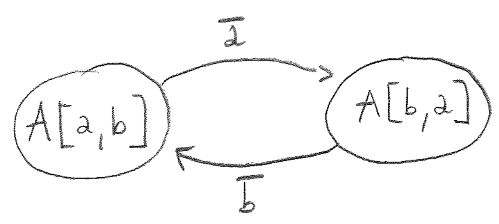
$$A[b,a] \triangleq \overline{b}.A[a,b]$$

To give meaning to expressions denoting sequential processes, we associate to each expression a LTS that fully specifies its behaviour. The configurations of this LTS are seq. proc. expr. themselves. the transitions are defined by the rule:

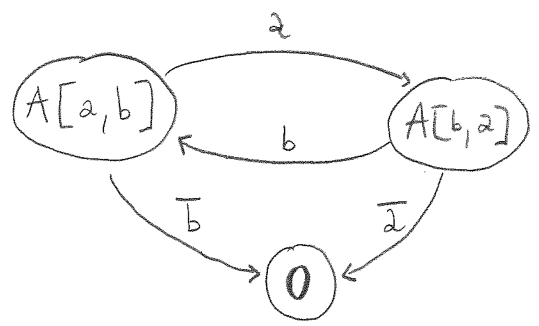
$$M \xrightarrow{\alpha} M_j \quad \text{if} \quad M \triangleq \sum_{i \in I} \alpha_i . P_i \,, \; j \in I$$

<u>Examples</u>

- The definition $A[a,b] := \bar{a}.A[b,a]$ induces the LTS



- The def $A[a,b] := a.A[b,a] + \bar{b}.\mathbf{0}$ induces



We use the abbreviation α for the term $\alpha.\mathbf{0}$ so the def. can be written as

$$A[a,b] := a.A[b,a] + \bar{b}$$

- $A[a] := a.A[a]$ induces $A[a]$ with a self-loop labelled a

The substitutions involved in unfolding this definition are all trivial (identity $a \mapsto a$). In this case we can omit the names that are not subject to substitution from the parameters of the definition:

$A := a.A$ describes the same process

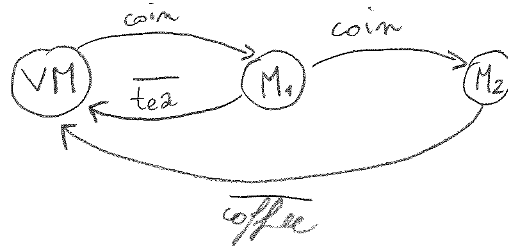
- The vending machine example can be described using the definitions

$$VM[coin, tea, coffee] := coin.(\overline{tea}.VM[coin, tea, coffee] + coin.\overline{coffee}.VM[coin, tea, coffee])$$

which can be abbreviated as

$$VM := coin.(\underbrace{\overline{tea}.VM + coin.\overbrace{\overline{coffee}.VM}^{M_2}}_{M_1})$$

inducing



- The behaviour of a counter can be described as a sequential process that reacts to requests of increments by the environment (inc action) or decrements when non-zero (dec action) and test for zero ($\overline{zero}$ action)

$$Count_0 := inc.Count_1 + \overline{zero}.Count_0$$
$$Count_{n+1} := inc.Count_{n+2} + dec.Count_n \quad \text{for all } n \in \mathbb{N}$$

which induces the infinite-state LTS

Note that in order to model an infinite config. space we needed an infinite set of definitions.

Calculus of Communicating Systems

Now we proceed to extend sequential processes with constructs to represent concurrency and encapsulation. We do this by studying the process algebra 'Calculus of Communicating Systems' (CCS) introduced by one of the fathers of the field, ROBIN MILNER (1934-2010) who won the Turing Award in 1991, for this and other contributions.

The syntax of CCS is described by the grammar

$$CCS \ni P ::= \underbrace{A[\vec{a}] \mid \overbrace{\sum_{i \in I} \alpha_i . P_i}^{\text{sums, ranged over by } M, N \ldots}}_{\text{Sequential processes}} \mid \underbrace{P_1 \parallel P_2}_{\text{Parallel Composition}} \mid \underbrace{\nu a . P}_{\text{Restriction}}$$

(P in $\nu a . P$: scope of the restriction)

As before definitions are in the form $A[\vec{x}] := \sum_{i \in I} \alpha_i . P_i$

The two added ingredients are the parallel operator $\parallel$ which expresses the fact that two components can be put together as independent modules free to interact, and the restriction operator $\nu x . P$ (read 'new x in P') which hides the name x from the environment of P, including other components that may be in parallel with it, offering a form of encapsulation.

The free names of a process P ($fn(P)$) is the set of names visible from outside P:

$fn(\mathbf{0}) := \emptyset$

$fn(A[\vec{a}]) := \vec{a}$

$fn(\sum_{i \in I} \alpha_i . P_i) := \bigcup_{i \in I} fn(\alpha_i) \cup fn(P_i)$

$fn(P_1 \parallel P_2) := fn(P_1) \cup fn(P_2)$

$fn(\nu a . P) := fn(P) \setminus \{a\}$

$fn(a) := fn(\bar{a}) := \{a\}$

$fn(\tau) := \emptyset$

A name is bound in P if it is not free in P.

$bn(P)$ is the set of names occurring bound in P.

Since νa makes a bound, ν is called a 'binder'.

Example $P = \nu a.\big((a.Q_1 + b.Q_2) \parallel \bar{a}\big) \parallel (\bar{b}.R_1 + \bar{a}.R_2)$

P_1: free names b, bound names a

P_2: free names a, b

Intuitively the restriction νa protects the name from interference caused by P_2, so the name a occurring in P_1 is not the same name occurring free in P_2.

To resolve this confusion we can use <u>α-conversion</u> which is the process of renaming a bound name with a fresh name:

$$P =_\alpha \nu a'.\big((a'.Q_1 + b.Q_2) \parallel \bar{a'}\big) \parallel (\bar{b}.R_1 + \bar{a}.R_2)$$

($=_\alpha$: equality up to α-conversion)

To avoid this kind of confusion we can make the following assumption, without loss of generality:

<u>NO-CLASH Assumption</u>: $fn(P) \cap bn(P) = \emptyset$ and we only bind a name if it is not already bound (i.e. the scopes of restrictions of the same name are disjoint)

for example $\nu a\big(a + \bar{a}.(\nu a(\bar{a} + b))\big)$ is not allowed because the inner restriction binds a name that is bound. (scope of νa)

Although we avoid it when writing terms, it is technically allowed to bind the same name in parallel $\nu a.(a + \bar{b}) \parallel \nu a(b + \bar{a})$ (scope of each νa)

<u>Substitutions never rename bound names.</u>

Technically this means that $P\sigma$ is defined (as expected) only when $dom(\sigma) \subseteq fn(P)$ $\{x \mid \exists y\ \sigma(y) = x\} \cap bn(P) = \emptyset$
i.e. when σ only replaces free names and $P\sigma$ satisfies NO-CLASH.

Before specifying the semantics of a process, we want to formalise the idea that some syntactic differences like the order of operands in parallel compositions, are not meaningful: they do not alter the behaviour.

<u>Def</u> A (process) context C is a term with a "hole" (repr. by '$[\,]$')
Formally a context is a term following the grammar

$$C ::= [\,] \mid \alpha.C + M \mid \nu a.C \mid C \parallel P \mid P \parallel C$$

A context C can be understood as a function that maps a process Q to the process $C[Q]$ by replacing the hole with Q.
We call the following context "elementary":

- $\alpha.[\,] + M$
- $\nu a.[\,]$
- $P \parallel [\,]$
- $[\,] \parallel P$

<u>Example</u>
$C = (\bar{a} \parallel b.(\nu c.[\,]) + a)$
$Q = \bar{c} \parallel b.c$
$C[Q] = (\bar{a} \parallel b.(\nu c.(\bar{c} \parallel b.c)) + a)$

<u>Def</u> (Process Congruence) Let $\cong$ be an equivalence relation between processes.
The relation $\cong$ is a (process) <u>congruence</u> if it is preserved by all elementary contexts, that is if $P \cong Q$ then

- $\alpha.P + M \cong \alpha.Q + M$
- $\nu a.P \cong \nu a.Q$
- $R \parallel P \cong R \parallel Q$
- $P \parallel R \cong Q \parallel R$

<u>Proposition</u> An equivalence relation $\cong$ is a process congruence if and only if it is preserved by <u>all</u> contexts, i.e. $\forall C: P \cong Q \Rightarrow C[P] \cong C[Q]$

This means that elementary contexts are the only we need to worry about to see if a relation is a congruence. Similarly if $P \cong Q$ for some congruence $\cong$ then no context can distinguish them.

Def Structural Congruence $\equiv$ is the smallest process congruence satisfying

1. α-equivalent processes are congruent $P =_\alpha Q \Rightarrow P \equiv Q$
2. commutativity and associativity laws for $\parallel$ and $+$ e.g. $P \parallel Q \equiv Q \parallel P$, $M + N \equiv N + M$ etc...
3. $P \parallel \mathbf{0} \equiv P$
4. $\nu a.(P \parallel Q) \equiv P \parallel \nu a Q$ when $a \notin fn(P)$
   This is called SCOPE EXTRUSION law
5. $\nu a.\nu b.P \equiv \nu b.\nu a.P$
   This is called EXCHANGE law
6. $\nu a.\mathbf{0} \equiv \mathbf{0}$

Consider a set of definitions Δ, we extend $\equiv$ to the congruence $\equiv_\Delta$ that in addition to 1 - 6 above satisfies $P \equiv_\Delta Q$ if $P \triangleq Q$, ie. $A[\vec{a}] \equiv_\Delta Q[\vec{a}/\vec{x}]$ if $(A[\vec{x}] := Q) \in \Delta$.

Now we are ready to specify by means of the REACTION relation $\longrightarrow$ the interactions that can take place between the components of a system:

REACTION RULES

$$\mathrm{TAU}_R: \ \tau.R + M \longrightarrow R$$

$$\mathrm{REACT}_R: \ (a.P + M) \parallel (\bar{a}.Q + N) \longrightarrow P \parallel Q$$

$$\mathrm{PAR}_R: \ \frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}$$

$$\mathrm{RES}_R: \ \frac{P \longrightarrow P'}{\nu a.P \longrightarrow \nu a P'}$$

$$\mathrm{STRUCT}_R: \ \frac{Q \equiv_\Delta P \longrightarrow P' \equiv_\Delta Q'}{Q \longrightarrow Q'}$$

The TAU_R rule formalises the idea of an internal action: no interaction is needed to be able to execute a τ action. Similarly REACT_R says that an input action a and an output action $\bar{a}$ can synchronise and make the two involved processes continue with the corresponding branch.

Example Assume $A[a,b] := a.Q_1 + \bar{b}.Q_2$

We can show that $\nu a\,(A[a,b] \parallel \bar{a}) \parallel b \longrightarrow \nu a\,(Q_2 \parallel \bar{a})$ by the following derivation

(recall that this (b) is abbrev. for $b.\mathbf{0}$)

$$\dfrac{\nu a.(A[a,b] \parallel \bar{a}) \parallel b \equiv_\Delta \nu a.(\bar{b}.Q_2 + a.Q_1 \parallel b \parallel \bar{a}) \qquad \dfrac{\dfrac{\dfrac{}{\bar{b}.Q_2 + a.Q_1 \parallel b.\mathbf{0} \longrightarrow Q_2 \parallel \mathbf{0}}\,\text{REACT}_R}{(\bar{b}.Q_2 + a.Q_1 \parallel b) \parallel \bar{a} \longrightarrow (Q_2 \parallel \mathbf{0}) \parallel \bar{a}}\,\text{PAR}_R}{\nu a.(\bar{b}.Q_2 + a.Q_1 \parallel b \parallel \bar{a}) \longrightarrow \nu a.(Q_2 \parallel \mathbf{0} \parallel \bar{a})}\,\text{RES}_R \qquad \nu a.(Q_2 \parallel \mathbf{0} \parallel \bar{a}) \equiv \nu a.(Q_2 \parallel \bar{a})}{\nu a.(A[a,b] \parallel \bar{a}) \parallel b \longrightarrow \nu a.(Q_2 \parallel \bar{a})}\,\text{STRUCT}_R$$

Example We want to model a coin flip: we could be tempted to write $Coin' := \overline{head} + \overline{tail}$ but this is not accurate because the choice of the outcome can be influenced by the environment. for example a player refusing to lose can be $Player := head.Win$

then $Coin' \parallel Player \longrightarrow Win$ is the only transition.

This kind of choice is called "external choice" because the env. can select a branch.

The correct definition employs "internal choice" which in CCS can be modelled by using the τ action

$$Coin := \tau.\overline{head} + \tau.\overline{tail}$$

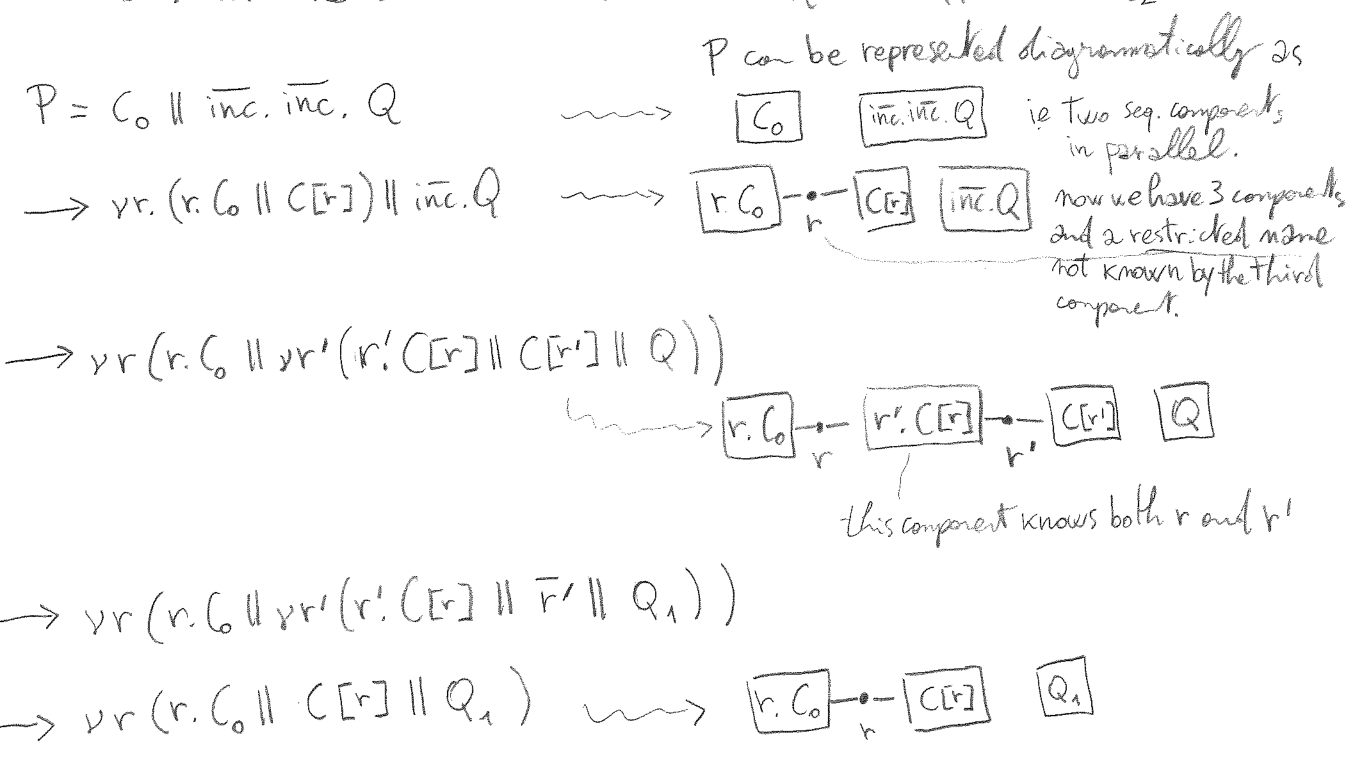
In this case, the coin makes an independent choice between the two outcomes, a choice that cannot be influence by the env.

Example While with sequential processes, in order to define some infinite-state system like $Count_n$ we needed infinitely many definitions, thanks to parallel composition (and restriction) we can describe them using only finitely many definitions. The system will be composed by many sequential components interacting internally to provide the required external behaviour. Let's see how a counter can be implemented:

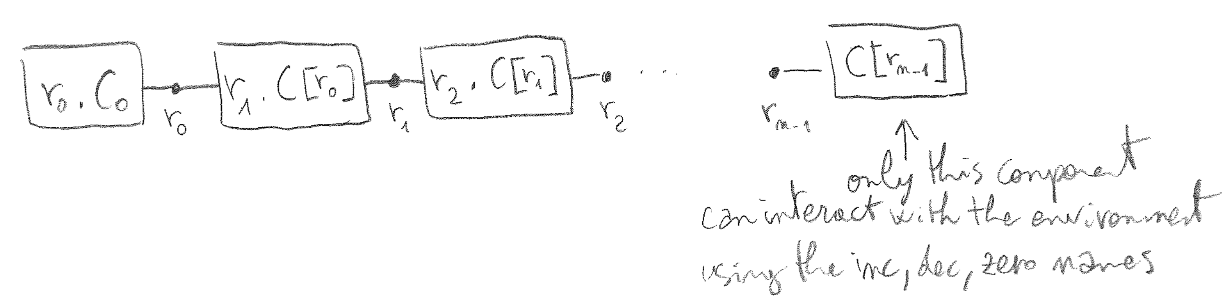
$$C_0 := inc.\nu r.(r.C_0 \parallel C[r]) + \overline{zero}.C_0$$

$$C[r] := inc.\nu r'.(r'.C[r] \parallel C[r']) + \overline{dec}.\overline{r}$$

Consider the term P below where $Q = \overline{dec}.Q_1 + zero.Q_2$



In general a counter holding the value n is repr. with a process



We would like to be able to express the behaviour of C_0 from the perspective of the env. as we did for seq. processes using LTS.
This opens up two opportunities: 1) we can then compare the LTS of C_0 and $Count_0$
2) we can describe internal interaction (reaction) as a special case of interaction with an environment: the case where one component act as the environment of another.

TRANSITION RULES

Notation: $\bar{\bar{a}} = a$, $\lambda \neq \tau$

SUM: $M + \alpha.P + N \xrightarrow{\alpha} P$

REACT: $$\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\bar{\lambda}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'}$$

L-PAR: $$\frac{P \xrightarrow{\alpha} P'}{P \| Q \xrightarrow{\alpha} P' \| Q}$$

R-PAR: $$\frac{Q \xrightarrow{\alpha} Q'}{P \| Q \xrightarrow{\alpha} P \| Q'}$$

RES: $$\frac{P \xrightarrow{\alpha} P'}{\nu a.P \xrightarrow{\alpha} \nu a.P'} \text{ if } \alpha \notin \{a, \bar{a}\}$$

DEF: $$\frac{A[\vec{a}] \triangleq Q \xrightarrow{\alpha} P}{A[\vec{a}] \xrightarrow{\alpha} P}$$

The sum rule simply states that any action in a sum can be taken by continuing with the corresponding branch.
The two PAR rules state that a component can interact with the environment as if it was alone, also in parallel with some other component.
The REACT rule says that when one component can perform the dual action of another, the two can interact resulting in an internal reaction, with label τ.
The RES rule enforces encapsulation: an interaction with the environment cannot involve a name which is protected by a restriction; i.e. all actions involving a restricted name will be internal in its scope.

Example Take the C_0 counter definitions.

Consider $r'.C[r] \parallel \overline{r'}$. Which trasitions can be derived from it?

$$\text{R-PAR}\ \dfrac{\text{SUM}\ \dfrac{}{\overline{r'} \xrightarrow{\overline{r'}} \mathbf{0}}}{r'.C[r] \parallel \overline{r'} \xrightarrow{\overline{r'}} r'.C[r] \parallel \mathbf{0}}
\qquad
\text{L-PAR}\ \dfrac{\text{SUM}\ \dfrac{}{r'.C[r] \xrightarrow{r'} C[r]}}{r'.C[r] \parallel \overline{r'} \xrightarrow{r'} C[r] \parallel \overline{r'}}$$

$$\text{REACT}\ \dfrac{\text{SUM}\ \dfrac{}{r'.C[r] \xrightarrow{r'} C[r]} \quad \text{SUM}\ \dfrac{}{\overline{r'} \xrightarrow{\overline{r'}} \mathbf{0}}}{r'.C[r] \parallel \overline{r'} \xrightarrow{\tau} C[r] \parallel \mathbf{0}}$$

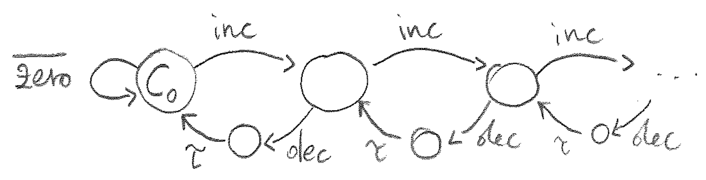
These three are the only possible transitions.

Now if we consider $\nu r'(r'.C[r] \parallel \overline{r'})$ the only applicable rule is RES, and we only have the above 3 cases that can act as a premise. However the side condition $\alpha \notin \{a, \overline{a}\}$ of RES forbids us to use the trasitions labelled with r' and $\overline{r'}$, so we are left with only one possible trasition

$$\text{RES}\ \dfrac{\dfrac{}{r'.C[r] \parallel \overline{r'} \xrightarrow{\tau} C[r] \parallel \mathbf{0}}\ \text{as above}}{\nu r'.(r'.C[r] \parallel \overline{r'}) \xrightarrow{\tau} \nu r'(C[r] \parallel \mathbf{0})}\ \tau \notin \{r', \overline{r'}\}$$

This is the mechanism by which RES protects the restricted actions to be used by external components.

The LTS from C_0 looks as follows



which is <u>almost</u> the same as the one from $Count_0$.

Later we will develop a notion of behavioural equivalence that is able to allow extra τ steps that do not fundamentally alter the behaviour of a system.

But for now let us study strong bisimilarity for CCS.

Strong Bisimulation for CCS

In the reaction rules we postulated that structural congruent processes have the same behaviour. The transition rules are more fundamental and explain reactions by a more general phenomenon.
We now need to check that the two semantics are in agreement w.r.t. internal actions. The first step is showing that $\equiv$ does indeed preserve behaviour.

Theorem $\equiv$ is a strong bisimulation

Proof Since $\equiv$ is symmetric by def, we only need to check that it is a strong simulation, that is if $P \xrightarrow{\alpha} P'$ and $P \equiv Q$ then there is $Q' \equiv P'$ with $Q \xrightarrow{\alpha} Q'$.

Ingredients: The proof uses the following structure which is very common

1. $P \equiv Q$ is the result of applying the structural congruence laws finitely many times to obtain Q from P. So we only need to prove the claim for the case where Q is obtained by applying a single law once to P; the general case would follow by iterating the argument

2. $P \xrightarrow{\alpha} P'$ is justified by a derivation using the transition rules. We prove the claim by induction on the depth of the derivation:
   - consider every possible case for the last step of the derivation
   - use the induction hypothesis on the premises for the needed assumptions.

For illustration we consider some cases, the other cases are analogous.

CASE Assume the last step of the derivation for $P \xrightarrow{\alpha} P'$ is an application of rule L-PAR. Then P is a parallel composition $P_1 \| P_2$ and $P' = P_1' \| P_2$. Now assume $P = P_1 \| P_2 \equiv Q$ is proven by using a single application of a str. cong. law. We consider two cases for illustration:

CASE we used commutativity of $\|$, then $Q = P_2 \| P_1$.
We can use R-PAR then to deduce

$$Q \xrightarrow{\alpha} P_2 \| P_1' = Q' \equiv P'$$

CASE we had $P_1 \equiv Q_1$ and used congruence to show $P = P_1 \| P_2 \equiv Q_1 \| P_2 = Q$.
Since $P_1 \xrightarrow{\alpha} P_1'$ is inferred using a shallower derivation, we can apply the induction hypothesis and obtain a Q_1' s.t. $Q_1 \xrightarrow{\alpha} Q_1' \equiv P_1'$.
Now $Q \xrightarrow{\alpha} Q' = Q_1' \| P_2$ is justified by L-PAR, and $Q_1' \| P_2 \equiv P_1' \| P_2 = P'$. $\square$

Now we are ready to show that reactions $\rightarrow$ coincide up to $\equiv$ with $\xrightarrow{\tau}$ tau transitions.

Theorem $P \xrightarrow{\tau} \equiv P'$ if and only if $P \longrightarrow P'$
$(\exists P'' \equiv P' : P \xrightarrow{\tau} P'')$

Proof ($\Leftarrow$) By induction on derivation depth for $P \longrightarrow P'$ with cases for each possible last step:

CASE TAU_R: $P = \tau.P' + M$ then by SUM $P \xrightarrow{\tau} P'$

CASE REACT_R: can be replicated by using SUM twice and then REACT

CASE PAR_R: by induction hypothesis + L-PAR

CASE RES_R: similar to PAR_R

CASE STRUCT_R: follows from $\equiv$ being a strong bisimulation

($\Rightarrow$) We can prove the stronger implication $P \xrightarrow{\tau} P' \Rightarrow P \rightarrow P'$ by induction on the depth of the derivation for $P \xrightarrow{\tau} P'$.

CASE SUM: $P = M + \tau.P' + N \equiv \tau.P' + (M+N)$
so by STRUCT_R + TAU_R we can derive $P \rightarrow P'$

CASE REACT: We first observe that if $P \xrightarrow{\lambda} P'$
then $P \equiv \nu\vec{x}((\lambda.Q + M) \parallel R)$ with $\lambda, \bar{\lambda} \notin \vec{x}$
and $P' \equiv \nu\vec{x}(Q \parallel R)$.
This can be shown again by case analysis.
Then we get $P = P_1 \parallel P_2$, $P_1 \xrightarrow{\lambda} P_1'$, $P_2 \xrightarrow{\bar{\lambda}} P_2'$
and $P' = P_1' \parallel P_2'$ so for $i = 1,2$ $\lambda_1 = \bar{\lambda}_2$

$$P_i \equiv \nu\vec{x}_i((\lambda_i.Q_i + M_i) \parallel R_i)$$
$$P_i' \equiv \nu\vec{x}_i.(Q_i \parallel R_i) \text{ with } \lambda, \bar{\lambda} \notin \vec{x}$$

so by STRUCT_R and REACT_R $P_1 \parallel P_2 \rightarrow P_1' \parallel P_2'$

OTHER CASES follow easily from induction hypothesis. □

We now state some easy properties without proof.

Proposition
a) Given P, there are only finitely many P', α s.t. $P \xrightarrow{\alpha} P'$
b) If $P \xrightarrow{\alpha} P'$ then $fn(P') \cup fn(\alpha) \subseteq fn(P)$
c) If $P \xrightarrow{\alpha} P'$ and σ is a substitution then $P\sigma \xrightarrow{\alpha\sigma} P'\sigma$

Theorem Let $\mathcal{B}$ be a strong bisimulation equivalence over $(S, \rightarrow)$.
The quotient of S under $\mathcal{B}$ is the LTS $(S/_{\mathcal{B}}, \rightarrow/_{\mathcal{B}})$
where $S/_{\mathcal{B}} = \{[s]_{\mathcal{B}} \mid s \in S\}$ and $[s]_{\mathcal{B}} \xrightarrow{\alpha}/_{\mathcal{B}} [t]_{\mathcal{B}}$ if $s \xrightarrow{\alpha} t$.
We have $s \sim [s]_{\mathcal{B}}$.

Proof See EXERCISE SHEET 9, PROBLEM 1

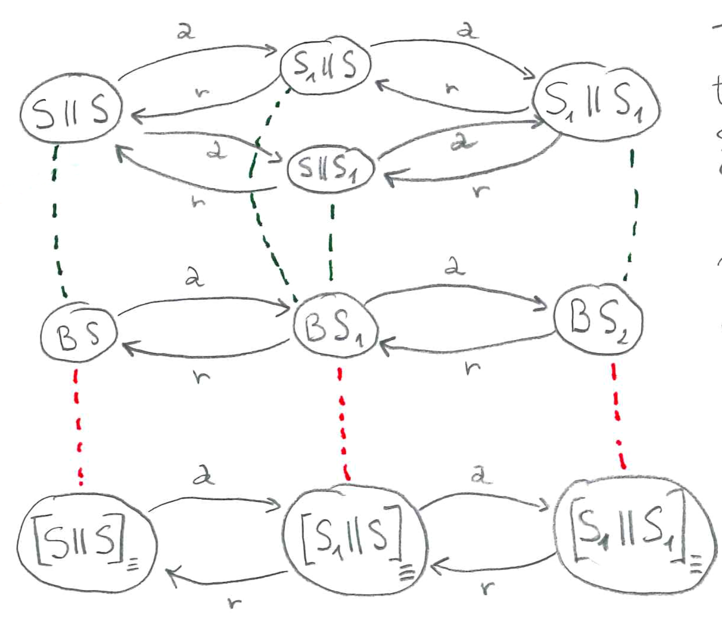
Since $S/_{\mathcal{B}}$ is usually a much smaller space, it is usually convenient to do bisimulation proofs in the quotient.
A common instance of this is by using $\equiv$ to quotient, since $\equiv$ is a bisimulation. Then the bisimulation proof carried in the quotient is called 'bisimulation up-to-$\equiv$'.

Example An n-ary semaphore is a kind of synchronisation mechanism that allows at most n components to acquire the semaphore. When a component tries to acquire a 'full' semaphore then it is blocked until some other component releases the semaphore.
Here is an implementation of a unary semaphore S (i.e. a lock) a binary semaphore BS

$$S := a.S_1 \qquad\qquad BS := a.BS_1$$
$$S_1 := r.S \qquad\qquad BS_1 := a.BS_2 + r.BS$$
$$BS_2 := r.BS_1$$

a ⇝ acquire
r ⇝ release

We want to show that a binary semaphore is nothing more than two unary semaphores: $S \| S \sim BS$

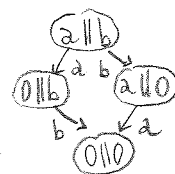


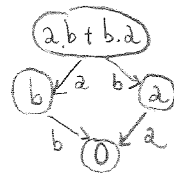
The green relation is the bisimulation
$\{(S\|S, BS), (S_1\|S, BS_1), (S\|S_1, BS_1), (S_1\|S_1, BS_2)\}$

By considering the quotient we remove the redundancy and need a smaller relation
$\{(S\|S, BS), (S_1\|S, BS_1), (S_1\|S_1, BS_2)\}$
which is now a bisim. up to $\equiv$, which proves
$S \| S \sim BS$

Algebraic Laws of Concurrency

Until now we carried out bisimulation proofs by working on LTS. We develop in this section some general bisimulation results that will allow us to establish bisimulation by algebraic manipulation of terms.





First we observe that there is a tight relation between sequential processes and general processes, for example $a \parallel b \sim a.b + b.a$ i.e. the parallel composition can be sequentialised. This is a general phenomenon:

<u>Theorem</u> For any CCS process P, we have $P \sim \sum \{ \alpha.Q \mid P \xrightarrow{\alpha} Q \}$

So every process, regardless of how concurrent its implementation is, is indistinguishable to a sum. We refine the result by characterising the shape of the sum:

<u>Theorem</u> EXPANSION LAW (aka STEP LAW)

$$\nu\vec{a}\,(P_1 \parallel \ldots \parallel P_n) \sim \sum \{ \alpha.\,\nu\vec{a}.\,(P_1 \parallel \cdots \parallel P_i' \parallel \cdots \parallel P_n) \mid P_i \xrightarrow{\alpha} P_i',\ \alpha, \overline{\alpha} \notin \vec{a} \} +$$
$$+ \sum \{ \tau.\,\nu\vec{a}.\,(P_1 \parallel \cdots \parallel P_i' \parallel \cdots \parallel P_j' \parallel \cdots \parallel P_n) \mid P_i \xrightarrow{a} P_i',\ P_j \xrightarrow{\overline{a}} P_j' \}$$

<u>Proof</u> is by easy induction and case analysis

The first summation accounts for actions visible to the environment, i.e. the ones not hidden by a restriction. The second summation accounts for the internal reactions (τ actions), which can happen regardless of restrictions.

<u>Example</u> $\nu r.(r.C[r] + a \parallel \overline{r}) \sim \tau.\nu r.C[r] + a.\nu r.\overline{r}$

Now, we also have $\nu r.\overline{r} \sim 0$ (which is also an instance of the expansion law) and we would like to use this fact to conclude $\nu r.(r.C[r] + a \parallel \overline{r}) \sim \tau.\nu r.C[r] + a$

However to be able to do so we need to make sure that the $\tau.\nu r.C[r] + a.[\,]$ context preserves bisimulation.

<u>Theorem</u> Strong bisimilarity $\sim$ is a process congruence

That is, if $P \sim Q$ then

① $\alpha.P + M \sim \alpha.Q + M$
② $\nu a.P \sim \nu a.Q$
③ $P \| R \sim Q \| R$
④ $R \| P \sim R \| Q$

<u>Proof</u> see EXSHEET 10 Problem 1

Equivalently, $P \sim Q$ implies $C[P] \sim C[Q]$ for every context C

<u>Note</u> $M + [\;] + N$ is <u>NOT</u> a context, you need to place the hole under a prefix $\alpha.[\;]$!

The previous theorem is extremely powerful. At the <u>conceptual</u> level, it is saying that $\sim$ is truly adequate as a behavioural equivalence because when the environment is formalised as a context, no environment can reveal the difference between two equivalent processes.
Similarly, there is no "experiment" we can do to distinguish two equiv. proc.
At the <u>practical</u> level, the result gives us a powerful tool to do bisimilarity proofs "compositionally": we can replace subcomponents of a system with "simpler" bisimilar components and transform gradually one term into another which is bisimilar.

Strong bisimilarity is nice but unfortunately too strong for comparing processes that differ only by some intermediate τ steps.
An example is $Count_0$ and C_0: both obviously implement a counter but $Count_0 \not\sim C_0$ because C_0 does an extra τ step after a decrement.
We have to be careful though: not all τ actions can be "ignored" for instance, in the coinflip example, the τ actions were the only difference between an external and an internal choice.
A candidate relaxation of $\sim$ should then relate $Count_0$ and C_0 but not $\tau.head + \tau.tail$ and $head + tail$.

Def (Weak Simulation)

a) Define $P \Rightarrow Q$ to mean $P \xrightarrow{\tau} \dots \xrightarrow{\tau} Q$ (ie. $\Rightarrow \; = \; \xrightarrow{\tau}{}^{*}$)

b) Let $s = \alpha_1, \dots, \alpha_n$ be a sequence of actions

Define $P \overset{s}{\Rightarrow} Q$ to mean $P \Rightarrow \xrightarrow{\alpha_1} P_1 \Rightarrow \xrightarrow{\alpha_2} \dots \Rightarrow \xrightarrow{\alpha_n} P_n \Rightarrow Q$

So $\overset{\alpha}{\Rightarrow} \; = \; \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*}$ and $\overset{\tau}{\Rightarrow} \; = \; \xrightarrow{\tau}{}^{+}$

c) A binary relation $\mathcal{S}$ over S is a <u>weak simulation</u> if whenever $P \mathcal{S} Q$ we have

$$\forall P' \, \forall s \in Act^{*} \text{ if } P \overset{s}{\Rightarrow} P' \text{ then } \exists Q' : Q \overset{s}{\Rightarrow} Q' \text{ and } P' \mathcal{S} Q'$$

The definition of weak simulation coincides with the notion of strong bisimulation if instead of the LTS $(S, \rightarrow)$ one considers the LTS $(S, \overset{s}{\Rightarrow})$. This clarifies the intent of the definition: we now consider observable s up to intermediary τ transitions.
The current definition is however very cumbersome to use because of the very heavy quantification $\forall s \in Act^{*}$.
We therefore give an alternative characterisation that only places conditions on single actions.

<u>Theorem</u> A relation $\mathcal{S}$ is a weak simulation if and only if when $P \mathcal{S} Q$ then

① if $P \xrightarrow{\tau} P'$ then $\exists Q' : Q \Rightarrow Q'$

② if $P \xrightarrow{\alpha} P'$ then $\exists Q' : Q \overset{\alpha}{\Rightarrow} Q'$

<u>Def</u> A binary relation $\mathcal{B}$ over S is a <u>weak bisimulation</u> if both $\mathcal{B}$ and $\mathcal{B}^{-1}$ are weak simulations.
We say s is weakly bisimilar to t (written $s \approx t$) when there exists a weak bisimulation relating s and t.

Many results that we saw for strong bisimulation also hold for $\approx$

<u>Theorem</u> The relation $\approx$ is the largest weak bisimulation relation and it is an equivalence.

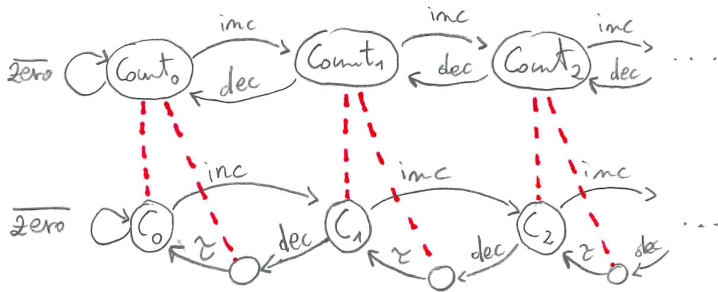
<u>Theorem</u> If an equivalence R is a strong bisimulation then $s \approx [s]_R$

<u>Theorem</u> Every strong bisimulation is also a weak bisimulation

<u>Corollary</u> $P \sim Q \Rightarrow P \approx Q$

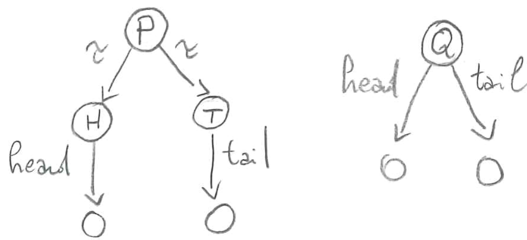
The fact that $\equiv$ is a strong bisim. allows us to do weak-bisimilarity proofs up to $\equiv$. Actually we can also do weak bisim. proofs up to $\sim$.

<u>Example</u> $Count_0 \approx C_0$



The <u>red</u> relation is a weak bisimulation

$P = \tau.head + \tau.tail \not\approx head + tail = Q$



The only hope to define a weak bis. would be given by being able to relate Q and H and T but H cannot reply the tail action available from Q and the same is for T and head.

Now, similarly to what we did for strong bisim., we want to establish some general lemmas enabling algebraic proofs of weak bisimilarity.

Importantly we have that weak bisim. is a congruence too

<u>Theorem</u> The relation $\approx$ is a process congruence.

<u>Proof</u> Analogous to the strong case.

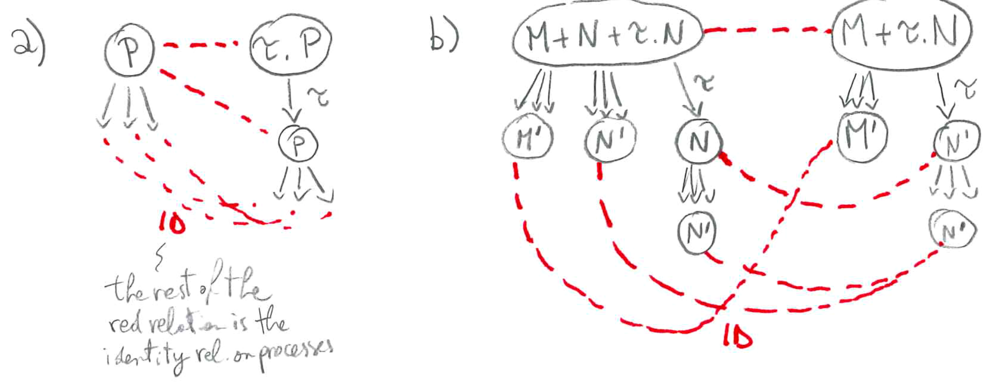
Lemma For any process P, sums M, N and actions α:

a) $P \approx \tau.P$

b) $M + N + \tau.N \approx M + \tau.N$

c) $M + \alpha.P + \alpha.(\tau.P + N) \approx M + \alpha.(\tau.P + N)$

Proof It is easy to produce weak bisimul. relations showing the claims. We illustrate the first two cases.



Now we have a powerful arsenal to use for weak bisimilarity proofs:

- we can use the EXPANSION LAW (because $P \sim Q \Rightarrow P \approx Q$)
- we can use ≡ to replace structurally congr. processes
- we can use process congruence to replace subterms (when in contexts)
- we can use the above lemma

Example $\nu c.(a.\bar{c} \parallel b.\bar{c} \parallel c.c.P) \approx a.b.P + b.a.P \quad (c \notin fn(P))$

steps

① $\nu c.(a.\bar{c} \parallel b.\bar{c} \parallel c.c.P) \sim a.(\nu c.(\bar{c} \parallel b.\bar{c} \parallel c.c.P)) + b.(\nu c.(a.\bar{c} \parallel \bar{c} \parallel c.c.P))$

② $\sim a.(b.\underbrace{\nu c.(\bar{c} \parallel \bar{c} \parallel c.c.P)}_{\sim \tau.\tau.\nu c.P \equiv \tau.\tau.P \text{ by } c \notin fn(P)} + \tau.\underbrace{\nu c.(b.\bar{c} \parallel c.P)}_{\sim b.\tau.\nu c.P \equiv b.\tau.P}) + b.(a.\underbrace{\nu c.(\bar{c} \parallel \bar{c} \parallel c.c.P)}_{\sim \tau.\tau.P} + \tau.\underbrace{\nu c.(a.\bar{c} \parallel c.P)}_{\sim a.\tau.P})$

③ $\sim a.(b.\underbrace{\tau.\tau.P}_{\approx P} + \tau.b.\underbrace{\tau.P}_{\approx P}) + b.(a.\underbrace{\tau.\tau.P}_{\approx P} + \tau.a.\underbrace{\tau.P}_{\approx P})$

④ $\approx a.(\underbrace{b.P + \tau.b.P}_{\approx b.P}) + b.(\underbrace{a.P + \tau.a.P}_{\approx a.P}) \approx a.b.P + b.a.P$

① is by expansion law, ② by exp law ×2 + congruence ③ used exp. law + ≡ on the bracketed expressions + congruence to replace them ④ is the first step where we do not have strong bisim. We used the lemmas for weak b. on the subterms and then congruence. We proved the eq. without considering the LTS!

Historical Note

The formal study of concurrency using Process Algebra started in the '70s when most mathematical understanding of concurrency was based on Petrinets (introduced by Petri in 1962). Robin Milner had a huge rôle in establishing the field introducing CCS. His approach starts by defining an abstract model of computation and studying the notion of behaviour on it.
A very influencial complementary approach was proposed roughly in the same period (one might say "concurrently"!) by Tony Hoare, who introduced CSP, "Communicating Sequential Processes". CSP is a language somewhat similar to CCS but with a crucial difference: its semantics is not specified in terms of LTS. In other words Hoare did not specify a model for concurrent behaviour. Instead he adopted what is commonly referred to as the axiomatic semantics approach: the "meaning" of a program is given implicitly in terms of which equivalences are satisfied by the program. One might say, for example, that the meaning of a step is given in CSP by the EXPANSION LAW.
The EXPANSION LAW becomes thus an axiom in CSP, while it is a theorem in CCS. These two approaches highlight different aspects of the same phenomenon and are both useful: there have been works that axiomatise CCS, and that provide models for CSP.

For more on this topic see

J. C. M. Baeten, 2005
"A brief history of Process Algebra"
Theoretical Computer Science, Vol 335, Elsevier.

Addendum on Bisimulation and Coinduction

While Concurrency Theory is still a young discipline with a large number of competing mathematical frameworks, the concept of BISIMULATION is one of the most natural and stable tools that the discipline produced.

The goal of this short note is to demonstrate its relevance also outside of concurrent systems analysis.

Consider the following HASKELL definitions

```
filter f (x:xs)
  | f x       = x : filter f xs
  | otherwise = filter f xs

ints n = n : ints (n+1)

ones = 1 : ones

bla n = filter odd (map (mod 2) (ints n))
```

> This is called "stream programming": Thanks to lazy evaluation these programs can manipulate infinite lists (provided the overall output only depends on a finite portion of them).

Now we would like to state that ones is the same stream as (bla 1) but certainly we cannot state it like (ones == bla 1) because checking equality would need to examine both infinite lists in full, leading to a non terminating computation.

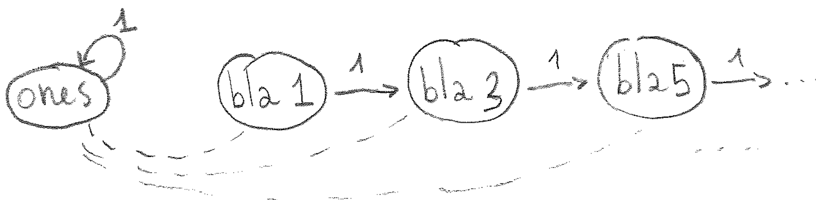
One way to solve the problem is by interpreting a stream $x_1 : x_2 : x_2 \ldots$ as a LTS $o \xrightarrow{x_1} o \xrightarrow{x_2} o \xrightarrow{x_3} \cdots$ and use bisimulations to relate the streams:

$\text{ones} \xrightarrow{1} \text{ones}$

$\text{bla } n \xrightarrow{1} \text{bla } (n+1) = \text{bla } (n+2)$

A bisimulation showing ones ~ bla 1 is

$\mathcal{B} = \{ (\text{ones}, \text{bla } n) \mid n \in \mathbb{N} \}$



These kind of proofs are called 'proofs by coinduction' to highlight the fact that the bisimulation proof principle can be seen (in a very formal sense) as the dual of the induction principle.

For the interested reader:

B. Jacobs, J. Rutten, 1997
"A tutorial on (Co)Algebra and (Co)Induction"
EATCS Bullettin, Vol. 62.

As a last remark, a common way to prove ones ~ bla 1 is by using what is sometimes called a "Take Lemma" which has the form

TAKE LEMMA: $\forall k$: take k ones == take k (bla 1)

which now is well defined because (take k l) returns the first k elements of the list l.

The attentive reader would see that the take lemma is nothing but STRATIFIED BISIMULATION in disguise: instead of checking that ones ~ bla 1 we check $\forall k$: ones $\sim_k$ bla 1.

Note that the two statements are equivalent only in some cases, for LTS, the needed assumption is finite branching.